The Pragmatic Programmers

Swift Style

Second Edition

An Opinionated Guide to an Opinionated Language

Erica Sadun

edited by Brian MacDonald

Early Praise for *Swift Style, Second Edition*

This is one of the most important resources on the language. A must-have on every Swift programmer's desk.

➤ **Paris Buttfield-Addison**
Founder, Secret Lab Pty. Ltd.

Swift Style, Second Edition is an opinionated style guide that gives readers a clear road map of best practices they should be following in their own projects. This is a must-have for any Swift development team, as it touches subjects and arguments for styling that can dramatically improve any project.

➤ **Grant Isom**
Team Lead, Associate Senior Software Engineer, Cerner Corp.

One of the books I always have with me, this gold mine taught me to be a better programmer, collaborator, and critic of Swift code. Also great for winning arguments.

➤ **Mars Geldard**
Author, *Practical Artificial Intelligence with Swift*

Style is inherently subjective, so it is unlikely you will agree with all of the recommendations this book contains. Fortunately, that is not the goal of *Swift Style, Second Edition*. Instead, it takes a close look at all of the details and considerations that go into intentionally and thoughtfully choosing your own personal or house style.

➤ **Matthew Johnson**
Owner and Principal Software Engineer, Anandabits, LLC

We've left this page blank to make the page numbers the same in the electronic and paper books.

We tried just leaving it out, but then people wrote us to ask about the missing pages.

Anyway, Eddy the Gerbil wanted to say "hello."

Swift Style, Second Edition

An Opinionated Guide to an Opinionated Language

Erica Sadun

The Pragmatic Bookshelf

Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

Swift and the Swift Logo are trademarks of Apple, Inc. and are used by permission. *Swift Style, Second Edition: An Opinionated Guide to an Opinionated Language* is an independent publication and has not been authorized, sponsored, or otherwise approved by Apple, Inc.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Brian MacDonald
Copy Editor: Nicole Abramowitz

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2019 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-627-3
Book version: P1.0—March 2019

This book is dedicated to the Swift community
both inside and outside Apple.

Contents

Welcome to *Swift Style, Second Edition*

This book offers a practical, powerful, and opinionated guide to coding style. It incorporates a multitude of best practices, guiding you to work successfully in this equally opinionated programming language.

This book's central theme is this: it's not just the compiler that needs to read your code. Code is more often read than written. Its crafting involves a process of review and clarification apart from developing algorithms and ensuring programmatic correctness. The better you style your code, the easier it becomes to understand and maintain. Well-structured code is an essential investment in future-proofing your work.

Since this book was first published, Swift hasn't stood still. This second edition retains the advice and insight from the first. I have revised examples to showcase an evolved Swift language, and some topics have expanded to accommodate Swift's ever-growing feature list. These pages offer you a treatise on style where the quaintness of some examples won't get in the way of your reading. This edition offers a broader discussion that follows the shape of the updated language.

Swift changed a little, but its style and this book's philosophy have not.

How This Book Got Here

This book didn't start out as a book. It began its life as a blog post ("Swift: Don't do that"[1]) and then as a GitHub repository.[2] I was inspired by one of Natasha Murashev's posts about the new open source SwiftLint[3] project, developed by JP Simard[4] and the folks at Realm.[5] At the time, Murashev wrote

1. http://ericasadun.com/2015/05/05/swift-dont-do-that/
2. https://github.com/erica/testlint
3. https://github.com/realm/SwiftLint
4. https://github.com/jpsim
5. https://github.com/realm

a weekly curated Swift newsletter that covered news and articles about the language.

At that time, SwiftLint was built on what can be kindly described as extreme hacking by hooking into Swift's then-private SourceKit services. As I wrote,[6] I loved the idea behind SwiftLint, but I needed a better tool for immediate use. I wanted something good enough for real code, even in imperfect form. It had to run from the command line and work with playgrounds. It had to be able to scan files added to Xcode projects by reference as well as those physically stored in the project root. Leveraging some of the code I had developed for a much earlier `Xcode Manifest` application, I built an Objective-C scanner[7] with simple matching rules based on regular expressions.

My linter worked line by line instead of parsing because I didn't want to use interprocess hacking. (Its implementation relied heavily on regular expression matching, waving my hands, and sacrificing chicken entrails.) It wasn't a great linter, but it was an amazing way to start thinking about the way I do and should write Swift. Over time, my rule base grew and grew as my interest evolved from "How do I automate simple code checks?" to "What are the right things to check in my code?"

I kept working to incorporate rules that I found valuable. The project quickly evolved from a project about linting into a way I could *explore* style rules for this new language. As I developed my style preferences, I found myself talking to Swift developers, asking for opinions and feedback, and incorporating their experiences into my writing.

My style sheet kept growing, maturing from a few pages of directives to a pamphlet to a small book. At a certain point, I realized I had something of more general utility than a personal style guide on my hands. And that was when I approached The Pragmatic Bookshelf about the possibility of transforming my material into a proper book: the book you are now reading.

From the time I started work on this project (way back in Swift 1!) to today, Swift settled down. It decided exactly what kind of language it was going to be, and it has finally realized and expressed that language in a much more stable form. A significant developer base has adopted the language, and these developers now have several years of Swift development experience behind them. Swift moved from the turbulence between Swift 2 and Swift 3 into calmer waters, introducing primarily additive features in Swift 4 and Swift 5.

6. http://ericasadun.com/2015/05/18/swift-alternative-lintage/
7. https://github.com/erica/testlint

The language became stronger and more reliable. It found its core philosophy and developed a firm opinion of what it was and what it wanted to be.

Today's Swift opens itself to evaluation, curation, and guidance. It allows you to weigh preferred coding styles even though the community as a whole has not yet adopted fixed or even universal conventions. You can finally build house guidelines without having the rug pulled out from under you every few months. Swift has *arrived*.

This book is written for Swift coders both new and experienced. The material in these pages ranges from "Captain Obvious" good practices to power strategies, spanning the full range between those points. I solicited opinions and preferences along the way, taking into account varied experience, programming domains, and conventions, to build a strong base of practical and practicing advice.

There are always ways to enhance your code and coding practices. While code can have issues, those issues should be evaluated with respect to strong design principles, not odor. Smell the roses, not the code, and embrace your personal style.

What's in This Book

Apple's Swift programming language has finally reached a level of stability that opens its use to a much wider audience. *Swift Style, Second Edition* guides you through the ins and outs of at least part of Swift programming's best practices. This book is written for Swift programmers both experienced and new to this language who want to explore the art of crafting code in this language.

Code style matters. Critical dos and don'ts for writing readable Swift code guide you to better craftsmanship. This book explores common coding challenges and the best practices that address them. From spacing, bracing, and semicolons to proper API style, discover both *what* you should do and the *whys* behind each recommendation.

A style guide establishes a consistent experience of well-crafted code that lets you focus on the code's underlying meaning, intent, and implementation. This book explores the areas of Swift where coding structure comes into play. Whether you're developing a personal style or a house style, there are always ways to enhance your code choices. You'll find here the ideas and principles to establish or evolve your own best style practices.

Begin with simple syntactical styling. Strengthen code bracing for easy readability. Style your closures for safety and resilience. Perfect your spacing and layout. Master literal initialization and typing. Optimize control flow layout and improve conditional style choices. Transition from Objective-C and move code into Swift the right way. Improve API design using proper naming and labeling. Elevate defaulted arguments and variadics to their right places. *Swift Style, Second Edition* covers it all.

Here's a chapter-by-chapter breakdown of what you'll find in this book:

- Chapter 1, Adopt Conventional Styling, on page 1: Conventional style is like a window. It allows you to look through to see the code intent or landscape that lies beyond it. When there's a streak on the window or an unconventional use in code, programmers naturally fixate on the wrong thing. This chapter explores common house-style elements. It surveys Swift language features you'll want to standardize and lock down for better and more consistent code.

- Chapter 2, Structure Your Code for Readability, on page 35: Swift code faces wrapping challenges you don't encounter in C-like languages. Swift's top-heavy style of output magnifies the importance of braces, wrapping, and line composition. This chapter explores ways you choose to build coding backbones. You'll learn to enhance code readability and emphasize code meaning and design intent.

- Chapter 3, Establish Preferred Practices, on page 73: Like any language, there's always more than one way to achieve your goals in Swift. Making good choices is a critical part of coding style. This chapter surveys typical design decision points and guides you through refactoring opportunities. Step beyond simple linting to explore the architectural points that affect your development.

- Chapter 4, Design the Right APIs, on page 139: An API establishes a contract for calling code. It describes the types, methods, and results produced by an implementation and how these will behave. A well-designed API provides a clear and understandable set of tools, with well-chosen names and thoughtful consideration to resilience and long-term code evolution. Learn the "Swifty" ways to design APIs by leveraging the principles of clarity, concision, and utility. From access control to naming, from nesting to defaults, this chapter shows you how to present your functionality safely and meaningfully.

- Chapter 5, Look to the Past and the Future, on page 197: Time plays a key role in Swift style. Seeing Swift in this larger scope is important in both adopting new practices and supporting code beyond its moment of creation. This short chapter explores time, guiding you into routines that enhance, document, and support the Swift code you're now writing. You'll learn ways you can best move on from your code's past, and discover how you will support its future.

- Chapter 6, Good Code, on page 217: This book wraps up with a short meditation about what good code *means* and how you can recognize what good coding *is*.

Contributing to This Book

When reading through this volume, if you see something that's missing or that's wrong, or if you just want to suggest an alternative viewpoint, drop me a note at erica@ericasadun.com and I will consider it for a future update. Thank you in advance for being part of this effort.

Online Resources

This book has its own web page,[8] where you can find more information about the book, and you can help improve the book by reporting errata, including content suggestions and typos.

Other valuable online resources include these:

Apple's Swift resources page[9] provides links to essential Swift and Xcode materials, including iTunes U courses, videos, sample code, and a link to the official Swift Programming Series books.

The Swift.org[10] website offers a central hub for the Swift programming language. It hosts a blog,[11] which provides regular updates about topics important to those who use Swift as part of their daily coding work. You'll find announcements about language releases, the tools that support language development, new language features, and more. Apple's original Swift blog[12] is no longer maintained.

8. https://pragprog.com/book/esswift2/swift-style
9. https://developer.apple.com/swift/resources/
10. https://swift.org
11. https://swift.org/blog
12. https://developer.apple.com/swift/blog/

Swift's core documentation can now be found at docs.swift.org.[13] This includes the current version of *The Swift Programming Language* and links to the latest *API Design Guidelines* and *Migration Guides*.

The Swift Programming Language is a free ebook offered by Apple on docs.swift.org, on the Apple Books store, and on its developer website.[14] It offers the authoritative Swift reference, including a guided tour to the language and a formal reference. The book is available under the Creative Commons Attribution 4.0 International License (CC BY 4.0), and volunteers are sought to translate it to other languages.

A second Apple ebook, *Using Swift with Cocoa and Objective-C*, is just as essential. This ebook[15] provides an overview of the topics related to interoperability between the two languages and the details of how API calls straddle the two. It's a much shorter volume than *The Swift Programming Language* because of its tight focus.

Apple's online Swift Standard Library Reference[16] adds an indispensable overview of Swift's base functionality layer. It offers an overview of fundamental data types, common data structures, functions, methods, and protocols. If you stop learning Swift at the language basics, you'll be missing out on this critical portion of core language expressiveness.

Join in at forums.swift.org.[17] This site offers user forums where you can discuss the language with others. Reserved discussion areas include language announcements, topics related to the Swift Evolution process, the development and implementation of Swift, user-to-user support, and projects related to Swift including SwiftLint,[18] Kitura,[19] Vapor,[20] SourceKitten,[21] and more.

Sadly, the official Apple Developer Forums[22] are underused. Most people join a variety of Slack and IRC discussions for peer support. The official Swift user forums are an excellent resource. The conversation, at least at this time, tends to be thoughtful and the traffic is light, with good access to high-level Swift

13. https://swift.org/documentation/
14. https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language
15. https://developer.apple.com/library/content/documentation/Swift/Conceptual/BuildingCocoaApps/index.html
16. https://developer.apple.com/library/ios/documentation/General/Reference/SwiftStandardLibraryReference/index.html
17. https://forums.swift.org
18. https://github.com/realm/SwiftLint
19. https://www.kitura.io
20. https://github.com/vapor/vapor
21. https://github.com/jpsim/SourceKitten
22. https://forums.developer.apple.com/community/xcode/swift

developers. As with any language development forum, this one is meant to discuss language features with other users and is not for debugging support.

File your bugs at bugs.swift.org.[23] This open bug-reporting system enables you to submit bugs, track them, search for topics, and more. There's no need for workarounds like Tim Burks' Open Radar (site[24], repo[25]) to struggle against Apple's internal opaqueness. Each bug is visible, readable, and trackable. Submitting a bug report for Swift is completely different from Apple's black hole. You can enact change that you'll be able to see and follow.

Track Swift's progress toward ABI stability at the swift.org ABI Dashboard.[26] The Swift ABI Stability Manifesto[27] describes the tasks needed to complete the process for ABI stability, and the dashboard lists the tasks, tracking issues, and status of this ongoing project.

The swift.org APIs project also has its own page,[28] supporting Swift for server-side development. Core capabilities for networking and security allow the creation of frameworks and server applications using pure Swift code.

The core Swift repository[29] provides public access to the Swift language. Anyone with an interest to do so can download a copy of the Swift source code and build the Swift compiler for a variety of platforms. If you're so inclined, you can make changes to the Swift source, whether bug fixes or enhancements, and submit pull requests with those changes.

If you'd like to keep on top of the language and its changes, I post regular updates about Swift and developer tools on Twitter[30] and at my personal website.[31]

Language Evolution

When Swift first went open source, developers were invited to take part in bug fixes and language design. Over 200 people outside of Apple began to work on the Swift programming language and contributed to the first open

23. https://bugs.swift.org
24. http://openradar.appspot.com
25. https://github.com/timburks/openradar
26. https://swift.org/abi-stability/
27. https://github.com/apple/swift/blob/master/docs/ABIStabilityManifesto.md
28. https://swift.org/server-apis/
29. https://github.com/apple/swift
30. http://twitter.com/ericasadun
31. http://ericasadun.com

source release, Swift 2.2, in March 2016. The number of Swift contributors is now in the mid to high hundreds.

Although many initial changes were little more than typo corrections (essentially low-hanging fruit), the scope and sophistication of external contributions has grown. Many new Apple employees started as open source contributors. If you're interested in participating in Swift language design, they use a strict internal style, which is touched on in this book.

The Swift Evolution repository[32] hosts an archive of proposals affecting user-visible enhancements for the Swift programming language. User-facing changes must proceed through the evolution process, which is chaired by a core team of language experts before being adopted into the language.

Its associated github.io page[33] provides an easy-to-consume list of proposals, including those in active review, those that have been accepted, and more. This Swift Evolution overview page goes in reverse order, with newer proposals listed at the top. This is where you find items that are just now being considered.

As you scroll down, you'll find accepted proposals, proposals that have been returned for revision, and deferred, withdrawn, and rejected proposals. I hold the current record for most-rejected proposals for the Swift programming language. And I still think that adding trailing commas in parameter lists and tuples is an excellent enhancement, especially since the language supports defaulted arguments.

Some proposals have been accepted but not implemented, such as SE-0068,[34] which expands the use of uppercase `Self` to class members and value types, as you would use it in Objective-C. It is proposals like this one that led Swift Evolution to adopt a rule that you must provide a working implementation *before* a proposal can be reviewed. Just because a proposal is a great idea does not mean that it is implementable.

Swift's change log[35] offers details about what has changed in each release, including releases that have not yet been distributed through Xcode. If you'd like to peek at upcoming features, the log is updated as each new feature is implemented and added to the master language branch.

32. https://github.com/apple/swift-evolution
33. https://apple.github.io/swift-evolution/
34. https://github.com/apple/swift-evolution/blob/master/proposals/0068-universal-self.md
35. https://github.com/apple/swift/blob/master/CHANGELOG.md

Any change log entry that starts with the letters SR means that it is responding to a bug report. Any entry listed with SE indicates a Swift Evolution proposal that has been adopted and implemented into the language. Like the status page, the changes scroll from most recent to least, and go back further than the open source epoch. It's fascinating to delve into the language's design decisions that predate December 2015, when Swift was opened up to the world.

New language versions that have not been released publicly can be downloaded via a system of nightly releases. Visit the swift.org downloads page.[36] You'll find both current installers for macOS and Ubuntu Linux, along with development builds. Although official support for installers are limited to Mac and Linux, Swift has been ported to Windows' Linux subsystem, Android, FreeBSD, Raspberry Pi, and more.

Ready to Get Going?

You're about to take a deep dive into Swift style. This book explores the fine details of language use in a fascinatingly pedantic (and possibly diagnostically obsessive) fashion. This book won't change your understanding of Swift and it won't reduce the pain of writing and debugging your code, but it should take a lot of the pain out of reading and reviewing that code at some later date. This book offers new ways to think about Swift, to structure Swift, and to make good coding choices.

Thank you for purchasing this book. Let's get started!

Credits

Samples from the Swift.org open source project are licensed under Apache License 2.0 with Runtime Library Exception. They are copyright 2014–2016 Apple Inc. and the Swift project authors. See https://swift.org/LICENSE.txt for license information. See https://swift.org/CONTRIBUTORS.txt for the list of Swift project authors. Quotes from *The Swift Programming Language* book by Apple Inc. are reproduced according to the terms of the Creative Commons Attribution 4.0 International (CC BY 4.0) License: https://creativecommons.org/licenses/by/4.0/

Thanks

My thanks to everyone who helped provide technical information and feedback during the development of this book. I'm deeply indebted to my technical

36. https://swift.org/download/

review team: Lily Ballard(!), Tony Gray, Maurice Kelly, Stephen Wolff, Stefan Turalski, Peter Wood, Nick Watts, Alex Henry, Arun Kumar, Daivid Morgan, Derek Graham, Dianne Siebold, Gábor László Hajba, Kenneth Geisshirt, Nick McGinness, Peter Hampton, and Phillip Apley. Additional shout-outs go to Steve Streza, Mike Ash, Alex Kempgen, Seth Willets, Gwynne Raskind, Xiaodi Wu, Kenny Leung, Paul Cantrell, Sebastian Celis, Jeremy Tregunna, Nate Cook, Pyry Jahkola, Russ Bishop, Jacob Bandes-Storch, Soroush Khanlou, Mars Geldard, Tim Vermeulen, Ian Mateus Vieira Manor, Matthew Johnson, Andrew Bennett, Kevin Cramer, Adam Sharp, Connor Neville, Anthony Mattox, Alfonso Urdaneta, Brent Royal-Gordon, Zev Eisenberg, Mohamed Fouad, Cezary Wojcik, Seivan Heidari, Jessy and Catie Catterwaul, Dave DeLong, Rainer Brockerhoff, Brian King, Peter Livesey, Joe Fabisevich, Mark Rowe, Grant Isom, Olivier Halligon, Tim Nugent, Jeff Forbes, and Greg Titus. Thanks go out to the Swift Evolution and Swift Users' Group members and everyone at Apple who has provided inspiration, support, and insight, including but not limited to Chris Lattner, Joe Groff, Ted Kremenek, John McCall, David Smith, Doug Gregor, Dave Abrahams, Jack Lawrence, Brian Gesiak, and Jordan Rose.

Thanks also to my Pragmatic Team: Brian MacDonald, development editor and all-around great guy who managed and oversaw with patience, insight, and humor; Andy Hunt, publisher; and everyone else at The Pragmatic Bookshelf who helped make this book a reality.

A final, special, thank you to Allen Denison, who helped make it possible for the slug on the cover of the second edition to hold a coffee cup with an official Swift logo! It's a tiny touch that took a lot of work to make happen and I'm so grateful for his perseverance through so many levels of bureaucracy to support me.

CHAPTER 1

Adopt Conventional Styling

Conventional style is like a window. You look through to see the code intent or landscape that lies beyond. When there's a streak on the window or an unconventional use in code, programmers naturally fixate on the wrong thing. Well-written code, like a clean window, is invisible. People don't normally stop and say, "Wow, that's especially well-written code." They pay attention when styling choices have failed for some reason.

It's natural to notice odd style choices. Many brilliant and insightful blog posts play host to comment sections that are full of corrections concerning typos and grammar errors. This is the same impulse that distracts you when noticing a wad of spinach caught between a colleague's teeth.

Adhering to a convention style enables code to fade into the background. It allows readers to pull out the underlying meaning and programmatic direction of code statements rather than focus on the details that went into crafting them.

This chapter surveys Swift conventions. By focusing on ways to mechanically format code details, you subdue specific language details and enhance the expression and importance of the routines they describe. Explore common house conventions and survey Swift language features you'll want to standardize and lock down for better and more consistent code.

Reaching for Consistency

Your style may not match my style. Your best practices may not be mine. Every coder and development group have their own way of doing things. A codebase evolves a dialect drawn from the background, experiences, and requirements of its development participants.

You who are writing code, the future you who reads that code, your extended team, and anyone who visits and maintains your repositories are all first-class consumers of the code you write today. Code that's readable, consistent, and comprehensible lowers costs in terms of life span, maintainability, and error prevention.

Style is not about code correctness. Code may be poorly implemented or incorrect or fragile. That's why testing, debugging, and review play such critical roles in development. Style creates a consistent experience that enables any reviewer to abstract away a file's line-by-line layout and focus on the underlying meaning, intent, and implementation of that development. Style reduces the mental energy it takes to read code. It enables you to dive into a code's semantics with minimal friction.

I have my own set of rules. They're ever changing, flexible, and adaptable. They, like yours, depend on circumstance and utility. Any style that produces readable, maintainable, and well-documented code is, by definition, successful, but there is no "golden mean" in code style.

The Swift community has begun to settle on certain standards, not all in agreement, that create conventional coding styles. I've been collecting and distilling these conventions: first to create a personal style for myself and then later curating them into this book.

This chapter, and the ones that follow, don't offer canonical answers. They incorporate many *conventional* answers, which you're welcome to adopt, ignore, ridicule, or embrace. They explore the areas of Swift where structure comes into play. Whether you're developing personal style or house style, there are always ways to enhance your code choices.

Style guides should be firm, definitive, and prescriptive. This discussion isn't *the* ultimate Swift style guide and it can't be. I'm writing this material outside of the specifics that propel the needs of a particular project or mission. What you need is to incorporate good Swift style and conventions into your daily programming practices to create your *own* house guidelines and preferred practices.

Here are ideas and principles for you to draw from and adopt. Use these topics to establish and enhance house rules and then start living by the rules you created. The results will be cleaner, consistent, and readable code. Style, no matter how individual or corporate, how quirky or conventional, how personalized or typical, always creates an improved coding, reading, and maintenance experience.

Understanding Inattentional Blindness

Humans naturally suffer from inattentional blindness. This refers to the simple common failure to pick up your own typos or style issues because your brain is focused on higher-level development tasks. Inattentional blindness is a psychological lack of attention. It isn't due to human faults like inexperience or poor coding skills. It's simply a cognitive aspect of how people work.

Coding is a heavily weighted task. It demands full attention as you focus on problem-solving and expression. As you focus on building code and debugging, basically where you're resolving obstacles that stand between you and working code, you likely won't place equal weight on the minutia of punctuation or spelling.

The term "inattentional blindness" is most associated with psychologists Arien Mack and Irvin Rock. They claimed that you could not properly perceive a task without intent. Any portion of your coding that is done automatically, without *specific attention*, is one that is subject to personal habits and typographic quirks. In short, you can code or you can style, but you probably won't do both at once.

Most people add extra passes when coding. They may "code golf" to shorten and simplify their code. Simpler solutions can often be more robust or more elegant, but these solutions arrive through a process of refinement. Coders first make it work and then clean it up.

Style follows the same pattern. You may need a further pass or two through code to isolate and remedy your style choices. You can code and you can enhance and you can style, but these are all relatively separate tasks. Working on any one may make you blind to the others.

When you don't work incrementally, when you defer cleanup too long, style issues make it harder to simplify code. Code that's difficult to read and complicated to follow may prevent you from understanding what you were aiming for when you first wrote it. Good style lowers the barriers between code and comprehension.

Fortunately, many styling conventions are mechanical. A rule sheet allows you to scan code, looking for out-of-spec code use. It's better to automate the process. A linter is a programmatic utility that flags nonstandard code constructs. A good one enables you to enforce rules that might be missed or overlooked during coding and compilation. A linter doesn't solve all problems related to code style, but it provides a way to get out of your head and highlight simple problems you've overlooked.

Linters, like spelling and grammar checkers, offer practical solutions for inattentional blindness. They allow you to mandate, flag, and correct minor styling errors that would otherwise detract from your code. Unlike spelling checkers, which use standard English dictionaries, you can't effectively apply a linter before first deciding what your house style *is*.

That isn't to say you can't develop your style incrementally. Good linting tools also let you add and remove rules over time. But absent a decision adopting a rule or even adopting pre-built house style rules from third parties, there's nothing to work with and no style to apply.

Should you use spaces when declaring generics? Should you allow parallel variable declarations? There are no absolute answers. You decide what house style you want to adopt. Then, if using a linter, you can mechanize at least some part of the scan to detect when code output fails to match your house style. Many developers use the built-in, community-driven rule sets with Realm's SwiftLint and then tweak those rules as they see how their code builds against the automated warnings emitted by the tools.

Linters are limited by the rules they describe and the structural parsing that makes those rules enforceable. You can't mandate how generic declarations are laid out in code unless you or an automated tool can recognize what a generic declaration is. This is an easy task for a person. It's a harder task for an application. A linter must use parsing to recognize the role of each token in code so it can evaluate that use and emit any warnings.

Once automated, though, linters are far better than humans at consistently spotting errors because they do not have competing mental processes to overcome. Anyone who has used a spellchecker knows that the automated solution will usually (but not always) find issues that a human has, please forgive me, knot.

Style consists of a mix of "easy" problems (like "don't add spaces before commas") and hard ones ("is this API well named?"). The former lend themselves to linting. The latter cannot. Automate what you can so you have more time to devote to evaluating and correcting other issues in your personal style guide.

- Just because code compiles and executes correctly doesn't mean it's well styled.
- Separate coding tasks into separate passes.

- Styling is like spelling and grammar check. It prepares code for third-party viewing.
- Linters automate common style oversight.
- It's not only the compiler that needs to read your code.

Adopting Conventional Spacing

The Swift developer community has adopted spacing conventions for many language elements. The following overview reviews the most common spacing conventions used in Swift code and considers whether the spacing style is universal or not:

Semicolons

`statement; statement`

Consensus: left-hugging, space follows.

A substantial number of developers omit semicolons entirely, mandating separate line use as part of their in-house style. Those that embrace semicolons will use this conventional style, which is not limited to the Swift language.

Commas

`x, y`

Consensus: left-hugging, space follows.

Commas are commonly used to separate arguments and parameters. Special rules often apply when working with generic parameters.

Generic parameters

`Type<T, U>` vs. `Type<T,U>`

The consensus on generic parameters is split.

Some developers add spacing after the comma, allowing generic type parameters to breathe. This spacing is more significant when generic parameters have real names or there are more than two generic parameters being used. Spacing ensures that each parameter is more readable as a separate word—for example, `Either<Left, Right>` vs. `Either<Left,Right>`, or `compose<T, U, V>` vs. `compose<T,U,V>`.

Other developers omit spaces as a rule. When generic parameters are part of the return type, it may seem strange to break the return type into chunks by adding spaces. Omitting spaces ensures that IDEs like Xcode wrap the entire type including type name, angle brackets, and generic type parameters (for example, `ArrayBridge<Element,CChar>`) as a single chunk instead of allowing line breaks after the comma.

Some mix and match these ideas, using spaces for functional declarations (`arrayDownCastIndirect<SourceValue, TargetValue>`) while omitting them for types (`NativeDictionaryEnumerator<AnyObject,AnyObject>`).

The Swift standard library team more often uses spaces than not.

Generics

`Type<T>`

Consensus: connect the type to its opening angle bracket.

Protocol property requirements

`{ get set }` and `{ get }` vs. `{get set}` and `{get}`

Consensus: both the Apple and emerging community standard is to use spaces, although it's quite common to see braces hugging the keywords.

Binary operators

`a + (b * c)`, `a + b * c`, or `a + b*c`

Consensus: add single-space padding before and after for all binary operators. This isn't just a style issue. It's also "The Law."™

Minor consensus: use spacing to emphasize operator precedence when chaining operations together—for example, `a + b + 2*c + d`. I would never personally do this.

Return arrow tokens

`f() -> T`

Consensus: spaces on both sides. Spaced arrows are easier to read, allowing your eye to differentiate between the parameter list and the return type.

Ranges

`1 ... 3`, `1.2 ..< 1.5`

There is no universal consensus.

Minor consensus: prefer spaces on both sides. Adding spaces gives your code room to breathe and allows ranges (open or closed) greater clarity than their non-spaced alternatives.

Alternatively, parenthesize floating-point numbers in unspaced floating-point ranges—for example `(1.2)..<(1.5)`.

The standard library prefers a no-space approach, using parentheses when needed.

Unterminated Ranges

`1...`, `...5`

Universal consensus because this is, otherwise, a compiler error. Omit spacing when using one number to establish a range, as with any unary operator.

Empty constructs

`[]`, `[:]`, `{}`, `f()`

Consensus: no internal spaces.

Whitespace characters

`"\u{00A0}"`, "\u{1680}", "\u{2000}", etc.

Consensus: apart from the standard horizontal space character, replace all non-printing "space"-like characters with their corresponding Unicode escape sequence.

Trailing closure

`function() { ... }`

Consensus: add a space before the opening brace.

Functional closure

`function({ ... })`

Consensus: trim spaces before opening parenthesis—for example, `compactMap({ $0 })`. As for whether you should trim spaces between the braces and the content they enclose, the prevailing view leans toward "spaces between braces."

Comments

`// comment text`

`/// comment text`

`//: comment text`

`/* comment text */`

`/** comment text */`

`/*: comment text */`

Consensus: add single-space padding between comment delimiters and text.

There are some very odd and varied commenting styles out there in the wild, such as the type that adds a leading * for each line in a /* */-block. One thing that most developers agree on is that the comment delimiter should be separated from its content by a space. (Yes, there are developers out there who do not.)

Trailing whitespace

n/a

Consensus: trim trailing spaces from each line of code.

Last file line

n/a

Consensus: end each file with a single new line, trimming away extra new lines.

Mandating Maximum Line Widths

Some style guides mandate a maximum column count like 80 (LLVM style[1]), 120, and 160 (LinkedIn style[2]). The Swift standard library group adopts LLVM style with 80-character lines, but the Foundation and Dispatch library teams have adopted different house styles that aren't tied to these limits.

When deciding on a recommended layout width, keep standard IDE font sizes and your team's laptop displays in mind. These play key roles with respect to code review readability. Shorter line lengths work better for teams who prefer to read code in terminal windows, incorporate code into pull requests, or print code to paper. It also makes it easier to open multiple files at once and display them side by side on smaller displays, including those ubiquitous laptops. (It's easier to adopt shorter lines in Swift than, say, in Objective-C, where egregious naming may cause a single symbol to exceed 80 precious characters.)

While this decision can be an important component of house style, there's no community consensus to guide you. Probably the best rule is to follow the preferred style of the person who receives the largest paycheck.

Many readers of this book will use Xcode as their primary IDE. Xcode offers a visible page guide that you set in the Preferences > Text Editing pane. This establishes a visual reference for your maximum line widths. If you're using

1. http://llvm.org/docs/CodingStandards.html
2. https://github.com/linkedin/swift-style-guide

a variable-width font, you're on your own. Width references don't work with variable-width fonts.

When working outside line-width limitations, reserve manual wrapping for clarity and emphasizing structure, such as when laying out parameters or distinguishing generic declarations. Xcode does a fine job of presenting long lines.

Xcode's Text Editing preference pane also enables you to do the following:

- Automatically trim trailing whitespace for built-in line hygiene.
- Hide or show line numbers, which is extremely useful when conversing about content with other team members.

Selecting Colon Styles

The Ash Rule (named for developer luminary Mike Ash) adopts left-hugging colons in all uses except operator declarations and ternary forms. His rule applies to dictionary references, protocols, and so forth:

```
let dict = ["a": 1, "b": 2] // Ash Rule
let dict = ["a" : 1, "b" : 2] // Commonly used
```

Here are a few examples of magnetic-left (that is, left-hugging) colons in common use cases:

```
let x: [String: String] = ["key": "value"]
let y = foo(param1: value1, param2: value2)
func bar<T: Hashable>(a: T) -> Void {}
```

Moving to this style felt odd for a while, but I've now grown used to it. This approach is consistent and prioritizes the role of the left item. It is, in my opinion, quite readable. Compare and contrast with my prior style, which I call the Full Monty spacing approach:

```
let x : [String : String] = ["key" : "value"]
let y = foo(param1 : value1, param2 : value2)
func bar<T : Hashable>(a : T) -> Void {}
```

The Ash Rule puts you in step with more than half of Apple source code and out of step with the rest. Colon magnetism tends to vary by group, but it is growing as a trend.

Some people use magnetic left except for protocol conformance and inheritance clauses. Adding spaces before colons elevates prominence. For example, in `extension AnyHashable : Equatable`, the spacing makes colons stand out more. In theory, this enables readers to better identify the colon's use point.

Here's an old example from Swift's source code that showcases this style before this type moved to magnetic-left style. Notice the Full Monty spacing for the generic type parameter and protocol conformance:

```
public struct EnumeratedIterator<
  Base : IteratorProtocol
> : IteratorProtocol, Sequence {
  internal var _base: Base
  internal var _count: Int
```

The current implementation updates both wrapping and colon use. (It also drops conformance to IteratorProtocol and Sequence.) I think it's easier to read now:

```
public struct EnumeratedIterator<Base: IteratorProtocol> {
  internal var _base: Base
  internal var _count: Int
```

Apple Developer Publications uses magnetic-left colons. (See this tweet,[3] for example.) This style is simple and readable:

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
```

Skip left-hugging colons in ternary expressions because they unnaturally prioritize the left value over the right:

```
let result = booleanCondition ? value1: value2 // no
let result = booleanCondition ? value1 : value2 // yes
```

In operator declarations, conjoined colons aren't legal. They confuse the compiler because Swift assumes the colon is part of the operator name:

```
infix operator &&: LogicalConjunctionPrecedence // no
infix operator && : LogicalConjunctionPrecedence // yes
```

Common Colon Styles

When it comes to colon style, there aren't any naturally right or wrong answers. Styles break down into the following common categories:

Label declarations

func f(x: T), f(x y: T)

3. https://twitter.com/_jackhl/status/646723367576276992

Most commonly magnetic left.

Constant and variable typing

`let x: T`

Most commonly magnetic left.

Dictionary declarations

`[key: value]`

Consensus is split between full spacing and magnetic left with magnetic left becoming prevalent.

Empty dictionaries

[:] and not [:]

Universal consensus: no spaces.

Cases

`case a:`, `case a, b:`

Universal consensus: magnetic left.

Attribute

`@attribute(key: value)`—for example, `@available(*, unavailable, renamed: “MyRenamed”)` `@available(swift, deprecated: 4.1, obsoleted: 5.0.0, renamed: "copyMemory(from:byteCount:)")`

Consensus: magnetic left.

Inheritance

`Derived: Parent`

Consensus is split between full spacing and magnetic left, with magnetic left becoming increasingly more common.

Conformance

`<T: A>`, `<T: A & B>`

Consensus is split between full spacing and magnetic left, with magnetic left becoming increasingly more common.

Operator conformance

`public func <= <T>...`

Add a space between the operator symbol and its generic declaration.

Ternary

`A ? B : C`

Always use fully spaced ternary expressions.

Some coders blaspheme and use no spaces at all between labels and argument values. Although this offers a slight advantage (Xcode wraps the two together on long lines), it's an uncommon style and possibly a sin of commission.

```
unsafeBitCast(type, to: Int.self) // yes
String(validatingUTF8: ptr) // yes
unsafeBitCast(type, to:Int.self) // no
String(validatingUTF8:ptr) // no
unsafeBitCast(type,to:Int.self) // no
```

Using Type-Magnetic Colons

As with any style, there are always a few outlier users. Leung-style typing (named for Kenny Leung) uses type-magnetic colons, as in the following example:

```
var soccer :Ball = newBall
```

Here, the colon hugs the type rather than being tied to the symbol or using full spacing. Under this paradigm, a colon introduces the type using a magnetic-right style. While this approach will never be my personal choice, if used consistently, it offers a rational coding alternative to the majority Swift convention.

- Colons are as worthy of style consideration as any other language member.
- Spaces communicate prominence.
- Adopt consistent styling, regardless of language construct.
- Type-magnetic colons are just plain weird. I avoid them.

Placing Attributes

Placing attributes like @objc, @testable, @available, and @discardableResult on their own lines before a member declaration has not only become a widely adopted Swift convention, but there's a flag called LongAttribute that tells Swift to put an attribute on its own line in generated headers:

```
@objc(objectAtIndex:) // yes
internal func objectAt(_ index: Int) -> AnyObject

@available(swift, deprecated: 3.2, message: "Use String directly") // yes
public typealias CharacterView = _CharacterView

@discardableResult // yes
public mutating func insert(
  _ newMember: Element
) -> (inserted: Bool, memberAfterInsert: Element)
```

This approach limits horizontal declaration length. It allows a member to float below its attribute and supports flush-left access modifiers, so internal, public, and the like appear in the left-most column. Official long attributes include @available and @discardableResult. There are a few other less known and less used ones detailed in the Swift source code.[4]

Some developers mix and match styles for short Swift attributes like @objc. You may see @objc(countByEnumeratingWithState:objects:count:) on its own line, while @objc moves down to the primary declaration, particularly for short single-line declarations:

```
@objc internal var count: Int // yes

@objc internal final class _SwiftDeferredNSArray
  : _SwiftNativeNSArrayWithContiguousStorage // okay
```

I prefer to break attributes out universally, regardless of length. This approach promotes consistent, left-most access modifiers:

```
@objc // yes
internal var count: Int

@objc // yes
internal final class _SwiftDeferredNSArray
  : _SwiftNativeNSArrayWithContiguousStorage
```

- Most Swift coders prefer to lead with access modifier keywords.
- It's okay to move attributes to their own lines, even short attributes.

Moving Variables and Constants to the Left

Some development houses treat variables and constants as having slight gravity that pulls them leftward. Developers adopting this styling prefer to mention bound symbols earlier in the line than stand-alone literals or type-supplied values. With this styling, you would prefer to add value + 3 to 3 + value:

```
let result = value + 3 // preferred
let result = 3 + value // not preferred

if value >= 0 // preferred
if 0 < value // not preferred

if value >= Double.pi // preferred
if Double.pi < value // not preferred
```

4. https://github.com/apple/swift/blob/master/include/swift/AST/Attr.def

Moving bound symbols to the left gives them prominence and emphasizes their role when reading lines of code out loud.

Formatting String Literals

In Swift 5 and later, limit your backslashes (\) and pound (#) delimiters to the absolute minimum. Avoid the pound sign entirely when it is not needed. Here are some examples that demonstrate minimal use of backslash and pound.

```
"Hello World" // yes
"Hello, \(name)" // yes, produces string interpolation
#"Hello, \(name)"# // yes, skips string interpolation
                   // to produce: `Hello, \(name)`
"Hello, \\(name)"  // yes, uses a minimal approach of a single backslash

#"Hello World"# // no, extra #
#####"Hello World"##### // no, excessive #
#"Hello, \#(name)"#  // no, string interpolation can be produced without #
#"Hello, \#\(name)"# // no, it's easier to skip string interpolation
                     // without a literal backslash as above for
                     // `Hello, \(name)`
```

When working with backslashes in strings—for example, when writing blocks of pre-escaped JSON—prefer pound sign delimiters to excessive backslashes:

```
// Yes
#"""
    [
        {
            "id": "12345",
            "title": "A title that \"contains\" \\\""
        }
    ]
    """#

// No
"""
    [
        {
            "id": "12345",
            "title": "A title that \\\"contains\\\" \\\\\\""
        }
    ]
    """
```

Formatting Number Literals

Two Swift conventions allow you to enhance *number legibility*, or how clearly numbers in your code can be read, understood, and checked for correctness.

Swift enables you to introduce both underscore separators and leading zeros. This formatting enhances legibility by reducing the cognitive load involved in inspecting numbers. Consistent formatting chunks long numbers into more manageable visual components. Adopting these simple visual flourishes offers a significant legibility win. They don't interfere with the compilation or execution of your code. At the same time, they present numbers in a more human-consumable fashion.

Adding Underscores

Swift ignores single underscores placed within number literals. Adding underscores enables you to break down number literals into more manageable components. If you adopt this style, limit underscores to figures containing four or more consecutive digits for decimal and hexadecimal numbers.

For decimal integers, place an underscore every three digits moving left from the decimal point:

```
let largeInteger = 1_732_500 // easily reviewed
let largeInteger = 1732500   // harder to inspect
```

With hex integers, place underscores every two digits:

```
let largeInteger = 0xff_ff_ff // easily reviewed
let largeInteger = 0xffffff   // 5 f's or 6?
```

Binary and octal integer formatting should reflect their underlying byte pattern without sacrificing readability. Moving beyond four-place underscores eliminates nearly all the advantages introduced by literal formatting:

```
let b1 = 0b01110001_10011111 // bytewise but hard to read
let b2 = 0b0111_0001_1001_1111 // nybblewise, more readable
let o1 = 0o7124_3226 // okay, bytewise
let o2 = 0o71_24_32_26 // okay, nybblewise
```

Prefer two- or four-place chunking even if the results do not correspond to a meaningful byte chunk. It's okay to aggregate four bits at a time for the sake of readability.

It's less common (a little weird but no less valuable) to add underscores to fractional decimal values. Adding underscores supports comprehension and reduces the mental burden associated with inspecting values for correctness, whether the number lies to the left or right of the decimal point.

```
let millionth = 0.000001     // harder to inspect
let millionth = 0.000_001    // more easily reviewed
```

Although Swift allows you to insert underscores haphazardly, avoid doing so. Adhering to the standard "by threes" and "by twos" conventions adds utility. Inserting random underscores just makes your code look silly. It means nothing but uses a form suggesting that it *should* be meaningful. Code isn't art. It's communication. This is misleading at best and actively harmful to your codebase at worst.

```
let test1 = 1_0_0_1 // no
let test2 = 1_0_0__1 // especially no
let test3 = 0118_999_881_99_9119_725_______3 // very very no
```

Avoid nonstandard chunking for structured information like Social Security numbers and phone numbers:

```
let social = 045_68_4425
let phone = 202_456_1111 // U.S. style
```

Strings almost always provide a better solution unless you have a more structured value type that breaks items into shorter subcomponents. Remember, half of U.S. phone numbers overflow 32-bit integers. Even though Apple is moving universally to 64 bits, Swift as a language is not limited to Apple platforms.

- Use three-place underscore chunking for decimal numbers.
- Use two-place underscore chunking for hex numbers.
- Weigh 8-bit byte-at-a-time vs. 4-bit nybble-at-a-time layout for octal and binary numbers.
- Avoid structured-information chunking for number literals that may overflow 32-bit platforms.
- ZIP codes are usually a bad choice for numeric representation as they may lose their leading zeros when printed, as with East Coast U.S. ZIP codes, which start with zero.

Using Zero Padding

Both integer and floating-point literals can be padded with extra zeros to allow column presentation or to adhere to well-defined number formatting. Swift ignores excess leading and trailing zeros when interpreting number literals:

```
let value1 = 005.0300
let value2 = 123.0000
let value3 = 000.0520
let value4 = 010.1621
```

Using zero padding is a "do no harm" flourish, which you're welcome to adopt or ignore per your personal style. I've never bothered using this style, but it's there if you need it. Some might argue that having fewer characters increases readability.

Regularizing Decimal Extent

Avoid code that uses different decimal extents for similar typed arguments. Doing this catches the reader's eye for the wrong reasons. As identically typed color channels, there is no semantic difference here between `alpha` and `red`. Their number literals should not imply otherwise.

```
let color = Color(red: 0.0, green: 0.5, blue: 0.5, alpha: 1.0000) // no
let color = Color(red: 0.0, green: 0.5, blue: 0.5, alpha: 1.0) // yes
```

Some house rules mix and match decimals and floating-point numbers based on their expected role. For example, you might mandate integer literals for coordinates (`CGPoint(x: 0, y: 0)`) and floating-point literals for colors (`UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0)`). Even though both APIs use decimal values, the former are expected to be whole numbers that lie at exact pixel points, and the latter to be levels ranging from 0.0 to 1.0.

Avoid mix-and-match argument styles at a single call site. Pick one literal style or the other for each use case and don't cross the streams. In the following example, there's no semantic distinction between the (x, y) origin and the (width, height) extent. Both refer to screen points (in the measurement meaning, not the location meaning) and both are expected to use round values. Using different styles for identical roles doesn't make sense.

```
let frame = CGRect(x: 0, y: 0, width: 400.0, height: 300.0) // no
```

Mixed styles call the wrong kind of attention to themselves and draw the eye during code review. They make readers wonder whether portions of a number have been omitted intentionally or deleted by accident: "Did 0 mean 0.0 or 0.2?" Rather than explaining yourself by adding unneeded comments, apply consistent literals at each call.

That said, many Swift coders prefer to omit fractions for whole numbers. When used *consistently*, there's no question of whether you meant to use a whole number or not:

```
let red = UIColor(red: 1, green: 0.231, blue: 0.65, alpha: 1) // yes
let blue = UIColor(red: 0, green: 0.231, blue: 0.95, alpha: 1) // yes
```

Balancing Inferred and Explicit Typing

Although the Swift compiler infers typing for literals like 2 and “Hello,” you will never cause an error or reduce safety by explicitly typing declarations. Consider the following examples:

```swift
let freezingPoint: Double = 32 // yes
let freezingPoint: Float = 32.0 // yes

let aDouble = 32 // no, and terrible name for Int instance
let aDouble: Int = 32 // still no because it's a terrible name
```

Casting distances types from variable and constant names. A type should annotate a symbol, not the value that’s being stored in it. Prefer to keep the type name close to its symbol:

```swift
let freezingPoint: CGFloat = 32.0  // yes
let freezingPoint = 32.0 as CGFloat // no
```

Starting in Swift 5, you can coerce literals using a construction form, like `UInt32(42)`, without calling a regular initializer. The `Type(Literal)` form becomes a special case of `Literal as Type`. This change ensures that an instance is constructed using a literal protocol whenever one is available. Even though `Double(32)` looks like a call to an initializer, the compiler now treats it differently from `Double.init(32)`.

The same style point applies to this form. I personally prefer to add explicit typing to variables and constants rather than relying on inference. The latter is not wrong so much as I think the explicit form is better:

```swift
let freezingPoint: CGFloat = 32.0  // yes
let freezingPoint = CGFloat(32.0) // unconventional
```

Mix-and-match inferencing creates inconsistent code:

```swift
// Mix and match: pick one style consistently or the other
public struct MyStruct { // inconsistent
    var a = 1
    var b = 2
    var c = 3
    var d: Int32 = 4
    var e = 5
    var f = “hello”
}
```

Adding types to every declaration produces a self-documenting style that doesn’t rely on casting or inference:

```swift
// Explicitly typed
public struct MyStruct {
    public var count: Int = 32
    public let name: String
}
```

A rigid adherence to explicit typing is not visual noise. It is consistent.

- Avoid type indicators in value symbol names like firstNameString or ageInt. Use them sparingly with reference instances like mainWindow and primaryView. When used, make sure the types are correct. Both mainController and mainViewController are acceptable for a view controller. mainView is not.
- Prefer typing the container (the constant or variable) instead of the value stored into it.
- Relax rules for code that's meant for a short life. Invest wisely in production code. Consistent typing may be annoying, but it's profoundly clear.

Preferring Initializers to Typing

Most types do not offer literal initializers. They rely instead on initializers and factory methods. For example, you can't create a CGPoint using 2-tuples. Swift does not allow the creation of new literal types to enable you to do so:

```
let myPoint: CGPoint = (10.0, 30.0) // not possible
let myPoint: CGPoint = CGPoint(x: 10.0, y: 30.0) // redundant
```

Under these circumstances, your house style *may* choose to omit the redundant type. In doing so, you allow the initializer just across the equal sign to carry the burden of both creating and documenting the item's type. The distance between the symbol and the initializer is minimal, and it's clear to readers what the type will be:

```
let myPoint = CGPoint(x: 10.0, y: 30.0) // okay
let myPoint: CGPoint = .init(x: 10.0, y: 30.0) // horrible
```

This decision, of course, plays no role when the point of declaration is separated from instantiation. In that case, you must type the declaration, whether a similarly named initializer is available or not.

Understanding Literal Inference Rules

The Swift compiler uses the following rules to infer types from literals in the *absence* of additional context:

Int (decimal numbers)

```
    42, 42_000
```

Int (binary numbers)
: `0b101010`

Int (octal numbers)
: `0o52`

Int (hex numbers)
: `0x2A, 0x00_2A`

Double (decimal points)
: `42.0, 0.42`

Double (exponentiation)
: Exponentiation: e (^10) and p (^2)

  `2e4, 4.2e1, 0x15p1, 0xA.8p2`

String (quoted constants)
: `"42 by gum"`

String (Unicode scalars)
: `"\u{203C}"`

String (extended grapheme clusters)
: `"\u{65}\u{301}"`

Bool (true and false)
: `true, false`

In addition, Swift infers types for color literals, images, and files. Extended literals, like the following examples, enable you to create cross-platform code independent of specific type details:

- `#colorLiteral(red: 0.9994240403, green: 0.9855536819, blue: 0, alpha: 1)`
- `#imageLiteral(resourceName: "flourish.png")`
- `#fileLiteral(resourceName: "README.txt")`

When working with these language literals, the inferred type depends on the availability of imported modules. A color literal is inferred as `UIColor` in UIKit but as `NSColor` in Cocoa, even though the declaration code is identical for both platforms.

I recommend you avoid these three literal types entirely. I have had too many bad experiences with them in Xcode, especially with colors. I have also heard too many stories of how they have caused production crashes. No matter how elegant and how cross-platform they are meant to be, development realities weigh against their widespread use. Prefer type initializers like `UIImage(named:)` and `UIColor(red:, green:, blue:, alpha:)` over these type literals. If you must make them cross-platform, use platform-specific typealiases that unify otherwise-identical initialization patterns to a single API:

```
#if canImport(UIKit)
public typealias Color = UIColor
#else
public typealias Color = NSColor
#endif

let yellow = Color(red: 1, green: 1, blue: 0, alpha: 1)
```

Overriding Default Inference

Swift uses inference rules to establish an item's typing. In the absence of additional context, default inferences allow the compiler to determine which type to apply. For example, an integer literal always defaults to Int and an exponentiated number to Double:

```
let x = 42 // infers Int
let y = 2e4 // infers Double
```

Swift literals are typeless. They have infinite precision and no fixed use case. Swift's inference rules say, "Use this kind of literal in its most common or likely role unless something tells you otherwise." Without any further information, the 42 literal will be seen and used as an integer. However, when used with strong typing, a normally Int-producing literal can initialize floating-point values, as in the following examples. That's because Double conforms to the ExpressibleByIntegerLiteral protocol. This protocol allows you to express non-integer values using integer literals:

```
let d: Double = 42 // legal, Double
let d: Double = 0b101010 // legal, Double, avoid
```

Just because you *can* express a double using a binary integer literal doesn't mean you should. While oddball declarations like creating a double instance using a binary pattern are *legal*, avoid them. Create more readable and obvious code by using (standard) integer and decimal-point literals to clarify when you intend to create double values:

```
let d: Double  = 42.0 // yes, obvious at a glance
let d: Double  = 42   // not wrong
```

It's less usual to initialize floating-point constants, especially Double instances, with integer literals, but it's exceedingly *common* to use integer literals when building view frames guaranteed to lie at point boundaries and to represent the extremes of color channels (namely, 0 and 1). Here are examples that use integer literals for readability.

```
let red = UIColor(red: 1, green: 0, blue: 0, alpha: 1) // yes
let frame = CGRect(x: 0, y: 0, width: 400, height: 300) // yes
```

Unless your house style mandates a trailing decimal component, there's no Swift difference between the consistent integers in the preceding code and decimal literals in the following example:

```
let red = UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0) // yes
let frame = CGRect(x: 0.0, y: 0.0, width: 400.0, height: 300.0) // yes
```

Using decimal literals for floating-point arguments amplifies your intent to populate Double, Float, or CGFloat arguments. That said, there's no real harm in using integer literal shorthand for initializers.

Reviewing Literal Protocols

Swift's ExpressibleBy* protocol family allows types to be expressed by the literal types they name. You can pass integer literals to color initializers (like this one) because of the technological support of these protocols:

```
let myColor = UIColor(red: 1, green: 1, blue: 0, alpha: 1) // yes
```

The number 1 can express an integer (let x = 1), a double (let x: Double = 1), and a CGFloat in the preceding myColor initialization example. Here's an overview of common literal protocols currently available in Swift, examples of how you can express elements of each family, and the conforming types you can initialize using the literals:

ExpressibleByArrayLiteral
: [], [1, 2, 3]

  Conforming types include arrays and sets and types conforming to OptionSet.

ExpressibleByBooleanLiteral
: true, false

  Conforming types include Bools.

ExpressibleByDictionaryLiteral
: [:], ["a": 1, "b": 2]

  Conforming types include dictionaries.

ExpressibleByFloatLiteral
: 0.2, 2e2

  Conforming types include doubles and floats.

ExpressibleByIntegerLiteral
: `0, -42`

  Conforming types include doubles, floats, and integers.

ExpressibleByNilLiteral
: `nil`

  Conforming types include optionals. Some believe no other types should ever conform to this protocol.

ExpressibleByStringInterpolation
: `"\(42)"`

  Conforming types include strings.

ExpressibleByStringLiteral
: `"", "forty two"`

  Conforming types include strings and substrings.

ExpressibleByExtendedGraphemeClusterLiteral
: `"\u{65}\u{301}"` and complex emojis

  Conforming types include strings and characters (but not `UnicodeScalar`).

ExpressibleByUnicodeScalarLiteral
: `"\u{203C}"` and complex emojis

  Conforming types include strings and characters (including `UnicodeScalar`).

For example, `Float80`, `Int64`, and `Double` are all expressible by integer literals. String and character instances can be established with string literals. The compiler always chooses the simplest protocol that fits a value. `"\u{203C}"` always uses `ExpressibleByUnicodeScalarLiteral`, and `"\u{65}\u{301}"` always uses `ExpressibleByExtendedGraphemeCluster`, even when initializing a `String`.

Focusing Literals

Uncentralized literals, also called "magic" values, make your code less general and less reusable. Using literals too liberally can indicate fragile code. Conventional wisdom says that you should limit arbitrary value constants wherever possible. You shouldn't have to search through code to tweak literal values scattered here and there.

Instead, create parameterized routines and move "magic" constants to a central location, preferably namespaced to a parent type. Create those values in a slightly longer but more sustainable way. Prefer `maxStudentCount` to `30`.

Static properties provide a Swift-y way to add those values to appropriate types. Centralizing “magic” values to properties ensures that their use is compiler-checked and the same value is consistent across all uses.

```
extension Classroom {
    static let maxStudentCount = 30
}

extension Vector {
    static let zero = Vector(dx: 0, dy: 0)
}

extension UIColor {
    static let duskyRose = UIColor(red:0.73, green:0.41, blue:0.45, alpha:1)
}
```

When working with strings, take this advice further and use localized content from the very start. Even if you never intend to take your code to another language, the structure should be there, prepared for the day when you might. And, when designing that localized content, don’t forget to think of the ways you can add hooks to support accessibility in your work.

Constructing Collections with Literals

Most Swift types require initializers or factory methods to create instances:

```
let instance = Type(arguments...)
```

Most collections support literal initialization. When available, prefer literal initialization to initializers, especially for array and dictionary literals:

```
let array: [String] = [] // yes
let dictionary: [AnyHashable: Any] = [:] // yes
let aSet: Set<String> = [] // yes
let array = [String]() // no
let array = Array<String>() // no
let dictionary = [AnyHashable: Any]() // no
let dictionary = Dictionary<AnyHashable: Any>() // no
let aSet = Set<String>() // no
```

Using literals allows you to maintain better type hygiene in moving the type name to the left of the assignment. This consistent approach enables you to extend your literals to include initial values as needed:

```
let array: [String] = ["a", "b", "c"] // yes
let array: [String] = [
    "a",
    "b",
    "c"] // yes
let array: [Character] = ["a", "b", "c"] // yes
let aSet: Set<String> = ["a", "b", "c"] // yes
let aSet: Set<Character> = ["a", "b", "c"] // yes
```

Limiting First-Item Typing

Avoid or limit typing the first item in an array or dictionary to establish a value. This practice, like general casting, moves type declarations away from the symbol they describe:

```
let array = ["a" as String, "b", "c"] // no
let array = ["a" as Character, "b", "c"] // no
let array = ["a", "b", "c"] as [String] // no
let array = ["a", "b", "c"] as [Character] // no
```

Annotate the symbol and let the compiler handle typing details:

```
let array: [String] = ["a", "b", "c"] // yes
let array: [Character] = ["a", "b", "c"] // yes
```

The same advice applies to static type properties and methods. Prefer typing the symbol and not the collection elements:

```
let array: [NSColor] = [.blue, .red, .green] // yes
let array = [NSColor.blue, .red, .green] // no
```

When looping, prefer to type the iteration binding (that is, the loop variable) rather than the collection:

```
for color: NSColor in [.blue, .red, .green] // yes
for char: Character in ["a", "b", "c"] // yes
for color in [NSColor.blue, .red, .green] // no
for char in ["a" as Character, "b", "c"] // no
for color in ([.blue, .red, .green] as [NSColor])  // no
for char in (["a", "b", "c"] as [Character]) // no
```

- Avoid first-item typing in assignment. Prefer to type the constant or variable, not the value assigned to it.
- Add explicit types to loop variables when literal inferencing does not produce the desired collection type.

Weighing Initializers vs. Literals

Prefer initializers when they are cleaner and more maintainable than their literal alternatives. For example, `repeating` initializers are simple and handy, plus you're far less likely to make a counting mistake:

```
let array: [Character] = ["x", "x", "x", "x", "x", "x"] // no
let array: [Character] = Array(repeating: "x", count: 6) // yes
```

The repeating array initializer is generic. It infers type from its first repeating: argument. The [Character] typing in the preceding example overrides default inference, which otherwise constructs a [String] array:

```
let array = Array(repeating: "x", count: 6) // [String]
let array = Array(repeating: 0.0, count: 6) // [Double]
let array = Array(repeating: 0, count: 6)   // [Int]
```

Adding a type to the repeated item argument produces the correct type but isn't as clean to read as explicit typing:

```
let array = Array(repeating: "x" as Character, count: 6) // no
let array = Array(repeating: Character("x"), count: 6) // no
let array: [Character] = Array(repeating: "x", count: 6) // yes
```

Other initializer variations are available but not recommended. These examples are confusing and can be laid out using cleaner styles:

```
let array = [Character](repeating: "x", count: 6) // no
let array = [Character].init(repeating: "x", count: 6) // no
let array = Array<Character>(repeating: "x", count: 6) // no
let array = Array<Character>.init(repeating: "x", count: 6) // no
```

- Clarity and inspectability trump rules like "prefer literals."
- Initializers provide cleaner solutions than run-on literals and are more adaptable should you need to change counts.

Selecting Collection Type Sugar

Both arrays and dictionaries offer syntactic sugar:

```
Array<Int> // Generic form
[Int] // Sugared
Dictionary<String, Any> // Generic form
[String: Any] // Sugared
```

The empty [] literal can initialize sets, option sets, and arrays, as well as custom types that conform to ExpressibleByArrayLiteral:

```
// Sugared type `[String]`, No sugar available for `Set`
var arrayOfStrings: [String] = ["a", "b", "c"]
var setOfStrings: Set<String> = ["a", "b", "c"]
```

When you do bump up against similar types that don't visually match, you may choose to swap the sugared array type declaration with its generic form to avoid the dissonance between the two. It may not be objectively better, but it is consistent:

```
var arrayOfStrings: Array<String> = ["a", "b", "c"] // Consistent
var setOfStrings:   Set<String>   = ["a", "b", "c"] // Consistent
```

After trying to keep to "all generic types all the time" for about a week, I quickly reverted to mixing and matching my [Arrays] and Sets<>. I don't think it's that big an issue for most coders, but you may or may not want to address this issue in your house style guide.

- Collection sugar is sweet.
- Prioritize. Outlier cases shouldn't be driving your house style.

Using Dictionary-Style Initialization

Types that use dictionary literal initialization outside of the standard library are rare but they do exist. Treat overlaps with sugared dictionaries the same way you treat array literal conflicts. I use sugared declarations regardless of whether they stand out against unsugared ones.

Avoiding Literal Initialization Calls

Avoid using literal initializers directly. These methods are not meant for public use. They exist to support ExpressibleBy* protocols, not in-code calls:

```
var anArray = Array(arrayLiteral: "x")  // do not use
var anArray: Array<String> = ["x"] // yes
```

Initializing Option Sets

Option sets are built on raw values, which are generally hidden from sight by the clever deployment of static presets and literal initializers. Always prefer option set literals to raw value initialization. The results are universally cleaner, more readable, and more maintainable, especially when using non-empty option sets.

```
// no
if let json = try JSONSerialization
    .jsonObject(with: data,
        options: JSONSerialization.ReadingOptions(rawValue: 0))
    as? NSArray { ... }

// yes
if let json = try JSONSerialization
    .jsonObject(with: data, options: [])
    as? NSArray { ... }
```

In the following example, the second call is self-explanatory. You needn't resort to math to either build the raw value (3 in this case) or break the bit flag down to its components and then look up meanings for each bit flag. Prefer the abstraction over the direct use of raw values:

```
// no
if let json = try JSONSerialization
    .jsonObject(with: data,
        options: JSONSerialization.ReadingOptions(rawValue: 3))
    as? NSArray { ... }

// yes
if let json = try JSONSerialization
    .jsonObject(with: data,
        options: [.mutableContainers, .mutableLeaves])
    as? NSArray { ... }
```

- Prefer option set flags to rawValue initializers. They're readable and easy to modify.
- If you work with common groupings, create static presets that combine multiple flags. For example, you might establish a static Energy Star member: public static let energyStar: LaundryOptions = [.lowWater, .lowHeat].

Using Option Sets at Call Sites

As much as I'd like option sets to be a type, they're not. They're a protocol. Their design was established to enable API evolution. Under the current design, developers can break down a single option into multiple refined options and still offer the original components. The following example demonstrates this flexibility. The compound energyStar and gentleStar options occupy an equal footing with basic component bit-shifted flags:

```
public struct LaundryOptions: OptionSet {
    public static let lowWater = LaundryOptions(rawValue: 1 << 0)
    public static let lowHeat = LaundryOptions(rawValue: 1 << 1)
    public static let gentleCycle = LaundryOptions(rawValue: 1 << 2)
    public static let tumbleDry = LaundryOptions(rawValue: 1 << 3)

    public static let energyStar: LaundryOptions = [.lowWater, .lowHeat]
    public static let gentleStar: LaundryOptions = [.energyStar, .gentleCycle]

    public init(rawValue: Int) {
        self.rawValue = rawValue
    }
    public var rawValue: Int
}
```

Although this design *looks* like you must use set-syntax to combine options, you really don't have to. The square bracket syntax is a bit of a cheat. The following example demonstrates how you can combine members with brackets and access members as option sets, all on an equal footing:

```
let options1: LaundryOptions = [.lowWater, .lowHeat]
let options2: LaundryOptions = .energyStar
let options3: LaundryOptions = [.energyStar, .lowHeat]

// prints 3 for each one
for options in [options1, options2, options3] {
    print(options.rawValue)
}
```

Surrounding an option set with brackets produces an identical option set. In Swift, the option set `[.foo]` is the same as the static option set type member `.foo`. When offered the opportunity, should you omit or use the brackets? Here's an example to consider:

```
accessQueue.sync(flags: [.barrier])
accessQueue.sync(flags: .barrier)
```

When the code expects an option set, make the argument look like an option set. Use brackets. Brackets add an affordance[5] that explicitly indicates set functionality. Bracketed options communicate the idea that you can expand the set by introducing more options. Without brackets, this use may not be intuitively obvious to someone less familiar with option sets. When reading your code, they may only see something that looks like an enumeration case and be confused by the use. Brackets are clear and guiding. These are exactly the qualities you want to reinforce in your code.

Optional Sugar

When working with optionals, prefer the question mark shorthand to spelling out `.some`:

```
switch (firstOptional, secondOptional) { // no
case let (.some(first), .some(second)): ...
...
default: break
}

switch (firstOptional, secondOptional) { // yes
case let (first?, second?): ...
...
default: break
}
```

5. https://en.wikipedia.org/wiki/Affordance

Swift's optional sugar simplifies the case let binding and prevents you from having to visually match the let keyword with each bound symbol with the .some case:

```
switch (firstOptional, secondOptional) { // no
case (.some(let first), .some(let second)): ...
...
default: break
}
```

Mitigating Optional Constipation

Although cascaded bindings avoid Swift's "pyramids of doom,"[6] they may lead instead to "constipated blocks of horror." When faced with excess serial bindings, you might interleave statements with newlines and comments:

```
guard
    // Access returnedValue as dictionary
    let dict = returnedValue as? [AnyHashable: Any],

    // Retrieve results array
    let resultsList = dict[resultsKey] as? [Any],

    // Extract first result
    let result = resultsList.first as? [AnyHashable: Any],

    // Extract name and price
    let name = result[trackNameKey] as? String,
    let price = result[priceKey] as? NSNumber

    else { return nil }
```

Just because conditions can coexist within a single statement doesn't mean they should. They should work toward a single goal, with each successive condition related to a single task. For example, you might work your way through a JSON structure or the stages of testing and opening a file stream. If you find yourself combining unrelated tasks into a single complex condition cascade, break them down into separate if or guard statements.

```
guard !skipImageCreation, // no
    let model = self.cgColor.colorSpace?.model
    else { return false }

guard !skipImageCreation else { return false } // yes
guard let model = self.cgColor.colorSpace?.model else { return false }
```

6. https://en.wikipedia.org/wiki/Pyramid_of_doom_(programming)

Converting to Tuples

Swift tuples offer a powerful and often overlooked styling feature in Swift. They enable you to collect related items and apply them in simultaneous statements. For example, you might consider using tuples to declare connected variables using a single line, as in the following code:

```
var (x, y) = (20, 50) // x: Int is 20, y: Int is 50
var (x, y): (Int, Int) = (20, 50) // ditto
```

This compact presentation establishes a parallel between the two symbols and provides a coherent elegance that isn't quite matched by the alternatives:

```
var x: Int = 20 // fine, boring
var y: Int = 20

var x = 20, y = 20 // ditto

var x = 20; var y = 50 // a little ugly
```

Tuples parallelize the way you lay out assignments. You might use this style in initializers or when extracting member values from an instance:

```
public init (x: T, y: U) {
    (self.x, self.y) = (x, y)
}

let (width, height) = (size.width, size.height)
let (r, g, b) = (color.red, color.blue, color.green)
```

Prefer tuple assignment and swapping to old-style `tmp`-assisted value interchange, as in the following code:

```
(first, second) = (second, first) // yes
swap(&first, &second) //  yes

let tmp = second // no
second = first
first = tmp
```

- Support natural relationships in assignments.
- Prefer parallel binding when it reflects a semantic relationship between component symbols.
- Know the tuple. Be the tuple. Adopt the tuple. Love the tuple.

Considering Comma-First Styles

Swift does not use comma-first styling. Coders moving from JavaScript and Haskell have made slight inroads in adopting this convention in collections and declarations:

```
var dictionary = [ "a": "apple"
    , "b": "bat"
    , "c": "cat"
    , "d": "dog"
    "e": "elephant"  // error on this line, missing comma
    , "f": "frog"]

func someLongFunction(arg1: Int
    , arg2: Int
    , arg3: Int
    , onCompletion: () -> Void) {
}
```

Supporters say it's easier to catch syntactic errors because of the left-most line of leading commas. Detractors respond that Swift's compiler will easily catch this class of errors, limiting the utility of an unusually ugly style choice. I'm not a fan. I'd rather align the first item with its brethren. I won't go so far as to say that commas belong at the end of lines ("as the good lord intended") or that left-floating commas look like a debris field, but they're certainly a style that takes some getting used to.

- Trailing commas are conventional. Leading commas look weird in Swift code.
- When working with multiline collections, prefer ubiquitous commas, even for last items.

Wrapping Up

You've now read about how the most minor details can draw attention away from reading and understanding your code. Between inattentional blindness, the source of most stylistic typos, and a code reader's disproportionate attention to errors, you've discovered how important it is to maintain consistent styling. An emphatic adherence to basic house style ensures that code review remains focused on implementation and not on the inconsistent use of spaces or commas.

You should have the essentials you need to create or revise your house styling rules for fine-grained layout. No detail is too trivial so long as the resulting style is reasonable, consistent, and final. A good style guide makes code specifics invisible, allowing meaning and algorithm to pop, supporting code review.

Having explored these specifics of layout, next turn your attention to structure. The following chapter explores how to declare and brace Swift's top-heavy code to enhance both readability and maintenance.

CHAPTER 2

Structure Your Code for Readability

Good layout counters code complexity and makes your code easier to read. Swift's top-heavy style magnifies the importance of braces, wrapping, and line composition. As a Swift declaration grows more complex, the mental effort (or *cognitive load*) required to understand the declaration increases as well.

Fortunately, there are many small ways to make code more readable. Well-presented code emphasizes its underlying meaning and design intent. Something as simple as placing braces can be driven by your code's requirements.

This chapter explores ways you compose code, from braces to line layout to semicolons. You'll learn to support code readability and emphasize code meaning. Read on to discover how well-considered layout enhances readability.

Taking Control of Swift Structure

Swift is crammed with power features like generics and defaulted arguments. These features mean Swift code faces wrapping challenges you don't encounter in many languages, especially C-like ones (even C-like ones such as Objective-C with its ridiculously long symbol lengths). Swift's complex language elements create an unusually dense declaration style.

There's an art to hiding structure and emphasizing intent. Good code layout takes idiomatic language features into account to enhance line-by-line readability. Well-considered choices enable you to focus on code meaning and design intent rather than drawing attention to the minutia of wrapping and bracing.

This section explores the basics of spacing and bracing and how you make good choices for both.

Exploring Spacing Concerns

I generally avoid the spacing vs. tabs debate. It doesn't really matter if your code uses spaces or tabs. You may be a two-denter (like the Swift standard library team). You may be a four-denter (like the Swift documentation group). You may tab-dent or even three-dent, especially if you're a gnuful rebel hipster. Whatever you are, however you indent, and however you set up your IDE to indent on your behalf, it's *okay*. Live and let space.

There are no guiding principles behind the tabs-vs.-spaces wars *so long as you are consistent and your code remains readable*. This becomes exponentially more important when teams work on the same code. Teams that fail to adopt a unified style guide introduce readability "turbulence," where style fluctuates from file to file and line to line. Inconsistent layout increases the work for code maintainers and makes it much harder to get new team members onboard and up to speed.

Don't draw attention to code spacing by adopting cute, clever, or downright ridiculous styles, such as Fibonacci spacing:

```
// no, just, no
func fibSpacing(arg1: Int, arg2: Int, arg3: Int, arg5: Int)
{
 inner1 {  // 1
  inner2 { // 1 + 1
    inner4 { // 1 + 1 + 2
       inner7 { // 1 + 1 + 2 + 3
            inner12 { // 1 + 1 + 2 + 3 + 5
                    inner20 { // 1 + 1 + 2 + 3 + 5 + 8
                    }
            }
       }
    }
  }
 }
}
```

Or flush-left layout:

```
// absolutely no
extension BinaryFloatingPoint {
public var doubleValue: Double {
guard !(self is Double) else { return self as! Double }
guard !(self is Float) else { return Double(self as! Float) }
guard !(self is Float80) else { return Double(self as! Float80) }
guard !(self is CGFloat) else { return Double(self as! CGFloat) }
fatalError("Unsupported floating point type")}
}
```

Or stepwise (aka "escalator") spacing:

```
// not even for a second
extension Collection {
  subscript(first: Index, second: Index, rest: Index...)
    -> [Iterator.Element] {
      var results: [Iterator.Element] = []
        results.reserveCapacity(rest.count + 2)
          results.append(self[first])
            results.append(self[second])
              results.append(contentsOf: rest.lazy.map({ self[$0] }))
                return results
                  }
                    }
```

There's no measurable value in adding artistic layouts to working code. Clever styles detract from readability, maintainability, and flexibility. These styles redirect code emphasis from algorithms to structure, which is exactly the attention you want to avoid. Code beauty flows from simple layouts, harmoniously and consistently applied, not from clever, overwrought, and unmaintainable spacing schemes.

- There's no "true" answer to the spaces-vs.-tabs debate. Pick one. (And only one.)
- Don't be clever or cute in your spacing choices.
- Be consistent.
- Configure your IDE to match your indentation choice. You can set per-project preferences in Xcode when different projects require different code styles. Select your project in the Project navigator, open the File inspector, and tweak the Text Settings attributes.

Understanding Mandatory Bracing

The Swift programming language adopts mandatory bracing for both single- and multiline clauses. This differs from both C and Objective-C, which use optional bracing with single-line clauses. Swift's strict bracing rules ensure that code line insertions are safe and cannot break your intended flow the way they might in C.

The following code demonstrates the fragility of a C-style approach. Scan down to the last condition, where an insertion error occurs. Although the compiler won't complain, there is a serious bug introduced by this new line.

```
#include <stdio.h>
#include <stdlib.h>
```

```
// Handle an even number
void handleEven(int evenNumber) {
    // ... some kind of handler here ...
}

int main(int argc, char **argv)
{
    // Usage
    if (argc < 2) {
        printf("Usage: %s number\n", argv[0]);
        exit(-1);
    }

    // Fetch user-supplied number
    int number = atoi(argv[1]);

    // Process even numbers
    if (number % 2 == 0)
        handleEven(number); // Original line
        printf("The number is even\n"); // New insertion
}
```

In this example, a coder intends to scope the newly inserted printf call into the if statement's success clause. The new line is co-indented with the print statement above it. However, an error occurs between the coder's desires and the program's behavior. The compiled code will always print "The number is even" regardless of whether the user-supplied argument is even or odd.

Unlike Swift, C doesn't force if statements to use mandatory bracing. Since C is not an indentation-scoped language like Python, the newly inserted line isn't incorporated into the if statement's logic. The success clause is limited to the unbraced single line following the modulo check. Because of this, the code invokes handleEven for all number values, not just even ones.

This kind of error was famously demonstrated in Apple's SSL/TLS CVE-2014-1266 bug,[1] also known as the "goto fail" bug or the "goto fail" bug bug. A single line of code (goto fail;) was duplicated. This introduced a serious security error, allowing unauthorized accesses during secure key exchange.

In the following code, the two successive calls to goto fail are not braced. This means the first call is conditional and the second is not. Therefore, the final test is unreachable and will never be called. The end of the function returns the current value of err, which is now guaranteed to be 0 at the second goto fail; line.

```
static OSStatus
SSLVerifySignedServerKeyExchange(
```

1. https://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c?txt

```
    SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
    uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail; // conditional, err !=0
        goto fail; // unconditional, err == 0, serious bug
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Braced-condition errors cannot happen in Swift because Swift requires bracing for *all* condition clauses. When you insert a new condition, Swift braces clarify whether the line of code falls into or outside of the intended scope. Swift was designed to promote safe and reliable code.

Mandatory bracing introduces additional styling tasks into Swift. As the number of braces grows, you must adopt consistent and readable bracing. Each brace should both support the underlying language requirements and emphasize your coding intent.

Swift Bracing

Swift bracing deserves a thoughtful exploration. Swift language features lend themselves to bracing challenges that you may not encounter in many other languages. Swift creates a particularly top-heavy style of code. Because of this, a *lot* of Swift can happen before a method or initializer body is ever reached.

Here's an old example from the Swift standard library. (This declaration has since been replaced.) This code initialized an instance of a bidirectional collection. I'm unable to outmatch the ridiculous top-heaviness of this declaration in the current standard library:

```
public init<C : BidirectionalCollection>(_ base: C)
  where C.Iterator.Element == Element,
  C.SubSequence : BidirectionalCollection,
  C.SubSequence.Iterator.Element == Element,
  C.SubSequence.Index == C.Index,
  C.SubSequence.Indices : BidirectionalCollection,
  C.SubSequence.Indices.Iterator.Element == C.Index,
  C.SubSequence.Indices.Index == C.Index,
  C.SubSequence.Indices.SubSequence == C.SubSequence.Indices,
  C.SubSequence.SubSequence == C.SubSequence,
  C.Indices : BidirectionalCollection,
  C.Indices.Iterator.Element == C.Index,
  C.Indices.Index == C.Index,
  C.Indices.SubSequence == C.Indices {
  self.base = base
}
```

Most of this example's generic constraints relate to indices. Indices are now handled much better by newer versions of the Swift compiler, softening many run-on function headers. Other best practices, like moving constraints to type extensions and creating constrained typealiases, also reduce overgrown declaration size. Even so, Swift declarations tend to run long.

Here is a more typical modern declaration, again top-heavy but nowhere as appalling. Like the former example, it consists of a single line of code packed below a complicated declaration:

```
public mutating func decode<I : IteratorProtocol>(
  _ input: inout I
) -> UnicodeDecodingResult where I.Element == CodeUnit {
  return UTF32._decode(&input)
}
```

And here's a one-liner that is complex only because of its extensive parameters and defaults:

```
public func trap(
  because reason: @autoclosure () -> CustomStringConvertible,
  file: StaticString = #file,
  line: UInt = #line,
  function: StaticString = #function
) -> Never {
  Abort.because(reason(), file: file, line: line, function: function)
}
```

Language elements including generics, defaulted arguments, external and internal labels, and type constraints mean declarations can grow large and complex. Each of the preceding snippets consists of a single line of code. The declaration that took place *before* that single line of code is an example of what makes Swift so hard to style effectively.

Top-heavy declarations play a big role in structuring code. Unless you actively work to make your code readable, you may be hiding important implementation "needles" in the "haystack" of your declaration. Choose the right layout style to differentiate your implementation from its declaring context.

The generally accepted standard for laying out code in both Swift and Objective-C is One True Brace Style, also known as 1TBS. 1TBS offers vertical compactness and wide adoption. While most Swift coders embrace Apple's 1TBS, as in the preceding code example, consistent 1TBS is not the be-all and end-all in Swift.

A minority adopt Allman style instead, aligning opening and closing braces. Allman prizes code readability above compact presentation. Generics, flexible labeling, and other power features build up Swift complexity. Your code may benefit from adopting a more relaxed hybrid approach that incorporates the best features from both 1TBS and Allman.

A few outliers use other styles such as Whitesmiths (also called Wishart), which aligns bracing with the code it contains. You see echos of this style in Swift's switch statement. While once popular in the eighties, very few Swift developers choose it for their production code today.

One True Brace Style

1TBS bracing[2] provides a variation of C-standard K&R bracing.[3] This name derives from pioneers Brian Kernighan and Dennis Ritchie, creators of C and authors of *The C Programming Language*. 1TBS style adopts these features:

- Opening braces appear at the end of statements that establish a clause.
- else statements appear between a closing brace and the next open brace.

1TBS is almost universally adopted by Apple house coders. A typical 1TBS Swift if statement looks like this:

```
if let value = key {
    // ...do something
} else {
    // ...do something else
}
```

This style minimizes vertical extent and pairs closing braces with opening statements. Code is then indented for each level of nested scope. The degree of indentation reflects the enclosing scope. Track the code's left edge to get a good idea of what elements are scoped at what level.

2. https://en.wikipedia.org/wiki/Indent_style#Variant:_1TBS_.28OTBS.29
3. https://en.wikipedia.org/wiki/Indent_style#K.26R_style

1TBS offsets verbosity in a mandatory-bracing language like Swift. It ensures that required braces draw minimal attention to themselves. Using 1TBS's compact style emphasizes code enclosed within a scope without emphasizing the braces.

Allman / BSD

Allman style[4] is named for Eric Allman, the UC Berkeley developer of Sendmail. Allman uses a more verbose approach than 1TBS. Also known as BSD style, Allman's style adds a carriage return before opening braces. The opening and closing braces always match up at the same indentation. The style hallmarks are these:

- Opening braces appear on a new line, horizontally aligned with the start of statements that establish a clause.
- else statements appear on their own line with closing and opening braces placed above and below their own lines, creating a three-line expression of the else condition.

This style establishes a clear visual path from a scope's opening brace to its close. This permits you to vertically scan from one to the other because you can draw a mental line between each matching pair of braces. Although many modern IDEs provide alternative ways to visualize scope, Allman's visual tracking remains beloved by adherents:

```
if let value = key
{
    // ...do something
}
else
{
    // ...do something else
}
```

Allman extends vertically compared to 1TBS. The five-lined 1TBS if statement becomes eight lines in Allman. This limits the code you can view on one screen at any time, which is the major drawback of Allman style. To Allman adopters, clarity and safety always win over succinct form.

Allman exposes scope to differentiate declaration and control structure from content. It prioritizes visual layout over concision. Detractors will point out that Allman-style braces are overemphasized, drawing attention away from scoped code.

4. https://en.wikipedia.org/wiki/Indent_style#Allman_style

Weighing Allman and 1TBS

Swift is not a language *demanding* vertical compactness. If anything, Swift is prone to run-on declarations that extend through many lines and many sections: the name of a method, property, or initializer; generic parameters; external and internal labels; default values; return type (which may involve multistage currying); and generic constraints (also known as a `where` clause). A lot of declaration can happen before the first line of implementation even begins. Because of this, Swift has picked up Allman aficionados as well as 1TBS evangelists.

Simple functions and methods flow well with 1TBS, and this is how the Xcode IDE expects you to work. Although complex generic functions may struggle with 1TBS, smaller and simpler generics can be effectively braced using 1TBS style:

```swift
func myFunction<T: Collection>(_ collection: T) {
    for idx in collection.indices {
        print(collection[idx as! T.Index])
    }
}
```

To override Xcode and adopt Allman style, insert carriage returns before each opening brace. The result is spacious. To those more comfortable using 1TBS, "spacious" may look overly and unnecessarily sparse:

```swift
func myFunction<T: Collection>(_ collection: T)
{
    for idx in collection.indices
    {
        print(collection[idx as! T.Index])
    }
}
```

You can easily travel back and forth between these layouts so long as you're willing to add and remove carriage returns. Xcode automatically handles indentation details so there's no layout effort involved beyond moving braces.

Adopting Mix-and-Match Bracing

Although every house style guide encourages coders to use consistent bracing throughout their code, Swift lends itself to mix-and-match bracing. If you're not a 1TBS or Allman purist, open yourself to the idea of introducing an Allman here and there to enhance your code when Allman is better suited for differentiating between Swift declarations and implementations. Even a primarily 1TBS developer can benefit from a well-placed Allman opening brace when styling complex functions.

A Swift function body or initializer may start after a significant number of elements have been added to the declaration. Between generic parameter declarations, external and internal labels, default parameters, protocol conformances, generic constraints, and more, Swift suffers from extreme declaration verbosity. To give you a sense of this scope, the following elements may appear in function declarations:

```
access-modifiers? function-head function-name
    generic-parameter-clause? func-arguments
    func-signature-result? generic-constraint-clause?
    function-body
```

Swift's top-heavy implementation style means a function declaration can match or exceed the extent of its body. Here's a representative example sourced from the Swift standard library. This function equates two raw representable values:

```
public func == <T : RawRepresentable>(lhs: T, rhs: T) -> Bool
  where T.RawValue : Equatable {
  return lhs.rawValue == rhs.rawValue
}
```

Cognitive load refers to the conceptual complexity of presented information. In this example, the where and return keywords line up even though they belong to two structurally separate sections. The 1TBS layout increases this code's cognitive load, blurring the distinction between body and declaration.

Ideally you want someone reading this code to mentally break it into two sections—the declaration and the body:

```
// declaration
public func == <T : RawRepresentable>(lhs: T, rhs: T) -> Bool
  where T.RawValue : Equatable {

  // body
  return lhs.rawValue == rhs.rawValue
}
```

Unfortunately, 1TBS interferes with that goal in this example. The alignment of where and return heightens the mental effort needed to process and understand the content. This alignment problem is not limited to top-heavy declarations. Here's another standard library example that incorporates a more typical body size:

```
public mutating func replaceSubrange<C: Collection>(
  _ target: Range<Index>, with replacement: C
  ) where C.Element == Element {
  _debugPrecondition(_isValid(target.lowerBound))
  _debugPrecondition(_isValid(target.upperBound))
  var r = _ValidUTF8Buffer()
  for x in self[..<target.lowerBound] { r.append(x) }
  for x in replacement                { r.append(x) }
  for x in self[target.upperBound...] { r.append(x) }
  self = r
}
```

Despite increased code, replaceSubrange struggles with similar layout and readability issues, especially with its indented closing parenthesis. The text creates visual confusion between the end of the declaration and the start of the body. Adding a carriage return at the start of each body scope converts the method container from 1TBS to Allman. The space introduced by the opening Allman separates the body from the declaration and "pops" the implementation. This code presents both of the preceding standard library examples rebraced to use opening Allman:

```
public func == <T : RawRepresentable>(lhs: T, rhs: T) -> Bool
  where T.RawValue : Equatable
{
  return lhs.rawValue == rhs.rawValue
}

public mutating func replaceSubrange<C: Collection>(
  _ target: Range<Index>, with replacement: C
  ) where C.Element == Element
{
  _debugPrecondition(_isValid(target.lowerBound))
  _debugPrecondition(_isValid(target.upperBound))
  var r = _ValidUTF8Buffer()
  for x in self[..<target.lowerBound] { r.append(x) }
  for x in replacement                { r.append(x) }
  for x in self[target.upperBound...] { r.append(x) }
  self = r
}
```

The result is visually cleaner. You don't have to search hard for a transition between declaration and body, and you can identify both elements with a glance.

You can also retain 1TBS and add blank lines to distinguish code sections. The spaces allow the body to pop from the declaration. However, the results feel less coherent and connected:

```
public func == <T : RawRepresentable>(lhs: T, rhs: T) -> Bool
  where T.RawValue : Equatable {

  return lhs.rawValue == rhs.rawValue
}

public mutating func replaceSubrange<C: Collection>(
  _ target: Range<Index>, with replacement: C
  ) where C.Element == Element {

  _debugPrecondition(_isValid(target.lowerBound))
  _debugPrecondition(_isValid(target.upperBound))
  var r = _ValidUTF8Buffer()
  for x in self[..<target.lowerBound] { r.append(x) }
  for x in replacement                { r.append(x) }
  for x in self[target.upperBound...] { r.append(x) }
  self = r

}
```

Both the Allman and the newline-padded-brace layouts highlight the transition from declaration to body. Cognitive load theory calls this distinction *chunking*. These updated examples allow you to *chunk* declarations and bodies into distinct visual groups with mentally separate categories. In doing so, you can better review the relationship between a method's declaration and its implementation.

Swift hybrid bracing offers a mixed style of Allman and 1TBS bracing, as demonstrated in the following example:

```
func f<T: Collection>(_ collection: T)
  where T.Index: Hashable
{ // Allman
  for idx in collection.indices { // 1TBS
    print(collection[idx])
  }
}
```

This style lets you choose the form that best supports each point in code. Allman bracing distinguishes the declaration from the implementation. Standard 1TBS internal bracing maintains language conventions. The result enhances overall readability.

When adopting hybrid bracing, your guidance can be as loose as "use as needed to differentiate an implementation body from its declaration" and as rigorous as "adopt Allman bracing for all declarations and 1TBS for implementations." Consider circumstantial exceptions when deciding between recommendations and laws.

- Apple and most Swift development groups prefer 1TBS bracing.
- 1TBS limits vertical space, especially for complex if statements with else clauses.

- Allman enhances scope tracing. It supports the ideal of clarity above brevity.
- Hybrid bracing separates declarations from implementation while otherwise retaining 1TBS conventions.
- Newline-padded bracing can be visually blocky.

Complex Wrapping

There's nothing intrinsically wrong with long lines and IDE-supplied wrapping. Styled wrapping can emphasize key elements such as parameter lists, `guard` statements, and ternaries, helping their logic pop. Extreme wrapping, if not thoughtful, can devolve into overkill levels. In the end, wrapping typically devolves to house style, of which there are few wrong answers and many right ones. Google's Swift style guide lies at the center of an emerging convergence of community opinion.

Few developers bother hand spacing their arguments, although the results can be more visually pleasing, because their hard work is just one copy-paste from being destroyed by most IDEs. In contrast, wrapping argument lines is fairly common. Adopt argument wrapping in declarations (and similarly at call sites) when

- Your declaration uses many parameters.
- A declaration becomes too hard to read as a single line.
- You want parameter name changes to pop in diff logs.
- Parameters use default argument values.
- Parameters use long type names.
- Parameters use long external label names and internal argument names.

When wrapping short argument lists, you may want to wrap the entire list to a single line. For longer argument details, multiple lines may be more suitable. Here are examples of several approaches from the standard library. The second of these uses a mix and match approach, placing its `file` and `line` arguments onto a single line:

```
// Unified line argument layout
public func index(
  _ i: Index, offsetBy n: Int, limitedBy limit: Index
) -> Index? {

// Mixed argument layout
public func assert(
  _ condition: @autoclosure () -> Bool,
  _ message: @autoclosure () -> String = String(),
  file: StaticString = #file, line: UInt = #line
) {
```

```
// Multi-line argument layout
public func split(
  maxSplits: Int = Int.max,
  omittingEmptySubsequences: Bool = true,
  whereSeparator isSeparator: (Element) throws -> Bool
) rethrows -> [SubSequence] {
```

The Swift community consensus prefers to keep all arguments on a single line or split them onto one line each. The preceding assert implementation from the standard library breaks this rule. Although it can be argued that file and line share enough common semantics that they can be meaningfully grouped together, the code looks slightly messy and unaudited in its current state.

Items that generally lend themselves to wrapping include: lists of generic arguments, comma-delimited method or function arguments, the return type (typically including the preceding arrow token), and any comma-delimited generic constraints that terminate the declaration in a where clause.

Understanding Swift Semicolons

When arriving in Swift from a semicolon-delimited language, adjust your thinking. Rather than terminating statements, approach semicolons as "statement glue." In Swift, semicolons enable you to join related components onto a single line, blending two thoughts into a single group:

```
defer { x = x + 1 }; return x // this is the same as x++
```

Joined-by-semicolons is not a widely adopted style—Swift purists prefer to omit semicolons entirely—but it is a useful one. For example, I use a pushDraw function to temporarily change the characteristics of an active drawing context. Because the scope of each request is so short, I often present them as a single line. Semicolons enable me to treat a two-stage request as a single drawing chunk:

```
// Draw path using the active `color`
// This call is nonperformant
pushDraw { color.set(); path.stroke() }
```

The semicolons combine to produce a one-thought approach.

Rejecting Swift Semicolons

The Google Swift style standard entirely rejects any use of the semicolon, whether to terminate or separate statements. It permits at most one statement per line and rejects conjoined semicolon-glued constructs.

Terminal Semicolons

C-styled code like the following snippet is not conventional in Swift, even after you've switched the signature from `void myFunction()` to `func myFunction()`:

```
func myFunction() {
    statement1; // no
    statement2;
    statement3;
}
```

Some defend the terminating semicolon, saying it makes copying and pasting code to and from Swift easier and more consistent. I disagree. Reusing code between languages is inherently unsafe. Treat Swift with the respect it's due as a distinct language. If your code belongs in C, keep it in C. Swift provides ever-improving interoperation.

- Omit statement-terminating semicolons in Swift.
- Use semicolons as statement glue for short or related concepts. Don't terminate; conjoin.
- When writing Swift, adopt its conventions.

Defer Clauses

Defer clauses provide the most common use case for semicolons. They allow you to consider how values are used immediately and how they may be used at the next stage of a loop:

```
let nextIdx = string.index(after: idx); defer { idx = nextIdx }
```

Semicolons enable you to pair unsafe setup with a corresponding teardown task, such as allocating and then deallocating memory.

```
// Create new buffer (this would normally be on a single Xcode line)
let buffer = UnsafeMutablePointer<UInt8>
    .allocate(capacity: 1024); defer { buffer.deallocate() }
```

This example is distinctly Swift. Semicolons combine statements. They aren't a cut-and-paste insertion from another language.

Ask yourself, "Do these statements have a relationship that links them together?"

- If they read better as a single semicolon-delineated group rather than as one line following the other, join them.
- If they perform two unrelated jobs or two lines of a sequential task (or they're too long for your house style), place them on separate lines without semicolons.

Some developers dislike semicolon glue, preferring to place `defer` statements on separate lines. Those who argue against semicolons point out that a `defer` may be masked by line length, which could lead to misunderstandings when readers don't scan sufficiently horizontally. Semicolons emphasize horizontal completion over vertical layout.

Structuring Single-Line Clauses

The Rule of Joshua (named for Joshua Weinberg, an Apple engineer who first introduced me to this style) requires space padding in colinear (single-line) bracing, as in the following example:

```
if x > y {return z} // no
if x > y { return z } // yes
```

Adopting this rule allows single-line clauses to breathe. They add visual whitespace that helps move the embedded statement or statements away from the control structure that surrounds them. Limit single-line clauses to statements that are quick and uncomplicated.

It's most common to use single-line clauses when chaining functions. In the following example, single-line scopes are passed to chained filter and map functions. The first selects items that are divisible by three. The second transforms the number to a string and prepends it with a number sign:

```
// With single line scoping
let threes = (1 ... 15) // yes
    .filter({ $0 % 3 == 0 })
    .map({ "#\($0)" })
print(threes) // ["#3", "#6", "#9", "#12", "#15"]

// Without single line scoping
let threes = (1 ... 15) // no
    .filter({
        $0 % 3 == 0
    })
    .map({
        "#\($0)"
    })
print(threes) // ["#3", "#6", "#9", "#12", "#15"]

// Alternatively
let threes = (1 ... 15).filter({ // no
        $0 % 3 == 0
    }).map({
        "#\($0)"
    })
print(threes) // ["#3", "#6", "#9", "#12", "#15"]
```

Look for situations where breaking down constituents into three or more separate lines would detract from your code story rather than enhance it. Many Swift purists refuse to consider single-line bracing outside of functional chains, where the results from one short method call feed into the next.

Single-line clauses are also popular in simple do-catch statements. It's reasonably common to append catch statements that print errors and nothing else. More serious error handling deserves more serious formatting.

```
do {
   ...
} catch { print(error) }
```

Single-line else clauses for guard statements provide another use case. When you're just going to continue, return, break, or throw, a single-line scope-leaving statement does the job well. Xcode indents else statements at the same level as the final condition clause, hanging one indentation unit to the right of guard. This layout punches the else clause's subordination to the guard statement.

```
guard
    condition,
    condition,
    ...
    else { return } // or continue, break, throw, etc.
```

Short braced elements provide a single, complete, readable chunk of intent that makes it easy to browse through code.

- Spaces let braces breathe.
- Avoid single-line clauses just to cut back on vertical space. Vertical space is cheap; readability is precious.
- Preferring single-line clauses when breaking items into three lines (or more) detracts from readability. A return, continue, break, or throw doesn't need multiple lines.

- When using single-line clauses, space consistently to enhance reading and emphasize whitespace.
- When building functional chains, either limit each full chain to a single line or place each chained function call on its own line starting with a period.
- Place `catch` on the same line as closing braces. (1TBS form prefers placing `else` clauses on the same line as closing braces as well.)

Hugging Parentheses

Unlike bracing, Swift parentheses want to hug their contents. In Swift, most coders eliminate spaces next to parentheses and square brackets. There's no loss in readability because the parentheses act as the differentiating whitespace between successive elements.

```
print("x:", x) // yes
print( "x:", x ) // uncommon
let paddedCount = String(count).appending(".0") // yes
let paddedCount = String( count ).appending( ".0" ) // uncommon
```

Some developers, many influenced by jQuery, prefer space-padded parentheses. This is known as liberal spacing. Spaces emphasize the separation between internal components and their surrounding construct. It's an uncommon style for Swift, but it is not an unreasonable one.

- Braces want to breathe; parentheses want to hug.
- Most developers adopt no-space rules for parentheses and square brackets.
- Liberal spacing is an uncommon but acceptable alternative.
- Use parentheses generously throughout the language to emphasize intent and precedence, even when parentheses aren't strictly required by the compiler.

Wrapping Argument Lists

In Swift, it's common to break down parenthesized argument declarations to multiple lines when working with complex calls. Here are three examples from the Swift standard library that showcase multiline, parenthesized argument lists:

```
// Declaration arguments on their own line: "Brace Like"
public func zip<Sequence1 : Sequence, Sequence2 : Sequence>(
  _ sequence1: Sequence1, _ sequence2: Sequence2
) -> Zip2Sequence<Sequence1, Sequence2> { ... }
```

```
// Callsite line-by-line arguments, with dangling
// end parenthesis
return Index(
    _base: UnicodeScalarView.Index(
        _position: i._utf16Index - predecessorLengthUTF16
    ),
    in: self
)
```

```
// Line-by-line callsite arguments, with hugging end
// parenthesis
_stdlib_makeAnyHashableUpcastingToHashableBaseType(
      base,
      storingResultInto: &self)
```

Wrapped call sites enhance clarity when comparing version control diffs, especially for complex variadic calls where parameters may be commented in and out during development.

There's no consensus about where the closing parenthesis belongs. Here are some thoughts about the final parenthesis:

- You can adopt brace-like behavior in declarations when a `where` clause or a complex return type follows. You see that in the first example.
- Although dangling parentheses (as in the second example) allow you to match the end of an expression to its start, they also add vertical height.
- Most developers hug the closing parenthesis at call sites, like the third example. Dangled parentheses add little to call site readability. Those who prefer dangling closing parentheses like the symmetry of indentation and the parallels with braces declaring scopes and collection layouts.

The following examples use a longer and less-nested parameter list. Although Xcode wrapping aligns a dangling parenthesis with the `let` keyword rather than the parameter column, I don't prefer this style choice.

```
let requestedFormat = AVAudioFormat( // yes
    commonFormat: .pcmFormatFloat32,
    sampleRate: outputFormat.sampleRate,
    channels: outputFormat.channelCount,
    interleaved: outputFormat.isInterleaved)
```

```
let requestedFormat = AVAudioFormat( // no
    commonFormat: .pcmFormatFloat32,
    sampleRate: outputFormat.sampleRate,
    channels: outputFormat.channelCount,
    interleaved: outputFormat.isInterleaved
)
```

My biased recommendations:

- When breaking out argument lists, whether in declarations or at call sites, prefer individual lines for each item.
- Parentheses want to hug. If you're not going to hug the opening parenthesis, at least allow the closing one to hug the final argument.

Pushing Declaration Argument Parentheses Leftward

Community consensus has grown to adopt left-adjusted declaration parentheses. In this style, the closing parenthesis is manually adjusted left from where Xcode normally leaves it, to align with the start of the declaration (with "public" in the following example), and the closing brace of the scope. This approach accomplishes two things. First, it creates an Allman-like natural balance with the closing brace. Second, it removes the alignment between the closing parenthesis and the first line of code. This clarifies the distinction as to where the code scope begins.

```
public func preconditionFailure( // yes
  because reason: @autoclosure () -> CustomStringConvertible,
  file: StaticString = #file,
  line: UInt = #line,
  function: StaticString = #function
) -> Never {
  ...
}

public func preconditionFailure( // no
  because reason: @autoclosure () -> CustomStringConvertible,
  file: StaticString = #file,
  line: UInt = #line,
  function: StaticString = #function
  ) -> Never {
  ...
}
```

Coaligning Assignments

Many developers align stacked assignments around the equal sign. Avoid this style. It is unnecessarily fussy and brittle:

```
// no
let userKey     = "User Name"
let passwordKey = "User Password"

// yes
let userKey = "User Name"
let passwordKey = "User Password"
```

Introducing a new long symbol forces you to update spacing for every line in your stack. Robust coding allows you to add, delete, and adapt lines without those changes rippling out to affect other lines of code. This kind of fragility is counter to good style practices.

There are times when it's appropriate to use spacing to enhance readability. Focus on code that is guaranteed to cover all possible uses, where you will not need to change and re-space. The following example shows a code makeover[5] for a switch statement. The styled version uses manual spacing to align each of the return cases:

```
// Before styling
switch averagePosts {
case 10000000...25000000: return .top
case 5000000...10000000: return .extreme
case 2500000...5000000: return .major
case 500000...2500000: return .verybusy
case 250000...500000: return .busy
case 100000...250000: return .average
case 50000...100000: return .low
default: return .few
}

// After styling, adding open ranges, numeric underscores,
// and manual spacing
switch averagePosts {
case 10_000_000...: return .top
case  5_000_000...: return .extreme
case  2_500_000...: return .major
case    500_000...: return .verybusy
case    250_000...: return .busy
case    100_000...: return .average
case     50_000...: return .low
default:            return .few
}
```

Using this style is fault tolerant. Each value moves down the case list until it is matched. Some developers prefer exact closed ranges over this style. In doing so, slight errors may allow gaps in those ranges, which are very hard to test for.

Prefer open ranges to type-specific extreme points, such as .max and .min when pattern matching.

5. https://ericasadun.com/2018/03/15/making-number-checks-pretty-with-unbounded-ranges-and-formatting/

Improving Closure Hygiene

Closures start with an optional capture list followed by an optional signature declaration that concludes with the `in` keyword. It's followed by statements that form the closure's body. Closures expressions may include any or all of the following components:

```
{
   [capture-list] (parameters and their types) throws -> result-type in
   statements
}
```

Add declaration elements reluctantly, preferring to include just those items required to make your closure compile and function properly. Any element that *can* be inferred often *should* be inferred. After that, follow your in-house standards regarding additional declaration items. Most Swift developers omit return and parameter types where allowed. A short signature is generally both readable and useful. If you cannot entirely omit a parameter in a closure signature, replace it with _.

Weighing Shorthand Argument Names

Shorthand argument names are a semantic convenience that Swift provides to closures. Instead of providing a complete argument signature, as you would with a function or method, you can refer to parameters positionally—for example, `$0`, `$1`, and so on. When used within a closure, you can omit a fully specified argument list. The arguments are inferred from the closure's expected function type.

Limit shorthand argument names to the simplest closures, such as those used when mapping, filtering, and sorting, especially in one-line function chains. In these cases, focus on the call (for example, < in `{ $0 < $1 }`) rather than on the arguments being passed to that call.

When the argument counts match the function signature, you can pass an operator (<) or function name (`min`, `power2`) instead of a closure:

```
// `min`
[(0, 1), (3, 2), (5, 9)].map({ min($0, $1) }) // [0, 2, 5], okay
[(0, 1), (3, 2), (5, 9)].map(min) // [0, 2, 5], better

// `<`
[(0, 1), (3, 2), (5, 9)].map({ $0 < $1 }) // [true, false, true], okay
[(0, 1), (3, 2), (5, 9)].map(<)  // [true, false, true], better

// `*`
func power2(_ exponent: UInt) -> Int {
  guard exponent > 0 else { return 1 }
  return Array(repeating: 2, count: Int(exponent))
    .reduce(1, *)
}

// `power2`
[1, 3, 5, 7].map(power2) // [2, 8, 32, 128]
```

Mapping a named function allows you to focus on the meaning of the mapped item rather than the details of its implementation. Prefer well-named functions. When you're going to use the same functionality in several places, build a function if one is not already available. Don't create special-purpose, single-use functions to avoid mapped closures unless there is a measurable gain in doing so.

When your closure extends beyond a line or two, establish argument names just as you would in a standard function or method declaration. Names ensure you won't perform mental gymnastics trying to remember what roles `$0` and `$1` correspond to. This issue always pops up for me when using `reduce`. I can never remember whether the partial result is $0 or $1. Adhering to this rule promotes recognition over recall in your code design. Names allow code maintainers to recognize each parameter in context. It's easier to know what to do with a `name` or `address` than a `$0` or a `$1`, requiring a much lower cognitive burden than recall.

- Reserve closure shorthand for short and simple elements.
- Prefer to name arguments for nontrivial implementations.
- Naming arguments emphasizes recognition above recall.
- Use `$0` when your parameter names are worthless.

Indenting Closures

There's not much to say about indenting closures, as most IDEs will handle this matter on your behalf. Ideally, you want to indent a closure's closing brace to the same level as the line that started it unless the closure can be trivially shown as a single line. This rule also applies to array and dictionary literals.

```
// Assignment
let isEven = { (value: Int) -> Bool in // yes
  return value % 2 == 0
}

// Trailing
return sequence.filter { value in // yes
```

```
  value % 2 == 0
}

// Functional
return sequence.filter({ value in // yes
  value % 2 == 0
})

// One liner
return sequence.filter({ $0 % 2 == 0 }) // yes
```

Trimming Closure Declarations

Where Swift permits, limit a declaration to parameter names, or omit the signature entirely (especially in `() -> Void` closures). Adding unneeded elements to your signatures clutters your code and cuts down on readability. Consider the following closure. Nearly every signature element in this example can be safely omitted.

```
{ (index: Index) -> Void in // no
    ...
}

{
  (index: Index) -> Void in // no
    ...
}
```

Refactoring the signature to `index in` provides an excellent compromise between practicality and concision. Named parameters enhance code readability, providing symbolic roles for each argument.

```
{ index in // yes
    ...
}

{
    index in // yes
    ...
}
```

Weighing Colinear in

Many 1TBS adherents place signature declarations on the same line as the closure's opening brace. This works best when closure declarations are short. It's a style I commit to when working with partially applied (also called *curried*) methods, as in the following example:

```
public static func convolve(kernel: [Int16])
    -> (_ image: UIImage, _ divisor: Int32)
    -> UIImage? {
```

```
    return { image, divisor in // same-line declaration
        ...
```

By moving declarations from the 1TBS opening brace to the following line, you mitigate over-long lines. Although less compact, this approach aligns a closure signature with the code that follows. Consider these examples:

```
data.withUnsafeMutableBytes {
    (bytePtr: UnsafeMutablePointer<Int8>) in
    buffer.data = UnsafeMutableRawPointer(mutating: bytePtr)
}

let _ = array.withUnsafeMutableBufferPointer({
    (arrayPtr: inout UnsafeMutableBufferPointer<Int16>) in
    source.copyBytes(to: arrayPtr)
})
```

Placing declarations on their own lines establishes a code “column.” You read progressively, starting with the declaration and moving through each line of implementation. This style is closer to the way you implement functions and methods, where the signature is normally toward the left, either by nature or wrapping, and depending on the degree of generics involved.

Avoid breaking down the `in` line further unless the closure signature is notably long and complex. In such cases, mimic the signature layout you’d use in a normal function, even when you’re placing that layout in the first lines of a closure.

Very few Swift developers move `in` to its own line, separating the closure signature from its implementation. The one-line `in` creates a vertical space between the two:

```
// Not great
let _ = array.withUnsafeMutableBufferPointer({
    (arrayPtr: inout UnsafeMutableBufferPointer<Int16>)
    in
    source.copyBytes(to: arrayPtr)
})
```

This style is rare and unconventional even if it serves a meaningful purpose. When necessary, I prefer to separate closure declarations from their bodies more conventionally, with a blank line.

- Embrace closure argument sugar. Prefer the concision of `image, divisor in` to `(image: UIImage, divisor: CGFloat) -> UIImage`.
- Focus on line length when deciding whether to place the closure signature on the same line as the opening brace or to move it to the following line.

- Prefer colinear in to single-line in. Single-line in is ugly.
- In nested scopes, group the closure declarations with the opening brace.
- Placing closure declarations on their own line can mirror the relationship between declaration and code in functions and methods.

Returning from Single-Line Closures

Some developers prefer to return from single-line closures. Some don't. Swift's syntactic shorthand enables you to evaluate and return single expressions with or without the return keyword:

```
/// Return the result of performing `c` on arguments `b` and `c`
func perform(a: Int, b: Int, c: (Int, Int) -> Int) -> Int {
    return c(a, b)
}

// Called with function argument
perform(a: 1, b: 2, c: +) // yes

// Trailing closure with inferred return
perform(a: 1, b: 2) {  // yes
    Int(pow(Double($0), Double($1)))
}

// Trailing closure with express return
perform(a: 1, b: 2) { // yes
    return Int(pow(Double($0), Double($1)))
}
```

There's no real harm when including return; there's no real point to it either. Some developers prefer inferred returns for functional chains and explicit returns for procedural calls.

- Swift code should be haikus, not epic poetry.
- Trim your code to the minimum necessary to support compilation, readability, and expression of your intent.
- As with all the advice in this book, establish your house style and adhere to it.

Incorporating Autoclosure

Swift autoclosures enable you to automatically wrap an expression into a closure for evaluation when passing the expression as an argument to a function. Its syntactic sugar enables you to omit functional braces.

```
func unless(_ condition: Bool, do action: () -> Void) {
    guard !test else { return }
    action()
}
unless(count < 10) { print("Done") }
```

Prefer to reserve autoclosure for lazy evaluation when short-circuiting expressions (for example, when performing &&):

```
public static func &&(lhs: Bool,
    rhs: @autoclosure () throws -> Bool) rethrows -> Bool
```

Autoclosure parameters are neither required nor recommended for beautification or convenience. Autoclosure should not be motivated by omitting ugly braces when passing an expression to a function parameter. Use autoclosures rarely and with great hesitation. Outside of known, system-supplied functions, autoclosures may be misinterpreted. Their deferred execution may be overlooked when reading code. If you *must* use autoclosure elements, label them carefully.

Apple offers the following autoclosure guidance in *The Swift Programming Language*:

- Use autoclosure carefully because there's no caller-side indication that argument evaluation is deferred (or may not ever be called at all).
- The context and function name should make it clear that evaluation is being deferred.
- Autoclosures are intentionally limited to take empty argument lists.
- Avoid autoclosures in any circumstance that feels like control flow.
- Incorporate autoclosures to provide useful semantics that people would expect (for example, a futures or promises API).
- Don't use autoclosures to optimize out closure braces.

Choosing Trailing Closures

The Rule of Lily (courtesy of Lily Ballard, one of my original technical reviewers and an amazing developer) adds parentheses around functional trailing closures where the closure returns a value, restricting bare braces to procedural calls that update state and/or have side effects. This style creates consistent readability. You always know when a value is expected to return because the parentheses tell you so.

Here are examples of functional elements, each using a brace-parenthesis combination:

```
// Select words that are at least 5 characters long
let words = sentence
    .split(separator: " ")
    .filter({ $0.count > 4 })
    .map({ String($0) })

// Ensure arguments are sequence of positive integers
let nums: [Int] = arguments
    .compactMap({ Int($0, radix: 10) })
    .compactMap({ $0 > 0 ? $0 : nil })
```

And here are some procedural ones, which prefer naked bracing:

```
// Sleep for n seconds then signal
dispatch(after: maxItem + 1) {
    semaphore.signal()
}

// Dispatch after delay
DispatchQueue
    .global(qos: .default)
    .asyncAfter(deadline: delay) {
        // ... execute code here ...
}

// Animate view transformation
UIView.animate(withDuration: 0.3) {
    // ... perform animations ...
}
```

Under this rule, procedural braces parallel naked scopes. Functional braces act more like parameters. This rule does not fully address more complex approaches like promises and signals, which can mix control flow with functional application.

Sometimes a function takes a single trailing closure parameter. The other parameters may be defaulted or there are no others to handle. Follow this rule and don't move parentheses before the trailing closure:

```
sequence.map({ String($0) }) // yes
sequence.map() { String($0) } // no
```

- Procedural != functional.
- Place parentheses around braces in functional contexts.
- Reserve raw braces for procedural closures.

Proactive Compilation Safety

Bracing functional calls ensures that you won't be caught by the compiler when iterating through a mapped result. Swift cannot compile the first of the following two examples. The second compiles without issue:

```
// Does not compile: "trailing closure requires parentheses
// for disambiguation in this context"
for value in items.map { pow($0, 2) } { // no
    print(value)
}

// Compiles
for value in items.map({ pow($0, 2) }) { // yes
    print(value)
}
```

Understanding Return Context

Swift doesn't use separate keywords to differentiate returning from closures and returning from a method or function call. It's surprisingly easy to get lost in the shuffle as you read code, especially when working with nontrivial closures and `guard` statements. Adhering strictly to the Rule of Lily reinforces `return` context.

When you see a paired closing brace and parenthesis, as in the following snippet, it guarantees you're returning from a closure and not from the enclosing method. This support does not apply when you return midway through a closure:

```
    return foo
})
```

Without the Rule of Lily, you cannot be sure whether you're returning from a method or from a closure to the surrounding method:

```
    return foo
}
```

Some developers adopt an in-house style to comment closure-level returns. These comments distinguish `return` statements that move control back to the enclosing function (in the following example, this applies to the `nil` and `outBuffer.uiImage` returns) from those that leave that function's scope (`return result` in this example).

```
let result = kernelBytes.withUnsafeBytes({
    (bytes: UnsafePointer<Int16>) -> UIImage? in

    // Perform convolution
    let error = vImageConvolve_ARGB8888(
        &inBuffer, &outBuffer, nil, 0, 0,
        bytes, kernelSize, kernelSize,
        divisor, &backColor,
        vImage_Flags(kvImageBackgroundColorFill)
    )

    // Check for error
    guard error == kvImageNoError else {
        printImageError(error: error)
        return nil // Closure return
    }

    // Return image
    return outBuffer.uiImage // Closure return
})

// Return result
return result // Function return
```

In Objective-C, it was possible to create a `blockReturn` macro and substitute it for block `return` statements. This customization cannot be duplicated in Swift without a change to the language. Fortunately, Swift compilers now warn on unused results, helping you catch unintentional closure returns.

Using Freestyle Trailing Closures

While some developers adopt a style of "no trailing closures, ever," others use trailing closures whenever the mood strikes. Is it the worst thing in the world to use a simple trailing functional closure? For example, does the following code suggest great style sins?

```
/// Select multiples of five
let fives = (1 ... 200).filter { $0 % 5 == 0 }

/// Establishes an infinite sequence of natural numbers
let naturalNumbers = sequence(first: 1) { $0 + 1 }
```

Of course not. The code is readable enough and it's not *wrong*. The Rule of Lily establishes consistency, not syntactic policy. Adhering to it adds a simple enhancement with measurable benefit and minimal cost. No one will arrest you for not adopting it in your house style.

Balancing Multiple Closure Arguments

Some method and function calls incorporate more than one closure argument. Use consistent parentheses for all of them. Don't parenthesize one and let another trail. It's ugly and unSwifty and unnaturally prioritizes one closure above the other. Prefer consistent closure arguments.

```
// yes
```

```
UIView.animate(withDuration: 2.0,
    animations: { v.removeFromSuperview() },
    completion: { _ in postNotification() })

// no
UIView.animate(withDuration: 2.0,
    animations: { v.removeFromSuperview() }) {
    _ in postNotification()
}
```

Laying Out Partial Application

Currying in Swift (and other programming languages) transforms a function that accepts multiple arguments into a series of partially applied functions using a subset of those arguments. The technique was developed by Moses Schönfinkel and Haskell Curry, which explains its curious name.

Partial application decomposes a function, allowing you to call it in stages, creating a primed, reusable method. In practice, it's similar to using argument defaults when you don't know at compile time what those defaults will be. Currying creates a customizable system, allowing you to set those defaults at runtime.

Here's a typical multistage example:

```
public func projected(function f: @escaping (CGFloat) -> CGFloat)
    -> (_ p0: CGPoint, _ p1: CGPoint)
    -> (_ percent: CGFloat)
    -> CGPoint
{
    return { p0, p1 in
        return { percent in
            let (dx, dy) = (p1.x - p0.x, p1.y - p0.y)
            let (magnitude, theta) = (hypot(dy, dx), atan2(dy, dx))
            let rotation = CGAffineTransform(rotationAngle: theta)
            let translation =
                CGAffineTransform(translationX: p0.x, y: p0.y)

            return CGPoint(x: percent * magnitude,
                           y: f(percent) * magnitude)
                .applying(rotation)
                .applying(translation)
        }
    }
}
```

This method projects a function (such as a sine wave generator or ease-in-ease-out modifier) onto a coordinate system defined by a line segment established by p0 and p1. Applying this function establishes a new custom function that accepts just one argument, a floating-point percentage, and returns a relative point along the projected function.

This example showcases both the curried declaration and its return stages. The declaration lays out partially applied return types in a column. Each partial application starts with the return arrow token. Within the function body, the staged returns correlate with the return declarations. The three `return` keywords correspond to the three return arrow tokens.

As staging grows, the return arrow layout becomes more critical. Enhance readability by placing return arrow tokens at the start of each line. In this example, line-by-line layout helps clarify that the first application accepts a `(CGFloat) -> CGFloat` function, the second accepts a `p0, p1` line segment tuple, and the final function transforms a `CGFloat` percentage into a `CGPoint`.

Structuring progressive closures within the curried implementation is equally important. When currying, keep the `return { closure-signature in` elements together—for example, `return { p0, p1 in`. This approach creates a stepwise approach into the heart of the function, which is otherwise indistinguishable from an uncurried version. Well-laid-out signatures and stepwise closures support currying readability.

At this time, the internal argument names in the curried elements are not picked up by Xcode's QuickHelp system or used by calling functions. They serve no purpose other than to create visual ties between a declaration and the closure signatures. Despite that, I encourage you to include them in your declarations and to use identical names in both places. While this practice is not compiler enforced, adhering to this practice ensures readers can connect the actors in each curried stage with the argument and types declared in the parent function.

- When currying, showcase each return stage.
- Create a stacked column of return type arrows.
- Adopt colinear argument declaration that matches the function signature.

Laying Out Complex Guard Statements

Xcode does a particularly bad job of visually formatting `guard` statements, especially when you include many conditions within one statement. The IDE doesn't align the first clause with the rest of the clauses unless you're using some kind of wacky six-space indentation. Xcode aligns the `else` with the final condition clause and adds an extra level of indentation to the body of the else clause. Here's an example of how a guard statement looks using default layout:

```
guard !arguments.isEmpty,
    nums.count == arguments.count
    else {
        // ... leave scope ...
        // (notice the indentation level)
}

// or

guard
    !arguments.isEmpty,
    nums.count == arguments.count
    else {
        // ... leave scope ...
        // (notice the indentation level)
}
```

It's always a good idea to place individual conditions onto their own lines when building complex statements. This enables you to add in-flow comments, comment out conditions, insert condition-specific breakpoints, or add diagnostic debugging output to each line.

Ensure each guard statement guides a single task. Every condition should relate to each other in a clear flow of logic. When faced with multiple guard conditions, you can fight Xcode to manually align your statements or you can break down the statement into individual tests. Single tests may provide simpler and cleaner results as in the following example:

```
guard !arguments.isEmpty else { ... leave scope ... }
guard nums.count == arguments.count else { ... leave scope ... }
```

If all you're doing in leaving scope involves a simple return, break, or continue, there's no harm in expressing that with a single line:

```
else { return }
```

When adding else clauses to the end of an already long condition, move the else { return } to its own line. Xcode indents and left-aligns it like a condition.

- Adopt single-task guard sequences. Break unrelated statements into separate conditions.
- Prefer clarity in guard statements. Wrap conditions to separate lines as needed to emphasize a sequence of steps.
- Embrace single-line clauses. Space and clarity permitting, it's acceptable to adopt single-line else clauses, regardless of whether they're appended to the end of the

condition clause or placed on a separate line. This approach is not well suited for complex conditions. It's also not generally embraced by 1TBS purists. Reserve single-line clauses for basic use cases like conditional binding with a simple return.

Laying Out Ternaries

Ternary operators enable you to operate on three targets. Like most programming languages, Swift implements a single ternary operator to express a conditional choice. You state a condition and select a value based on its truth value. The Swift version is essentially identical to those you'd find in C and Objective-C, among other languages:

```
let result = condition ? valueIfTrue : valueIfFalse
```

The common-sense rules of ternary layout are these:

Keep it simple. Ternary statements are best for uncomplicated evaluation, especially when the corresponding `if` statement requires additional lines without adding clarity or safety.

Use a single line for simple values. Prefer `return value > 0 ? 1 : -1` to a multiline version of the same.

Align. When using multiple lines, align the ? and : characters. This allows the two outcomes to form a vertical column, offering a clear A/B choice presentation:

```
return style == .left // yes
    ? padString + self
    : self + padString

return style == .left ? // no
    padString + self :
    self + padString
```

Prefer coalescing. With optionals, prefer `nil` coalescing to ternary expressions, especially when your statement logic tests for `nil` and presents a fallback value. The `nil`-coalescing variation is shorter, easier to read, and more conventional to Swift eyes. It involves just two values and one operator:

```
return optValue == nil ? fallback : optValue! // no
return optValue ?? fallback // yes
```

Don't chain or nest ternary statements. Prefer `switch` statements instead.

```
return condition1 ? value1 : (condition2 ? value2: value3) // no
```

```
switch (condition1, condition2) { // yes
case (true, _): return value1
case (_, true): return value2
default: return value3
}
```

Use Booleans. Swift's strict typing requires Boolean values and not integers or optionals as the condition. If you're moving from Objective-C to Swift, keep this in mind when constructing your ternary expressions.

Binary Conditionals

Swift does not support *binary conditionals*. A binary conditional combines a question mark and colon (`A ?: C` vs. `A ? B : C`) and omits the second ("B") argument representing the "positive result" return value.

In Objective-C, `?:` returns the left-hand "A" value for any "looks like a non-zero value." Because Swift ternaries do not infer non-nil/non-null/non-zero conditions, Swift does not include the binary `?:` conditional operator in the language. Swift's `nil`-coalescing `??` optionals operator adopts the `?:` operator's valid-value-or-fallback approach.

Laying Out Long Collections

When expressing extensive collections in code, using multiline layout offers advantages:

- Individual lines are easier to comment—and to comment out if needed.
- Distinct lines provide better diffs when comparing committed revisions.

```
/// Mens colors provide a restricted set of name categories
/// Magenta and Cyan are renamed to more common names
/// and a secondary color walk fills in other colors
public let mensColorNames: [String: Color] = [
    // Primary color walk
    "Red": .red,
    "Orange": .orange,
    "Yellow": .yellow,
    "Green": .green,
    "Blue": .blue,
    "Purple": .purple,
    "Blue-Green": .cyan,
    "Red-Purple": .magenta,

    // Secondary "missing" colors
    "Green-Blue":  Color(red: 0.0, green: 0.5, blue: 0.5,  alpha: 1.0),
    "Dark Purple": Color(red: 0.5, green: 0.0, blue: 0.5,  alpha: 1.0),
    "Khaki":       Color(red: 0.0, green: 0.5, blue: 0.5,  alpha: 1.0),
    "Pink":         Color(red: 1.0, green: 0.0, blue: 0.75, alpha: 1.0),

    // Brown
    "Brown": .brown,

    // Monochrome
    "Black": .black,
    "Gray": .gray,
    "Dark Gray": .darkGray,
]
```

Prefer to use trailing commas in multiline collections such as dictionaries and arrays that cannot practically fit onto a single line. The "Dark Gray" entry includes a comma even though it's the final item in this dictionary. This is legal in Swift. Adopting this practice means you're less likely to introduce errors when commenting out lines or adding new items. It also creates cleaner diffs in version control.

Let your brackets dangle, as if they were scoping braces. Reserving a closing bracket to its own line as in the color names example provides more consistent layout from the first member to the last. It also lets you add new members without having to move the closing bracket. Short arrays, sets, and dictionaries are automatically exempt from this kind of bracketing rule when they fit on a single line, as in the following examples:

```
let items = ["cat", "banana", "pony"]
var primes: Set<Int> = [1, 2, 3, 5, 7, 11]
var counts = ["red": 3, "blue": 6, "green": 1]
```

- Use single lines for short collections.
- Use multiple lines for long collections.
- Use trailing commas for multiline collections.
- Allow brackets to dangle for multiline collections.

Weighing Late Property Declaration

There's a dedicated movement of Swift coders who prefer to add constant and variable declarations at the end of a type implementation rather than the start. Here's an abbreviated example from the Swift standard library. The original is a fairly long example that highlights a subtle declaration layout style. Look at the two internal let declarations at the end of this structure:

```
public struct Zip2Sequence<
  Sequence1 : Sequence, Sequence2 : Sequence>
  : Sequence {
  public typealias Stream1 = Sequence1.Iterator
  public typealias Stream2 = Sequence2.Iterator

  // ...

  /// Creates an instance that makes pairs of elements
  /// from `sequence1` and `sequence2`.
  public // @testable
  init(_sequence1 sequence1: Sequence1,
       _sequence2 sequence2: Sequence2) {
    (_sequence1, _sequence2) = (sequence1, sequence2)
  }

  // ...

  internal let _sequence1: Sequence1
  internal let _sequence2: Sequence2
}
```

This code offers a sense of how this layout affects expectations. Most developers use traditional coding sequence: declare the type, then declare the variables and constants, and then use them within type members and initializers. When I first encountered late declaration, I was a little bewildered. I had to search for where the variables were established. Using consistent late declaration enables you to always locate them, but it may read as *surprising* to other eyes, violating the basic principle of least astonishment.

In this example, the variable names, roles, and typing are obvious. There's little to be gained by forcing their declarations to the start of the type declaration. I come from the school of "early typealias"/"early let and var" declarations. Late declaration slightly declutters the earlier work performed in the type implementation.

- If you use late variable declarations, do so consistently.
- Adopting late variable declarations will not condemn your soul to eternal torment, and that's about all I can really say.

Wrapping Up

You've now seen several examples of structure that you may not have thought about before. In particular, you explored how good layout creates a consistent code-reading experience. You read how thoughtful composition emphasizes structure, allowing you to create well-considered functional organization. Code that defers to and supports human cognition is easier to understand and to maintain.

Thinking about structure enables you to examine code and evaluate it in a new way. “Do this code’s bracing and wrapping styles support or hinder its understanding?” and “How might you restructure this code layout to better support its mission?” These new skills allow you to cast a critical eye on structure, enabling you to evolve guidelines that establish better layout criteria.

After exploring these specifics of layout, it’s now time to look at algorithms. In the next chapter, you’ll move your attention from structure to preferred practices, so your guidelines can emphasize a house approach to common coding tasks.

CHAPTER 3

Establish Preferred Practices

As with any programming language, Swift offers many ways to achieve goals. Some coding practices are better than others. Good guidance emphasizes robust and reliable code. Develop your coding guidelines to select from the merits of competing approaches. Strong recommendations ensure that members of your team always prefer the house approach. Considered rules produce consistent code; that consistency extends throughout large projects. They also ensure lower maintenance costs and reduce the time it takes any team member to get up to speed.

When there's a preferred approach, you eliminate debates. You don't argue about whether you should always, for example, include a reference to `self` or allow stand-alone functions. The decision has been considered, debated, weighed, and adopted. All team members should follow your house guidelines, even when individuals prefer or are used to different rules.

At the same time, your preferred practices should never be fixed in stone. Keep your guidelines open to revision as both the language and your team evolve. In adopting guidelines, there's always the danger of inappropriate rigidity. If a particular rule proves too costly or not of general benefit, change it. Just make sure your review and refinement follow at least as rigorous a process as you did when first adopting that rule.

There's a measurable cost to refining rules after a house style has been adopted. You may need to invest in updating old code as well as training with new stipulations. Always consider the cost of migrating to modified rules. If it will be too expensive and incremental to justify the advantages, you should retain your current system, even one that is sub-ideal. In development, as in life, perfect is the enemy of good.

This chapter surveys common design decision points and guides you through refactoring opportunities. Step beyond simple linting opportunities to look at architectural design points that affect your development.

Testing Assumptions

In Swift, using `precondition` vs. `assert` is not simply a matter of, "Will it run in distribution builds?" The two functions play distinct roles:

- Assertions check your code for internal errors, ensuring the consistency of logic.
- Preconditions check the arguments passed by clients, testing whether they are valid.

As for builds, assertions are not checked in optimized builds, and preconditions are not checked in unchecked builds.

Side Effects

Never introduce side effects in preconditions or assert conditions or, worse, rely on side effects from evaluating the condition. Apart from being horrific style crimes, side-effect code in these statements may be compiled away based on your project's build configuration. Tests that run in debug builds that rely on side effects to pass give a false impression of code stability in release builds. Preconditions and assertions are not the place to incorporate error recovery and mitigation, no matter how well you construct your Boolean test components.

Using Assertions

Assertions test assumptions and help you locate design errors during development. Use them to test conditions that have the potential to be false but *must be true* for your code to execute properly. An explanatory message is optional but highly recommended.

An assertion mandates conditions necessary for proper execution. They simplify debugging by ruling out conditions you *know* to be true, ensuring that failures occur for more exotic reasons. As a rule, the simpler the assertion, the more valuable it tends to be. Break down and separate complex assertions to their own lines to pinpoint specific problems. Here's a trivial example to check for a positive or zero offset:

```
assert(offset >= 0, "offset cannot be negative")
```

Or you might test that you have sufficient memory:

```
assert(MemoryLayout<SwitchCache>.size <= MemoryLayout<Builtin.Word>.size)
```

Use assertions during development to catch logic errors. You and your team are the beneficiaries of assertions, which help you locate logic errors internal to your project.

Assertions are normally limited to debug builds, which are typically unoptimized internal versions. You can override this setting with a compiler flag (-assert-config) when creating, for example, optimized beta distribution.

Assertions can be costly to run. While this won't be a factor in most release builds, where they are suppressed and code is optimized, they may affect performance in debug builds.

- Assertions should be simple.
- Assertions should express the basic conditions that must hold true for execution to continue.
- Add the second message argument. It provides valuable feedback while debugging and it documents your intent. Expressing *why* your assertion should hold true with meaningful words forces you to justify each assertion's existence.

Using Assertion Failures

Use assertion failures (assertionFailure) in code destinations that either are guaranteed to be unreachable or explain why a condition that should never have happened *did* happen. This code demonstrates an unreachable default case:

```
switch value {
case ..<0: print("negative")
case 0...: print("zero or positive")
default:
    assertionFailure("Integer cannot fall outside .min ... .max")
}
```

This switch statement is exhaustive, but the compiler cannot guarantee that. It forces the inclusion of a default case even when that pattern will never be reached. Adding a well-stated assertion failure ensures your assumptions about exhaustive cases were properly founded.

- Use assertion failures for unreachable destinations.
- Use assertion failures or fatalError() in stubs, such as dummy methods that must be overridden by a subclass.

- Prefer guard and assertion failures to forced unwrapping.

Establishing Preconditions

Unlike assertions, a precondition's customer is assumed to be external. The precondition mandates a calling contract. It ensures that satisfactory parameters have been passed to the function or method. Preconditions remain in effect in optimized release builds, so they should be lightweight and inexpensive in execution. While assertions guard the logic within a method, preconditions guard the conditions that must hold true as the method is called.

Preconditions represent the only mechanism Swift offers to enforce input hygiene, so document all parameter restrictions using structured markup. At the same time, understand that many API consumers may not read your documentation, especially for obvious API calls like setting a password parameter to a string.

If your API requires a password to be at least eight characters long, even when fully documented, you may cause some nasty runtime crashes by implementing that check as a precondition rather than throwing an error. If you pass a seven-character password to a library and the application crashes, you're much more likely to toss that library than add password-length checking prior to the call.

Preconditions should express sensible requirements. When there's obvious behavior for your API to perform given a certain input, your precondition should not reject that input. Preconditions should be used to express things like maxSplit >= 0 when calling split, because passing a maxSplit < 0 is obviously nonsense.

```
precondition(index <= endIndex, "Index is out of range")
precondition(!potentialRange.isEmpty, "Can't form an empty closed range")
precondition(dropCount >= 0,
    "Can't drop a negative number of elements from a collection")
```

- Preconditions establish calling contracts.
- Preconditions persist to release builds, so keep them lightweight and inexpensive.
- "Preconditions validate that your input matches your invariants, but that doesn't give you free reign to design really bad invariants."—Lily Ballard

Weighing Guard and If Statements

Scattered throughout my early Swift code were uncountable variations on the following statement:

```
if some condition { continue, break, return, throw, etc }
```

One of my first responsibilities in migrating early Swift to modern releases involved deciding which if statements were better stated as guards. A well-constructed guard statement takes on the metaphoric characteristics of a human traffic flagman. It allows your code to proceed only when it conforms to certain outcomes.

guard statements can include Boolean conditions, conditional binding, and availability tests. They act as non-fatal scope-exiting assertions. When a guard statement's conditions are not successful, you must leave scope in some way, whether through continue, break, return, throw, and so on or by calling a function with a Never return type. In loops, guard statements limit whether each iteration is fully run. In functions, they prevent bad data from causing errors.

Guards are not substitutes for normal if clauses. Avoid guards when else clauses introduce side effects. Guards establish required conditions for code to execute properly, offering early exit—from the method, loops, or other scope—on failure: "this optional must unwrap an actual value," "this test must be true," "this feature must be available," and so forth.

The guard keyword jumps out from code in a way that unambiguously distinguishes these statements from if. Even though your code can be written exclusively using if statements, upgrading to guard from if enhances the clarity of your implementation. It is a style choice that emphasizes the statement's role in testing and routing control flow.

One particularly Swifty refactor involves looking for constant fallback values returned in an if-let statement's else clause. This can usually be refactored to nil coalescing:

```
if let str = gtk_entry_get_text(mainWidget) { // no
    return String(cString: str)
} else {
    return "" // returns a fallback constant
}
```

```
let str = gtk_entry_get_text(mainWidget) ?? "" // yes
return String(cString:  str)
```

When you perform significant computation in the positive branch, refactor from if to a guard statement and return the fallback value in the else clause.

Move the primary computation to the top level of scope. This styling emphasizes positive flow and reduces the prominence of the defaulted return value.

```
if condition { // no
    ...perform significant computation
    return ...value
} else {
    return ...fallback constant
}
```

```
guard !condition  // yes
    else { return ...fallback constant }
...perform significant computation
return ...value
```

When your else clause is fully symmetric and of equal semantic weight, consider using a switch statement instead of if:

```
if someCondition {
    doSomething()
    return foo
} else {
    doSomethingElse()
    return bar
}
```

Re-architecting to switch brings both clauses into equal prominence.

```
switch someCondition {
case true:
    doSomething()
    return foo
case false:
    doSomethingElse()
    return bar
}
```

- Use guard to unwrap and bind a constant or variable.
- Prefer early return with guard to nested if statements that form "pyramids of doom."[1]
- Prefer guard to if when the else clause is simple and non-symmetric.
- Prefer switch to if when the else clause is symmetric and of equal weight.
- Prefer if when the else clause can be omitted and execution continues for both positive and negative outcomes.
- Break down complex comma-delimited conditions into separate guard statements that perform one task each, even if the task contains several conditional steps.

1. https://en.wikipedia.org/wiki/Pyramid_of_doom_(programming)

- Join conditions when they are related or represent a single conditional procedural flow. Break them apart otherwise.
- Adding a blank line after `guard` statement groups separates the `guard` logic from the rest of your body scope.
- When your `guard` statement's `else` clause is more than a line or two long, reconsider your design. You may want to convert your method to a throwing version, allowing the error-handling system to provide better recovery opportunities.
- Alternatively, refactor complex guard statements to an `if` statement. This allows your method to respond at length to a failed condition before leaving scope.
- Question any use of `if-not-else`. You may want to flip the logic to form a more conventional `if-else` approach unless the `if-not` test forms the more important part of your logic story.
- *Quis custodiet ipsos custodes?* In Swift, each testing construct plays an important and distinct role. From assertions and preconditions to `guard` and `if` statements, make sure you're using the right kind of checker for the task.

Choosing Optionals

When a variable's value may be undefined or represent a "missing value" at any point during its lifetime, represent it with an optional type. Use optionals for simple not-yet-defined or currently undefined roles. Optionals canonically capture "possibly undefined" semantics. Optionals might include an omittable field returned by a web service, a missing dictionary value for a non-existent key, or a value that has not yet been created by an asynchronous process.

There are other approaches for biased-value states. For example, you might create a `WebData` type with success (200) and non-success (403, 404, and so forth) cases. The `Optional` type represents a reduction of this concept to its core essentials: a case that stores a value of some kind (`.some`) or no value (`.none`). The "no value" case uses the keyword `nil`.

Errors and optionals play distinct roles in Swift. Optionals represent the absence of a value, providing a placeholder where that value can be stored at some point. Errors are "thrown," to transfer control when an application reaches an inconsistent state. For example, a method might return `nil` to indicate when a matching value could not be found within a collection. A different method might throw when it attempts and fails to read a list of files in the file system. By throwing, rethrowing, and catching errors, your application can mitigate serious issues. Optionals, on the other hand, are just a way to handle the lack of a value.

It is not uncommon to build a function that both returns an optional and can throw an error. For example, imagine you're looking up the path for a filename. You'd return nil if the file was not found but throw an error if the file system was unreadable.

Optional Defaults

Optional variable properties always default to nil. This rule does not hold for let declarations. Explicit initial values lower the "is this a var or let burden," where the reader has to remember the defaulting rule. Prefer explicit assignment. Consistent initial assignment differentiates var from let at a glance, at the cost of a few extra characters.

```
var optionalVarProperty: T? // looks like `let`, defaults to `nil`
let optionalLetProperty: T? // assignment must occur in init

var optionalVarProperty: T? = nil // yes, clearly `var`
```

Optional Booleans

Avoid optional Booleans (unless the wrapped value is type-erased so it's not really an Optional<Bool>). They create a kind of three-state problem in which a value can be true, false, or nil. If you must deal with someone else's Bool? return type, use nil coalescing to test for true:

```
if optBool ?? false {
    // handle true
} else {
    // handle nil or false
}
```

In the highly unlikely case that you should test for all three states, use a switch statement:

```
switch optBool {
case true?: // handle true
case false?: // handle false
default: // handle nil
}
```

Same-Named Shadowing

Conditional binding with guard let, guard var, if let, and if var allows you to create a new constant or variable using the wrapped value of a non-nil optional. When the optional is nil, the binding fails, the new variable is not established, and you either leave scope (with guard) or move to the next statement (with if):

```
guard let x = x else { ...leave scope... }
// x is now unwrapped once the guard
// is satisfied
if let x = x { ...use unwrapped x... }
```

Shadowing[2] (also known as name masking) enables you to reuse variable names. You either establish an explicit scope, as with the if statement, or an implicit one, as with guard. The guard statement was introduced in Swift 2 to avoid pyramid of doom[3] scenarios, causing massive nested indentation from if statements, which compromised the overall readability of the language.

The guard statement mitigates pyramids of doom. When its conditions fail, its else clause must leave the current scope. This rule allows Swift to perform conditional binding while shadowing variable names at the current scoping level. Once a Swift same-name conditional binding guard statement succeeds, the unwrapped optional can be used without further testing or workarounds.

Prefer using same-name shadowing in conditional binding. This approach is both conventional and safe:

```
guard let x = x else { ... } // yes
guard let y = x else { ... } // no
```

Same-name shadowing ensures that the unwrapped value's role follows the logical intent established by the name prior to the if or guard statement. Introducing a new name blurs the line of those semantics. It can prove dangerous. Consistent same-named shadowing prevents you from adding an *unintentional* shadow to symbols in the parent scope.

If y was previously bound to another symbol, using it to unwrap x creates a hazard that isn't caught by the compiler. The preshadow y semantics are compromised by unintentionally rebinding the symbol name. The compiler won't check or warn on your y = x conditional binding, and you will almost always introduce a bug.

This issue is most common when using a typical or obvious name to unwrap an optional to avoid shadowing. Your chance of bugs increases because the name might have been already used within the originating scope:

```
// earlier in scope
let fileName = ...
...
// later in scope
guard let fileName = result as? String else { ... } // bug
```

2. https://en.wikipedia.org/wiki/Variable_shadowing
3. https://en.wikipedia.org/wiki/Pyramid_of_doom_(programming)

Avoid the unwrappedX = x workaround to avoid shadowing. This approach is wordy and unnecessary. You won't need to reaccess the optional wrapped version of x after conditional binding. This widely accepted Swift community shadowing convention ensures your meaning of "newly bound same-name unwrapped version" is preserved when you simply use x:

```
guard let unwrappedX = x { ... } // no
guard let x = x { ... } // yes
```

- Prefer same-name shadows when using conditional binding.
- Avoid unwrappedVersion workarounds.
- Avoid arbitrary unwrapped names. The compiler does not warn you when you shadow existing symbols.

Testing for nil

It is *extremely* rare that you test whether an optional value exists where you don't then *use* the wrapped value. In these cases, where your interest in the wrapped value is guaranteed to be non-existent, prefer a simple nil-check to wildcard optional binding:

```
func tappedImage() { // yes
  guard mainImageView?.image != nil else { return }
  delegate?.tappedImage(on: self)
}

func tappedImage() { // no
  guard let _ = mainImageView?.image else { return }
  delegate?.tappedImage(on: self)
}
```

Nil Coalescing and Side Effects

Never introduce side effects on the right-hand side of a nil coalescing operator. Side effects are a bad strategy, signaling deeper code issues. Prefer to re-architect code to avoid nil checks, or if that's not possible, use conventional Swift conditional binding (if let, guard let) or switch statements. Explicit checks for == nil or != nil, where you don't use the unwrapped variable, normally indicate there is probably a better way to model your task.

```
view.subviews.first ?? { // no
  let newView = UIView()
  view.addSubview(newView)
  return newView
}()
```

```
if view.subviews.isEmpty { // yes
  view.addSubview(UIView())
}

subscript(key: Key) -> Value { // no
    return storage[key] ?? computeAndSave(key)
}

subscript(key: Key) -> Value { // yes
    if let value = storage[key] {
      return value
    }

    let value = ...compute key...
    storage[key] = value
    return value
  }
}
```

In the latter example pair, computeAndSave(key) was written specifically for use with nil coalescing and is not used anywhere else in code. It makes much more sense to take a few more lines and skip the nil coalescing. Computing the key and adding the extra lines to store the value is verbose but clear. The rewrite removes dependencies and localizes key computation to its natural use point. Some developers introduce custom operators such as ??= for "assign on nil" to reduce code length without abusing nil coalescing.

Optional Safety

Avoid forced casts and forced unwrapping. The saying goes, "Every time you use an exclamation point in Swift, a kitten dies." Justify every exclamation mark. Avoiding ! may produce more expansive and wordy code, but the results are almost always less fragile and more reliable, readable, and maintainable. Although no actual kittens have been killed during the writing and production of this book, it's best if your applications do not forcibly unwrap nil, especially in production code.

Don't test for nil and forced unwrap like this:

```
let image = UIImage(named: "Background")
if image == nil { // no
    print("Image not loaded ")
    return
}
let size = image!.frame.size.scaled(by: 2.0)
```

It is better to use conditional binding at the point of declaration to forced unwraps at the point of use. Don't insert a forced unwrap operator mid-call like this:

```
// no
let resourceData = try String(contentsOf: url!, encoding: .utf8)
```

Distinguish whether an optional value reflects an overlooked error condition, in which case rearchitect to introduce a better pattern, or if the value in question is guaranteed to never contain nil, you can safely support an unwrap at the point of declaration.

When a conditional binding succeeds, it establishes a new variable or constant (often shadowed—that is, using the exact name as the optional) to store the unwrapped value. This pattern may seem odd to those new to Swift, but it's safe and, to trained Swift eyes, both conventional and readable:

```
let image = UIImage(named: "Background")
guard let image = image else { // yes
    print("Image not loaded"); return
}
let size = image.size.scaled(by: 2.0)
```

If a value is guaranteed to never contain nil and can safely support an unwrap, use a guard/fatalError pattern at the point of declaration. Using conditional binding for resources you know will never fail ensures that should a Black Swan[4] event happen, runtime logs will provide richer feedback.

```
guard let url = URL(string: ...url location...)
  else { fatalError("URL was not constructible") }
let resourceData = try String(contentsOf: url, encoding: .utf8)
```

Some developers create an unwrap method for optionals that adds a reason field why the unwrap should never fail, performing the same guard-trap actions in a more readable form:

```
let url = URL(string: ...url location...)
  .unwrap(guarantee: .constructableURLSource)
```

Guard statements, assertions, preconditions, and fatal errors allow developers to backtrack and correct assumptions that underlie their design. When an application traps from annotated assertions and preconditions, debug console output explains why straight away. You don't have to hunt down the code, read the line that failed, then establish the context around the line to understand the reason. Embedded rationales are even more important when you didn't write this code yourself.

A misconception exists that forced unwraps are, in and of themselves, bad: that they were only created to accommodate legacy apps, and that you should

4. https://en.wikipedia.org/wiki/Black_swan_theory

never use force-unwrapping in your code. This isn't true. There are many good reasons to use forced unwraps, though if you're often reaching for it, it's a bad sign. Force-unwrapping can be a better choice than throwing in meaningless default values with nil coalescing or applying optional chaining when the presence of nil would indicate a serious failure.

- Limit forced casts (`as!`) and forced unwraps.
- If you must use exclamation points, prefer scenarios where you know either that a crash *cannot* happen or that it's desirable for your app to crash early and loudly, for example, with asset misconfiguration.
- Where possible, reconfigure repeated "guaranteed unwrap" patterns with custom approaches that enhance readability.
- *"I! can! make! Swift! apps! crash!"*

Guaranteed Resources and Guided Landings

Some developers use exclamation points to deliberately crash when an unwrap fails. Proper execution may rely on hardcoded images or known URL constants. To offer a real-world case, consider this outlier where conditional binding could have saved the user experience for an otherwise unhappy consumer. A developer friend of mine crafted production code that mixed forced unwraps with a guaranteed API. His deployed code crashes occurred while querying Apple's smart battery interface on a Hackintosh,[5] a non-Apple platform running macOS/OS X code.

The laptop in question wasn't an actual Apple platform. It used a simulated Apple smart battery manager[6] interface rather than the real one. In this case, the simulated manager didn't publish the full suite of values normally guaranteed by the manager's API.

The developer's API-driven contract assumptions meant that forced unwraps broke his app for that user. The destination system returned dictionaries missing critical keys. Applications normally can't plan for, anticipate, or provide workarounds for code running on unofficial platforms. There are too many unforeseen factors that cannot be incorporated into realistic code that ships.

5. https://www.techopedia.com/definition/19709/hackintosh
6. https://opensource.apple.com/source/PowerManagement/PowerManagement-97.5/AppleSmartBatteryManager/AppleSmartBatteryManager.cpp

Adopting a universal style of conditional binding enables you to "guide the landing" on unexpected failures, including those failures that occur under less-exotic circumstances. Conditional binding lets you introduce a user-facing "bad stuff happened" notification system, like the following example:

```
guard let value = dict[guaranteedKey]
    else {
        dispatchCriticalAlert(
            "Compromised functionality during smart " +
            "battery dictionary lookup.")
        return
}
```

Contrast the preceding safe-landing approach with a forced unwrap approach like this one:

```
// Guaranteed application crash
let value: String = dict[guaranteedKey]!
```

Prefer a style that establishes a positive pathway for recovery and user support. Provide fallbacks and user-facing alerts *even when your assumptions are guaranteed to be correct* as a positive coding style. Universal conditional binding reduces the overhead involved in debugging these unexpected black swan deployments and allows you to respond with, "My software is only guaranteed to work on official platforms." This style adds robustness and assumes that in reality, bad execution can happen for the oddest reasons.

Implicitly Unwrapped Optionals

Hardwired asset setup provides the textbook case for implicitly unwrapped optionals (IUOs). An IUO is an optional that is guaranteed to have a value (.some(value)) after its initial assignment, allowing it to be used in its unwrapped form thereafter:

```
// Implicitly unwrapped property assigned to failable
// initialization call
let splashImage: UIImage! = UIImage(named: "BackgroundSplash")
// splashImage is unwrapped here
```

Unlike other optionals, which may be undefined at some points in their lifetime, an IUO will always have a valid value, or the application will crash the first time you attempt to access the wrapped value. Use IUOs for type initialization to load known assets. It's the preferred approach for Interface Builder (IB), which is where most people first encounter the Type! declaration.

Implicitly unwrapped optionals are only allowed where the implicit unwrapping is part of a declaration. Consider ! as a synonym for ?, where the declaration is flagged to add implicit unwrapping. You'll see this in compiler diagnostics, which refer to ? rather than !, even when working with T! declarations.

Apple explains on its swift.org blog,[7] "You can read String! as 'this value has the type Optional<String> and also carries information saying that it can be implicitly unwrapped if needed.'" The compiler treats all uses of T! as T?, adding the unwrapping flag. Specific details and edge cases can be read at the official blog post.

- Use IUOs if it's clear from your program's structure that an optional will always have a value after that value is first set and can never otherwise be nil. IUOs are an integral part of Interface Builder use.
- When working with non-IUO optionals, prefer guard and fatalError to forced unwraps. It's always better to provide an explanation than to randomly crash. Always provide an explanation for each fatalError by supplying it with a message string.
- Outside IUO hardwired asset setup, excess exclamation points are a reliable indicator of unsafe code.
- Starting in Swift 4.2,[8] IUOs are simply a special case of Optional rather than a separate type.

Casting Conditionally

Type casting enables you to reinterpret an instance's type at runtime. Swift offers three such operators: as, as?, and as!. These operators cast instances to other types by downcasting (converting to a more specialized type) or upcasting (converting to a less-specialized, typically type-erased type). Casting is most commonly encountered when working with type-erased collections, such as web-sourced JSON data. Any erased type (Any, AnyHashable, AnyObject), protocol, or root class (NSObject) offers a likely opportunity to work with Swift type casting.

The type-checking is operator tests whether an instance can be resolved to another type:

```
// If you try this code in a playground, ignore
// the warnings, as these values can be determined
// at compile time and are not usual for production
// code

import Foundation
```

7. https://swift.org/blog/iuo/
8. https://swift.org/blog/iuo/

```
protocol Number {}
extension Int8: Number {}
let date = Date(); let number = Int8(1)

date is Date // true
date is NSObject // true

Int8(1) is Number // true, conformance
Int8(1) is Int // false
```

The as? operator applies conditional casts. It checks an instance and returns an optional that contains the typecast value (if the cast succeeds) or nil. Prefer as? and conditional casting to is-based conversion. Using as? is safe, succinct, and conventional (as is the undecorated as for any type pairs that offer free conversion, like String and NSString):

```
var dict: [AnyHashable: Any] = [:]
dict["Key"] = 23

// no
if let anyValue = dict["Key"], // `Any`
    anyValue is Int // Check if cast will succeed
{
    let intValue = anyValue as! Int // forced downcast
    // ...
}

// yes
guard let intValue = dict["Key"] as? Int
    else { return }
```

This approach applies two stages: first the conditional cast to the right and then a conditional binding on the left. A little Swift sugar/magic happens here. Although you might expect the lookup (an optional result) and the conditional cast (another optional result) to produce a doubly wrapped optional—basically T??—Swift provides implicit flattening for conditional casts. When either the lookup or the conditional cast fails, the guard's else statement is called. If not, the conditional binding treats the results on the right side as a simple optional.

Implicit flattening applies to all conditional casts, not just those used in binding. Here are some examples that demonstrate this non-intuitive approach:

```
let wrappedValue: AnyHashable? = "value" // typed AnyHashable?
let unwrappedValue: AnyHashable = "value" // typed AnyHashable

// Conditional casting produces optional
let optString1 = unwrappedValue as? String // typed String?

// Implicit flattening for conditional casts produces
// String? and not String??
let optString2 = wrappedValue as? String // typed String?
```

Swift also implicitly lifts optionals. However, outside of conditional casts, Swift *does not* implicitly flatten them:

```
// Implicit lifts
let veryOptyString: String???????????? = optString2
    // Typed String????????????

// No implicit flatten
let lessOptyString: String??? = veryOptyString // Error!
```

The results of conditional casts are no different than any other optional. You can use them with `map`, `flatMap`, optional chaining, and so forth. Although working with optionals requires more overhead, prefer conditional over forced casts except when you unambiguously know in advance that a cast *must* and *will* succeed:

```
// optional chaining is simpler than `map`
(object as? Date).map({ $0.timeIntervalSinceReferenceDate })

// yes
(object as? Date)?.timeIntervalSinceReferenceDate

// no
(object as! Date).timeIntervalSinceReferenceDate
```

- Prefer conditional casting to forced casts, especially when working with type-erased data like web-sourced JSON results.
- Prefer optional chaining and mapping to conditionally applied functionality (for example, `if x is y`).
- Combine conditional casting with conditional binding in chained extraction operations. For example, you might cast a JSON object to an array and then select the nth dictionary. Using conditional casting and binding (`if let array = json as? Array<Any>`) enables you to cascade each step of the extraction process. This is simple, effective, and Swifty.

Chaining Calls

In Swift, you chain methods and properties by appending period-delimited selectors. Each function in the chain returns an intermediate value, which is then passed as the receiver of the next method. This allows calls to be joined into a single statement without requiring variables that store intermediate results:

```
myInstance.surname.count
```

This is ideally a parsimonious and more readable expression of a set of operations you want to consider as a single unit. A danger, of course, lies in overchaining.

If you're producing enormous lines of code that are difficult to debug and hard to read, or if they cannot be easily commented or differentiated on updates, you're doing something wrong. Ask yourself, "Would I ever need to put a breakpoint in this statement or step through it?" If the answer is yes, you are overchaining.

This same restriction applies to overly nested calls. Where would you put a breakpoint in the following call? Break down complex calls into individual tasks rather than glomming them together for the sake of concision. Extra storage and symbols at a local scope can be optimized away by the compiler.

```
public var isRelieved: Bool = true { // no
    willSet {
        buttonSetRelief(
            unsafeBitCast(fetchMainWidget(),
                to: Button.self),
            newValue ? .normal : .none)
    }
}
```

Swift method calls may return optionals, and you must take this into account when forming chains. Optional chaining offers a different approach for accessing and transforming values. An optional chain fails gracefully when any constituent link resolves to nil. Unlike conditional binding, conditional chaining returns optionals, so adapt your code to work with optional return values or provide sensible, non-arbitrary fallback values to coalesce your optionals back to unwrapped forms:

```
// returns size as an optional
let size = image?.size.scaled(by: 2.0) // yes

// uses nil coalescing
let size = image?.size.scaled(by: 2.0) ?? .zero // yes
```

When working with long chains, optional or not, prefer to add carriage returns before dots and let the dots align in an indented column below the receiver:

```
characterArray // yes
    .map({ self.applyTransform($0) })
    .filter({ self.contains($0) })
```

A column of dot-demarcated functional calls is easy to read and comment. It supports easy scanning of functional flow. It allows you to comment out any part of the chain and to better view those differences in revision history comparisons.

- Don't overchain.
- Structure your chain as a column. It's easier to read and follow the flow.
- Enable chaining by designing failable methods that return optionals instead of throwing errors.
- Consider the Law of Demeter[9] when constructing chains. Reserve chaining to items that exhibit a natural relationship to each other. Just because you *can* chain methods doesn't mean you *should*.

Moving from Thrown Errors to Optionals

Swift offers two variations of try that bypass normal error handling. try! halts execution if an error is thrown. try? converts throwing results into optionals, returning a wrapped value on success or nil otherwise. Using try? discards errors.

Forcing crashes (with try!) and discarding errors by coercing results to optionals (with try?) are not good practices for anyone without sufficient experience to deploy them meaningfully. At a minimum, prefer to catch and log errors instead of ignoring them entirely:

```
public func attempt<T>(closure: () throws -> T) -> T? {
    do {
        return try closure()
    } catch { print(error); return nil }
}
```

- Avoid forced trys (try!) outside of playgrounds and simple shell scripts.
- Limit try? to rare occasions when you never need to know *why* a valid result was not returned.
- Prefer to handle an error over printing it. Prefer to print an error over discarding it.

9. https://en.wikipedia.org/wiki/Law_of_Demeter

Unwrapping Variables

When working with conditional binding of *variables* (not constants), prefer guard var, case var, and if var to their let variations. This practice prevents you from introducing an unnecessary second shadow to support mutability:

```
if let x = x { // no
    var x = x
    // use x as a variable
}
```

```
if var x = x { // yes
    // use x as a variable
}
```

Distinguishing flatMap from compactMap

The most recent versions of Swift renamed flatMap to compactMap for some uses of the method.

Flat mapping operates on a sequence of sequences. It returns an array that concatenates the results of the applied transformation across all the subsequences:

```
// [1, 2, 3, 4, 5, 6]
[[1, 2, 3], [4, 5, 6]].flatMap({ $0 })
```

Compact mapping operates over an array of optionals, returning an array excluding all nil values.

```
// [1, 2, 3, 4, 5, 6]
["a", "1", "2", "3", "b", "4", "5", "c", "6"]
    .compactMap({ Int.init($0) })
```

When used with optionals, you can map and flatMap, but you cannot compactMap. The former two allow you to conditionally operate on the value of an optional, producing nil should a value not exist. Flattening allows you to perform a failable operation on a value while flattening the results to return a normal optional. Mapping returns a nested optional.

```
return stringValue.map({ Int.init($0) }) // returns `Int??`
```

```
return stringValue.flatMap({ Int.init($0) }) // returns `Int?`
```

Mapping in Condition Clauses

I rarely use optional mapping in condition clauses. I think it's confusing to most readers. Prefer clarity even when that causes you to bind an extra variable, as in the following example. Here, the key unwraps and shadows a variable defined as part of the problem statement. This is entirely safe:

```
if let value = key.flatMap({ dict[$0] }) ... // no
if let key = key, value = dict[key] ... // yes
```

If you rephrase the call away from the let x = x pattern, as in the following snippet, you may shadow an existing variable. This would cause a conflict in the if statement's success clause. It's an unlikely scenario, but it's one you should consider. The non-shadowing, optional-mapping alternative presents no such issues:

```
// `key` could inadvertently shadow here
if let key = foo(), value = dict[key] ...

// This approach removes the use of `key` but is so rarely
// used that it's a confusing construct to read
if let value = foo().flatMap({ dict[$0] }) ...
```

Despite the minimal danger, I think the two-stage approach wins in clarity:

- Does the optional map/flatMap approach more effectively avoid potential errors? *It can avoid unplanned shadowing, but it's hard to understand.*

- Does the optional map/flatMap approach better communicate the programming story? *Not for me, but it may for people more comfortable with thinking "mapping" and "optional" together.*

- Will the optional map/flatMap approach be more maintainable? *I really don't think so.*

I regularly use optional mapping outside of condition clauses, particularly when applying closures and functions. It's just this one overly complex use case (as a shorthand for conditional binding) that provides a specific coding stumbling block for some developers.

Iterating Collections of Optionals

When looping over a collection whose components are optionals, prefer case let and optional sugar (the question mark) to conditional binding. These approaches save you a scope and offer top-level access to each non-nil item. The following example shows the simplicity you gain by adopting optional sugar:

```
for case let item? in array {  // yes
    // iterates through each non-nil item in array
    ...
}
```

```
for item in array { // no
    if let item = item {
        // iterates through each non-nil item in array
        ...
    }
}
```

When working with optional elements, apply compactMap to filter a collection. This approach converts the following array of optionals into a new collection of non-optional elements. The new non-optional array consumes some extra amount of memory but can be accessed more simply.

```
// Select all non-nil optionals from [T?]
let array: [T?] = ... // [T?]
let compactArray = array.compactMap({ $0 }) // [T]
```

compactMap works well in chained processing, allowing you to exclude nils:

```
// Convert optional time intervals to array of date strings
let dates = timeIntervalArray.lazy
    .compactMap({ $0 })
    .map({ Date(timeIntervalSinceReferenceDate: $0) })
    .map({ formatter.string(from: $0) })
```

Working with Optional Collections

Optional collections (as opposed to a collection of optionals) are common when working with system calls and web service results. When a function call returns an optional collection (not a collection of optionals but an optional return type), you may consider any of several iteration approaches. These include nil coalescing (the simplest solution), functional chaining, and nested conditional binding.

nil Coalescing

nil coalescing avoids conditional binding overhead, and the result is always well behaved. It provides a simple and reliable way to coerce an optional collection to a non-optional alternative, providing a safe fallback for nil results:

```
// nil coalescing removes the need to unwrap
for item in (items ?? []) {
    print(item)
}
```

Note that I parenthesized the coalescing in this example. I did this for two reasons. First, it enhances readability, and second, nil coalescing is always of low precedence. Make a habit of using parentheses to ensure coalescing is separate from other expression that might interfere with your intent.

When possible, prefer to use empty collections over optional ones. When you do not control the design, nil coalescing gets you from optional to empty with a single safe step.

Functional Chaining with forEach

You may also apply functional chaining and forEach with optional collections. The results are reasonably clean and succinct. They use one body scope and no guard checks:

```
FileManager
    .default
    .componentsToDisplay(
        forPath: "/Users/ericasadun/Library")?
    .forEach { print($0) }
```

Avoid if-conditional binding. Conditional binding may use unnecessary nesting and is overly verbose and, as you've already seen, cleaner alternatives exist.

```
// Conditional binding with nested scope
if let items = items { // no
    for item in items { // nested
        print(item)
    }
}
```

The same advice applies to guard binding. Prefer a guard variant *only* when you know you'll reuse items later within the same scope and it's important to keep an unwrapped version on hand.

```
// Conditional binding, then for-in
guard let items = items  // wordier but reusable
    else { ...leave scope... }

for item in items {
    print(item)
}
```

Adding Lazy Evaluation

Lazy evaluation defers computation until a value is required. Also known as call by need, lazy evaluation enables Swift to delay execution until a computation is needed. If the value is never used, you save whatever time would have been used to evaluate that computation.

You use lazy evaluation with collections and sequences to avoid creating intermediate arrays when performing eager evaluation (via map, flatMap, filter, compactMap, and the like). This saves on storage overhead. Lazy evaluation also limits computation time since you don't calculate the next member until it's required:

```
// The `lazy` keyword prevents eager evaluation
internal init(_ seq1: Sequence1, _ seq2: Sequence2) {
    let sequence = seq1.lazy.compactMap ({ // yes
        item1 in seq2.lazy.map ({ // yes
            item2 in (item1, item2)
        })
    })
    _iterator = sequence.makeIterator()
}
```

If lazy doesn't appear anywhere in your code, then map, filter, and so on eagerly[10] produce arrays.

```
// If you want to test out this example, the **
// operator is defined later in this chapter

// Eager array creation
stride(from: 0, through: 10, by: 2).map({ $0 ** 2 })

// Lazy stride, returns a lazy map sequence
stride(from: 0, through: 10, by: 2).lazy.map({ $0 ** 2 })
```

Typically, any method that takes a non-escaping closure will only be lazy when you add the lazy keyword before that call. A lazy operation that accepts a closure will hold onto that closure, requiring it to be escaping in its declaration. An eager operation will use a non-escaping closure, ensuring that the closure can be released as soon as the function is applied.

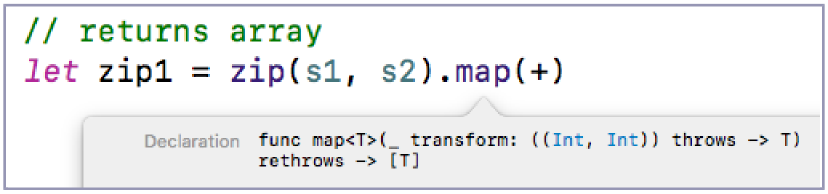
```
// Sequence/Collection version of map
public func map<T>(
    _ transform: (Self.Element) throws -> T) rethrows
    -> [T]

// LazySequenceProtocol/LazyCollectionProtocol version
public func map<U>(
    _ transform: @escaping (Self.Elements.Element) -> U)
    -> LazyMapCollection<Self.Elements, U>
```

Methods that do not use closures may *choose* to be lazy at any time it makes sense to do so. For example, zip is inherently lazy, but mapping on zip results is not, unless you've specifically added the lazy keyword:

```
// returns array
zip(s1, s2).map(+)

// returns LazyMapSequence
zip(s1, s2).lazy.map(+)
```

10. https://en.wikipedia.org/wiki/Eager_evaluation

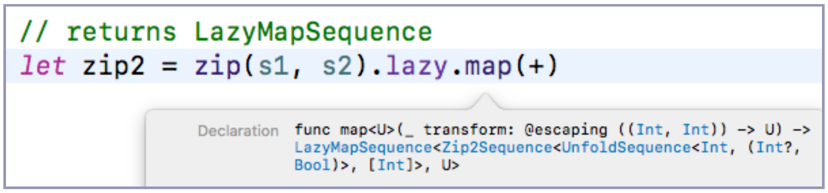
Xcode's built-in QuickHelp context pop-ups enable you to check whether the map (or filter and so on) function being applied is eager or not. In the first of these examples shown here, it is eager. There is no keyword, and closures default to non-escaping behavior.



```
// returns array
let zip1 = zip(s1, s2).map(+)
```

```
Declaration  func map<T>(_ transform: ((Int, Int)) throws -> T)
             rethrows -> [T]
```

In the second example shown, the lazy keyword forces the creation of a lazy sequence. A different @escaping implementation of map ensures lazy evaluation:



```
// returns LazyMapSequence
let zip2 = zip(s1, s2).lazy.map(+)
```

```
Declaration  func map<U>(_ transform: @escaping ((Int, Int)) -> U) ->
             LazyMapSequence<Zip2Sequence<UnfoldSequence<Int, (Int?,
             Bool)>, [Int]>, U>
```

- Prefer lazy evaluation that avoids the unnecessary creation of temporary arrays.
- Don't store lazy values in properties unless you know exactly what you're doing.
- Throw lazy into any functional chain that combines a potentially infinite sequence with mapping. This avoids eager creation of infinite arrays. Subsequence operations, including prefix and suffix, should also make you reach for lazy. (Speaking of which, always filter before you map to have fewer elements to map over.)
- Swift lazily initializes global constants and variables. You do not need to add a lazy attribute.
- Swift eagerly initializes local constants and variables and never evaluates them lazily.
- Always question any use of defer at the end of a scope. In that location, it's no different than just including the code itself and often indicates a planned coding direction that never fully formed.

Selecting Sequences and Strides for Numeric Progressions

Swift offers several built-in approaches to numeric progressions. These include ranges, stride functions, and sequence functions. Select your solution based on the progression you'll produce. The latest versions of Swift provide excellent tools for establishing sequences.

In particular, Swift has eliminated many of the issues that previously detracted from floating-point strides. Strides no longer introduce cumulative errors during their execution the way traditional C for loops did. When choosing between stride and sequence, base your selection on utility, not accuracy concerns.

Implementing Linear Progressions

Use ranges to establish bounded, increasing-integer progressions. They provide the simplest, most obvious, and inspectable solutions:

```
// Iterates through 1, 2, ..., 9
for value in 1 ..< 10 { ... } // yes
for value in stride(from: 1, to: 10, by: 1) { ... } // no
```

When crafting with non-unit progressions—for example, adding twos or threes—or when working with floating-point numbers or moving in a negative direction, prefer stride:

```
// start, start + increment, start + 2 * increment, ...
for value in stride(from: start, through: end, by: increment) // yes
{
    ...
}
```

Swift provides two stride styles, both of which allow both positive and negative progressions. The to variant approaches but does not reach the end value you specify in the second argument. The through variant terminates when the end value is reached or passed. Avoid overly complex solutions that use sequence or apply maps over ranges to build simple incremental progressions:

```
for value in // no
    sequence(start: start, next: { $0 + increment })
    .prefix(while: { $0 <= end })
{
    ...
}

for value in (0 ..< iterationCount) // no
    .map({ start + increment * ($0 - 1) })
{
    ...
}
```

- Prefer ranges for positive unit progressions.
- Prefer strides for positive or negative linear progressions.

Implementing Nonlinear Progressions

Use strides or ranges combined with map to create nonlinear progressions, such as f(x) = n^x. Establish a monotonically increasing or decreasing index and apply map to apply a function f(x) to each x:

```
infix operator **

/// Return base ^ exponent
func **(base: Int, exponent: Int) -> Int
{
    return repeatElement(base, count: exponent)
        .reduce(1, *)
}

// f(x) = x ^ 2, x in 0, 1, 2, 3, 4, 5
// prints 0, 1, 4, 9, 16, 25
for value in (0 ... 5).lazy.map({ $0 ** 2 }) // yes
{
    print(value)
}

// f(x) = x ^ 2, x in 5, 4, 3, 2, 1
// prints 25, 16, 9, 4, 1, 0
for value in stride(from: 5, through: 0, by: -1) // yes
    .lazy.map({ $0 ** 2 })
{
    print(value)
}

// f(x) = x ^ 2, x in 0, 2, 4, 6, 8, 10
// prints 0, 4, 16, 36, 64, 100
for value in stride(from: 0, through: 10, by: 2) // yes
    .lazy.map({ $0 ** 2 })
{
    print(value)
}
```

For simple f(x) progressions, avoid implementations that incorporate tests within the loop body, the next: closure, or require a prefix call to terminate. These are unnecessarily complicated substitutes for working simple ranges and strides, which are all that's necessary for this class of progressions.

```
for value in  // no
    sequence(first: 0, next: { $0 + 2 })
    .lazy.prefix(while: { $0 <= 10 })
    .lazy.map({ $0 ** 2 })
{
    print(value)
}

for value in // no
```

```
    sequence(first: 0, next: { $0 + 2 })
    .lazy.map({ $0 ** 2 })
{
    if value > 100 { break }
    print(value)
}
```

- Nonlinear progressions are easily implemented with mapping.
- Prefer using ranges and strides as seed values. They offer natural end points without having to implement additional tests.

Implementing Inductive Progressions

Use sequence to create inductive progressions, where each f(x) application depends on the result of f(x-1). Terminate your sequences by applying some variation of .prefix(while:). This creates a natural stopping point for the sequence, without affecting the loop body.

```
// f(x) = 2 * f(x - 1): 1, 2, 4, 8, 16, 32, 64, 128
for value in // yes
    sequence(first: 1, next: { $0 * 2 })
    .lazy.prefix(while: { $0 <= 128 })
{
    print(value)
}
```

Avoid using stateful sequences with overly complex next closures. Expansive next clauses create top-heavy, unreadable Swift, and they provide little measurable advantage over the sequence(first:, next:) approach:

```
// f(x) = 2 * f(x - 1): 1, 2, 4, 8, 16, 32, 64, 128
for value in // no
    sequence(state: 1, next: {
        (state: inout Int) -> Int? in
        defer { state = state * 2 }
        return state > 128 ? nil : state
    })
{
    print(value)
}
```

Reserve stateful sequences for progressions where there's a true transformation (often a type change) between the generating state and the iterator elements it produces.

- Use sequence functions to produce inductive sequence elements, whose results depend on the previously computed sequence value.
- Terminate inductive sequences within the loop body or by applying a prefix function rather than adding logic to the next clause.

Looping

Here are a few ways to consider how you approach iteration in Swift. From choosing variables to selecting the right iterator, you have various practices to evaluate and consider.

Choosing Iteration Variables

Where reasonable, avoid meaningless iteration variables like i, j, and tmp, or use them judiciously. Reserve x and y iterators for coordinate systems, where there's a fundamental tie between their names and their use. Although I'm fond of using item as a semantically neutral iterator, adopt Swift's generic type names where possible. Names like element, key, value, and index are drawn from Swift's associated type vocabulary. They provide an excellent basis for iteration in the absence of more specific semantics.

If you're a big image-processing geek and find yourself using a lot of y-x-j-i loops, remember that convolution libraries including Apple's Accelerate are almost always more performant and less error prone than writing your own four-times-or-more nested loop routines. Accelerate handles RGBA, RGB, and ARGB ordering variations, offering an excellent resource for image processing.

Considering while Loops

Both while and repeat-while loops are uncommon in Swift practice. The most common use case is the relatively unsafe while true that drives an infinite or breakable loop.

- When your code uses an abundance of while and repeat-while control flows (especially code whose origins lie in C-like languages), evaluate your source with an eye toward refactoring unless you're performing some truly state-machine-like task.
- I hold a mild personal animus against repeat-while's ergonomics (in addition to any other concerns) because its condition cannot be read until you've worked your way down the entire loop body. This makes it difficult to establish your code's intent at a glance.

- View control transfer labels with extreme caution. Relying on break label and continue label suggests you're filtering data. Swiftier approaches, including sequences and filters, may offer better solutions.

Comparing forEach with for-in

Swift's forEach and for-in loops are superficially similar, so how do you choose which to use? Here are some ways to decide between these statements:

forEach works well with function references. Prefer forEach to for-in when you can apply a function reference to each member of a sequence:

```
array.forEach(performTask) // yes
for item in array { performTask(onValue: item) } // no
```

forEach works well with function chains. Prefer forEach to for-in for short simple calls at the end of a long functional chain. If the iteration source is an excessively large functional group, you don't want to have to search for the iteration block:

```
value.something.something.something // yes
    .something.something.forEach({ print($0) })

for item in value.something.something.something // no
    .something.something {
    print($0)
}
```

You cannot pass a functional closure (one that returns a typed value) to forEach. It expects a body that returns Void. In the rare case when you need to apply a functional closure with side effects, transform the loop into a for-in style:

```
array.forEach { _ = sideEffecting($0) } // no
for item in array { sideEffecting(item) } // yes
```

Treat for-in as your default. In the absence of other compelling reasons leading you to choose one style or the other, prefer for-in to forEach.

for-in is more natural than forEach. return, break, and continue behave as you expect. For example, using return from for-in returns from the enclosing functional scope. Using return from forEach returns from a closure *to* the functional scope.

```
// This for-in example returns "Inner"

func forInExample() -> String {
    for _ in [1, 2, 3] {
        return "Inner"
    }
    return "Outer"
}
let y = forInExample() // Inner
// This forEach example returns "Outer"
func forEachExample() -> String {
    [1, 2, 3].forEach { _ in
        return "Inner"
    }
    return "Outer"
}
let x = forEachExample() // "Outer"
```

Xcode warns on unused results, as in the second example's `return "Inner"` line. This offers you a helpful contextual clue that your intended return may not act the way you expect.

A `for-in` loop always names arguments. Names document the role of iterated values, making loops read more naturally.

```
for (key, value) in dictionary { ... } // yes
for value in 3 ... 10 { ... } // yes
```

`forEach` arguments are less intuitive. You either add named arguments to `forEach` in the closure declaration, which is longer, or use anonymous positional arguments, which are less self-documenting. Prefer argument names to anonymous arguments in nontrivial bodies.

```
for item in array { print(item) }
array.forEach { item in print(item) }
array.forEach { print($0) }
```

You can save vertical space by leaving the closure signature on the same line as the closure opening or on the same line as a simple function call:

```
items.forEach { item in // yes
    print(item)
}
items.forEach { // yes
    item in print(item)
}
```

- Prefer meaningful iteration variables.
- Retaining control flow from other languages isn't always the best Swift solution.
- Choose your iterator thoughtfully.

Indexing and Enumerating Collections

Swift offers helpful ways to optimize the way you interact with collections. Here are some tips and pointers about collections that you may want to adopt into your preferred practices.

Preferring Values to Indices

Working with slices (for contiguous values) or indices (for non-contiguous ones) limits memory overheads. When iterating, prefer values over indices. This transfers complexity from a loop body to its declaration to create cleaner implementations:

```
// Iterate over a slice
for value in array[2...5] { f(value) } // yes

// Iterate using indices
for index in 2...5 { f(array[index]) } // no
```

Retaining Slices

Slices are great for iterating and creating subcollections, but a tiny ArraySlice can be backed by a much larger array. Don't hold onto slices any longer than you need to. When you require access for significant periods of time, convert slices to arrays (Array(slice)) and allow the original source array to deallocate.

Dictionary Keys and Values

Iterate dictionaries using keys and values rather than enumerating them. Dictionaries naturally produce key/value pairs that better lend themselves to immediate use and avoid accessing tuples by their offsets:

```
for (key, value) in dict { // yes
    print(key, value)
}

for item in dict { // no
    print(item.0, item.1)
}

for item in dict { // no
  print(item.key, item.value)
}

for (_, item) in dict.enumerated() { // extra no
    print(item.0, item.1)
}
```

Working with Indices

Calling enumerated() on a collection does not produce indices. It generates a monotonically increasing integer, which happens to be the same as the index for Array but not for anything else, especially slices:

```
let array = [0, 1, 2, 3, 4, 5, 6, 7]
let slice = array[3...6]
for (idx, value) in slice.enumerated() {
    print(idx, value) // (0, 3), (1, 4), (2, 5), (3, 6)
}
```

When you need to iterate through both the index and the member, iterate the indices and look up the member or zip the collection with its indices (although there is a slight performance hit with this):

```
for index in collection.indices { // yes
    let member = collection[index]
    ...
}
for (index, item) in zip(collection.indices, collection) {  // yes
    ...
}
```

Some indices are essentially meaningless on their own, such as those used in sets and dictionaries. They're meant for Swift consumption, not for direct developer use. You can confirm this by printing one out:

```
let dict = ["a": 1]
print(dict.indices.first!)
```

This prints the following:

```
Index(_value: Swift.DictionaryIndexRepresentation<Swift.String, Swift.Int>
    ._native(Swift._NativeDictionaryIndex<Swift.String, Swift.Int>(
    offset: 0)))
```

This information is neither helpful nor meant for direct developer consumption. Prefer iterating members (for example, for item in items) to working directly with indices, unless the index is in and of itself of interest to your algorithms.

Indexing arrays is inherently unsafe, so check bounds and/or implement a safety-checked collection method.

Switch Statements

Refactor comparison-heavy and nested if statements to switch statements where possible. Good targets of opportunity include series of conditions based on the same variables. For example, the following code checks for variations of condition1 and condition2 truth values. This flow can easily be refactored to switch. Each condition is basically matching against the same pattern:

```
if condition1 && condition2 {  // no
    ...
} else if condition1 && !condition2 {
    ...
} else if !condition1 && condition2 {
    ...
} else { // or !condition1 && !condition2
         // or !(condition1 || condition2)
    ...
}
```

A tuple-based switch statement creates an easy-to-inspect list. It transforms the complex snaky flow of the preceding example into a simple truth table:

```
switch (condition1, condition2) { // yes
case (true, true):
    ...
case (true, false):
    ...
case (false, true):
    ...
case (false, false):
    ...
}
```

Real-world code is rarely this clean. The following example, sourced from a friend, shows a more typical example of complex if statements. There's even a bug, one that's easily bypassed by refactoring to switch.

```
fileprivate func chargeState(for battery: BatteryService)
    --> ChargeState
{
    if battery.state == .isACPowered {
        if battery.isCalculating {
            return .calculating(isDischarging: false)
        } else if battery.isCharging {
            return .charging
        } else if battery.isCharged {
            return .acPower
        } // error here, non-exhaustive tests
    } else {
        if battery.isCalculating {
            return .calculating(isDischarging: true)
        } else {
            return .batteryPower
        }
    }
}
```

The following refactored switch version collects state and isCalculating into a single tuple, reserving isCharging to a where clause. (That test is only of interest in the one case.) The resulting switch statement is easy to scan and validate:

```
fileprivate func chargeState(for battery: BatteryService)
    -> ChargeState
{
    switch (battery.state, battery.isCalculating) {
    case (.isACPowered, true):
        return .calculating(isDischarging: false)
    case (.isACPowered, _) where battery.isCharging:
        return .charging
    case (.isACPowered, _):
        return .acPower // fix error
    case (_, true):
        return .calculating(isDischarging: true)
    default:
        return .batteryPower
    }
}
```

This refactor uses the wildcard symbol (_) instead of false in the second and third cases. Prefer wildcards to express "match anything" or "don't care." Using false suggests the case has a specific interest in whether the service is actively calculating a value or not. It has no such interest. With the wildcard, you reinforce to your code readers that a specific value does not matter.

Adopting tuple labels further enhances readability in the following update:

```
fileprivate func chargeState(for battery: BatteryService)
    -> ChargeState
{
    switch (battery.state, calculating: battery.isCalculating) {
    case (.isACPowered, calculating: true): // *
        return .calculating(isDischarging: false)
    case (.isACPowered, _) where battery.isCharging:
        return .charging
    case (.isACPowered, _):
        return .acPower
    case (_, calculating: true): // *
        return .calculating(isDischarging: true)
    default:
        return .batteryPower
    }
}
```

Labels annotate roles in the matching pattern. You see this in the asterisked lines in the preceding example. You can tell at a glance that the true and false values refer to whether the service is calculating, without having to refer to the initial values passed to the switch statement.

Labels are optional for wildcard patterns. The third pattern remains (.isACPowered, _) after the change. Otherwise, you must include the label for any explicit value.

- switch statement targets of opportunity include any pattern matching that can be listed in a truth or state table. Refactor if statements that traverse multi-element truth tables.
- When your switch statement is interested in only one case, consider whether it's better represented as an if statement.
- Combine the states that matter most into a unifying tuple and perform pattern matching against that tuple.
- Use where clauses for logic that applies to only one or two cases at most.
- Prefer wildcard (_) pattern matching when representing "don't care" conditions, even when coding logic has narrowed that argument to one remaining value. Including an actual value detracts from the "don't care" meaning.
- Use tuple labels to add meaning to raw values, especially in complex switch statements. Labels establish a *role* for pattern tuple members.
- switch is also excellent for testing membership in ranges (case 1 ... 10) and collections (case .red, .green, .yellow).

Satisfying Exhaustive Conditions

When working with enumeration cases, prefer mentioning every case than catching final cases with default. You want the compiler to error when adding a new case, even if that means listing many cases in your switch statement. Reserve default cases for these situations:

- Switches where only a few enumeration cases are handled and at least two or more cases are satisfied with default
- Complex case statements that cannot be guaranteed as exhaustive by the compiler

When you know your case statements are exhaustive but the compiler cannot guarantee this, it's appropriate to place a well-stated fatal error in the default case. This helps ensure your logic really was consistent during debug testing.

```
switch value {
case .min ..< 0: print("negative")
case 0 ... .max: print("positive")
default:
    fatalError("Integer cannot fall outside .min ... .max")
}
```

Weighing Conditional Choices

Avoid clever conversions from if statements to switch statements, no matter how ugly the if statement. If the conditions aren't naturally related in some way and can't be represented in a simple table, a switch statement is not the right solution. Here are examples that showcase a badly chosen move from if to switch:

```
if condition1 { // yes
    ...
} else if condition 2 {
    ...
} else {
    ...
}

switch true { // no
case _ where condition1: ...
case _ where condition2: ...
default: ...
}

switch (condition1, condition2, condition3) { // no
case (true, _, _): ...
case (_, true, _): ...
case (_, _, true): ...
default: ...
}
```

Limiting Unneeded Breaks

For safety, switch cases naturally terminate before the following case unless a fallthrough keyword is used. Avoid break except in the absence of other statements within a case or default clause. (In that situation, you must always use a break statement.) This enables you to intentionally exclude conditions that would otherwise catch in other cases. The following snippets demonstrate how you should (and should not) be using the break statement:

```
switch count { // yes
case 1...3:
    print("small quantity")
case 4...6:
    break // deliberate skip
default:
    print("large quantity")
}

switch count { // no
case 1...3:
    print("small quantity")
    break
case 4...6:
    print("medium quantity")
    break
default:
    print("large quantity")
    break
}
```

Swift does not emit warnings when compiling break-terminated cases. Like semicolons, automatic break-ing can be a holdover from other languages. Train yourself away from this. Prefer to err on the side of too-few break statements. If you skip a necessary break, the compiler will catch the issue for you. A case or default label in a switch statement must have at least one executable statement, which may be as simple as a break statement.

Adopting Open Ranges

Open ranges make it easier to tweak your cases by providing only a single point of change. In the preceding section, you could change the limit for a small quantity from 3 to 4 only by editing two different cases. Incorporating open ranges produces simpler code, with just one modification point, as in this rewrite, which drops each lower bound:

```
switch count {
case ...3:
  print("small quantity")
case ...6:
  print("medium quantity")
default:
  print("large quantity")
}
```

Alternatively, you can open the upper bounds by reversing the case order, as you see here. You must reverse the order because this approach creates increasingly larger ranges for matching:

```
switch count {
case 7...:
  print("large quantity")
case 4...:
  print("medium quantity")
default:
  print("small quantity")
}
```

Some developers prefer exclusive ranges as they are slightly easier to mentally process at first glance. Having explicit start and end points may be more familiar and comfortable for reading. In such case, the advantages of open ranges (a single modification point for future maintenance) must be weighed against readability.

Implementing Void Breaks

Swift allows you to pass () to an ignored case statement instead of break. Avoid this. It's ugly and nonconventional:

```
switch value {
case .first: () // no
case .second: break // yes
}
```

Prefer break to () for any case you intentionally ignore. Using break communicates to code maintainers that you've chosen to either intentionally ignore the case or must bail before other less-constrained cases can potentially match the same pattern.

Nesting Tests

Avoid nesting switch statements with further if or switch conditions. Nesting creates complex, unmaintainable code that's hard to read, debug, and trace.

Here's how to avoid nesting:

- Add where clauses to cases to resolve branching.
- Add condition tuples as shown in the previous example to represent multiple states.
- Refactor to another control flow.

The following examples show you how to work around nested tests in switch statements:

```
switch condition1 { // no
case true:
    // Avoid nested tests
    if array.isEmpty {
        ...
    }
case true: ...
default: ...
}
```

```
// Integrate `where` clause
switch condition1 { // yes
case true where array.isEmpty: ...
case true: ...
default: ...
}

// Test multiple state
switch (condition1, array.isEmpty) { // yes
case (true, true): ...
case (true, _): ...
default: ...
}

// Refactor to another control flow
if condition && array.isEmpty { // maybe
    ...
} else if condition {
   ...
} else {
   ...
}
```

- Prefer exhaustive cases to defaults. A complete set of explicit cases is always more meaningful than all-but-one cases.
- If you're working too hard to transform conditional code to a switch statement, an if statement may supply a more natural and better match to your needs.
- Avoid adding unneeded break statements. Swift naturally ends evaluation at the end of a case. Don't pad your code to look like other languages that don't offer this safety feature.
- Prefer break to (), even if the behavior is functionally equivalent.
- Nested tests suggest that you're fighting the language. Either incorporate multiple test values or refactor your code to another control flow.

Declaring Number Constants and Variables

Here are a few quick rules that apply to declaring constants and variables of number types:

- Prefer Int to more specialized variants unless you need to work with a specific size of integer. Int supports code consistency and interoperability and is fully supported on 32-bit as well as 64-bit platforms.
- Reserve UInt use for when you specifically need an unsigned integer using the platform's native word size. In all other cases, Apple recommends you

prefer Int, even when values are *known to be non-negative*. Apple writes, “A consistent use of Int for integer values aids code interoperability, avoids the need to convert between different number types, and matches integer type inference.”

- Prefer Double to Float unless you have some compelling reason to do otherwise. (Prefer CGFloat in Core Graphics routines, where they are required.) Wherever both types are appropriate, prefer Double. Double offers at least fifteen decimal digits of precision. Float may use as few as six.

Implementing Getters and Setters

Swift allows you to move beyond simple properties to implement custom getters and setters as well as property observers that allow you to react to value changes. Here are some thoughts about preferred practices to use when implementing these features.

Side Effects

Getter and setter side effects may be less useful than you’d hope:

- Side effects won’t run during the initial property assignment of stored properties in an initializer.
- Side effects can be convenient but absent a well-chosen property name, a more explicit method may lead to better readability.

Inferring get

When working with read-only computed properties, remove the get keyword and surrounding braces. In the following example, the first getter is more concise and just as effective as the second:

```
var area: Double { // yes
    return width * height
}

var area: Double { // no
    get { return width * height }
}
```

Ordering get and set

Always place your get first, followed by either set or property observers. If you use willSet, place it before any didSet observer.

Overriding newValue and oldValue

Swift setters and property observers supply predefined implicit value argument symbols. These magic arguments are `newValue` for `set` and `willSet`, and `oldValue` for `didSet`. The implicit names are always available—you don't need to take any action to have access to them—and they are instantly recognizable in context.

In addition, Swift allows you to override `newValue` and `oldValue` by supplying a name in parentheses after the `set`/`willSet`/`didSet` keyword—for example:

```
set(temperature) {
   // use temperature instead of newValue
}
```

This feature is an attractive nuisance for the following reasons:

Preferring `newValue` *and* `oldValue` *to custom names is consistent.* Both names are well chosen, intuitively obvious, and clear. You don't have to recognize a new and unfamiliar symbol in a setter or observer context. The override feature is almost exclusively used to avoid name conflicts rather than improve on default implicit names.

Preferring `newValue` *and* `oldValue` *to custom names avoids errors.* Many developers prefer to name *all* mentioned values for the sake of consistency, clarity, and readability like this:

```
set(newValue) {...}
```

Developers who name all values may accidentally insert `newValue` or `oldValue` in the wrong observer. It is not that uncommon a mistake. (See this tweet,[11] for example.) Swift does not (yet) check for name mismatches, specifically for the common error of using `oldValue` in place of `newValue` or vice versa. (I have submitted a bug report[12] and I hope this issue will be resolved.)

When using property observers and custom setters, use these guidelines:

- Prefer to retain the `newValue` and `oldValue` magic argument names. This approach enhances readability and consistency.
- Prefer to mention `newValue` and `oldValue` symbols in each `willSet`/`didSet` declaration. This is a style adopted by many experienced developers. It removes some of the magic from the implicit names by establishing a clear and readable declaration.

11. http://twitter.com/studobster/status/804064325715378176
12. https://bugs.swift.org/browse/SR-3310

- If you name symbols in observers and setters, prefer to add a custom rule for manual inspection or automated linting that specifically disallows set(oldValue), willSet(oldValue), and didSet(newValue). These three uses can all be considered unambiguous errors, and Swift really should automatically warn on their use.

An extreme set of guidelines would disallow the use of newValue and oldValue members, reserving those words for setters and observers. This book does not go so far since newValue and oldValue are reasonable property names for a generic ChangeSet<T> struct. Fortunately, Swift properly differentiates between members (self.newValue) and the newValue argument, as you see here:

```
struct ChangeItem {
    var newValue: Int = 0
    var observedMember: Int {
        willSet {
            print(newValue)
            // newValue = 100 // error, `newValue` is immutable
            self.newValue = 100
        }
    }
}

var test = ChangeItem(newValue: 0, observedMember: 50)
test.observedMember = 60 // prints 60
test.newValue // 100
```

When you encounter name conflicts, don't override the implicit names. Use self to access type members instead.

Returning Void

Prefer the Void typealias to () for types. It reads better. It also better matches return conventions in the C and Objective-C world. While this is not a particularly compelling reason for selecting a Swift style, it's not a meaningless one either. Apple is divided on whether a void function returns () or Void. *The Swift Programming Language* book refers to both -> () and -> Void. The standard library returns -> Void (and, technically, -> Swift.Void).

Some time back, an old-style Dev Forums post (now dead) clarified Apple's stance at that time:

> FWIW, we've recently decided to standardize on () -> Void (generally, () for parameters and Void for return types) across all of our documentation.

Speaking of which, don't literally return Void or () from methods or functions.

```
func myFunction() -> Void { // No, no, no
   guard someCondition else { return () }
   ... do work here ...
}

func myFunction() { // yes
   guard someCondition else { return }
   ... do work here ...
}
```

Don’t assign void returns to symbols. The results are generally unexpected, normally unintended, and probably universally useless:

```
// Constant 'myVoid' inferred to have type '()', which may be unexpected
let myVoid = print("Hello World!")
```

- Prefer Void for type names.
- Prefer () for argument signatures and values.

Styling Void

Prefer inferred Void over an explicit mention in function and method return types. The extra words and symbols are not needed. They detract from a well-understood convention that an unmentioned return type defaults to Void.

```
func f<T>(label: T) // yes
func f<T>(label: T) -> Void // no
```

Prefer explicit Void typing for closure arguments, for closure return types, and when currying. Always include the Void type in closure parameters and curried function declarations, as in the following examples:

```
func f(action: (T) -> Void) // yes
func f() -> () -> Void // yes
func f(S) -> (T, U) -> (V) -> Void  // yes
```

Avoid explicit Void typing *inside* closures:

```
let printer = { (value: Int) in print(value) } // yes
let printer = { (value: Int) -> Void in print(value) } // no
```

- Prefer inferred Void for method and function return types.
- Use explicit Void when currying and typing closures.

Using Never Return Types

Swift’s Never return type expresses the condition where a method or function will never return control to the call site. This dead-end return type replaces the @noreturn function attribute that was in earlier versions of Swift:

```
/// The type of expressions that can never happen.
public enum Never { /*no values*/ }

func foo() -> Never {
  fatalError("no way out!")
}

func usage() -> Never {
    print("Usage: \(appName) location1 location2")
    exit(-1)
}
```

Never has a limited use range: normal process exit (exit), abnormal process exit (abort, fatalError, and the like), and infinite loops. A Never return type also allows a function or method to throw—for example, () throws -> Never. Throwing introduces a path for error remediation even in functions that are not expected to return.

Using Never return types both adds developer-facing documentation about the function and allows the compiler to test for the expected behavior. For example, you might call a Never-returning function from a guard’s else clause. Every Never call can be guaranteed by the compiler to leave scope.

Prefer Never to Void return for any truly non-returning function, such as methods that wrap fatalError to perform extra logging before halting the program. A nuanced approach to Never provides a cornerstone for good diagnostic output.

```
func f() -> Never { while true {} } // yes
func f() -> Void { while true {} } // no
func f() { while true {} } // no
```

- If you’re returning nothing, prefer Void.
- If you’re never going to return, prefer Never.

Grouping Initializers

Many Apple-supplied types don’t fully set up an instance for use. They’re sourced from Cocoa APIs that predate Swift. You’re expected to create a new instance and then establish all the member-wise details that allow the instance to be properly used.

Here's one example that uses Process to perform a spotlight search on playgrounds. The launch path, the arguments, and the output pipe are all manually assigned after the declaration, creating a terror tower of successive assignment and function calls. This code suffers from excess blockiness and lacks a clear distinction between its setup tasks and instance use:

```
let task = Process() // instance creation
task.launchPath = "/usr/bin/mdfind" // setup starts here
task.arguments = ["kMDItemDisplayName == *.playground"]
task.standardOutput = pipe
task.launch() // execution starts here
task.waitUntilExit()
```

Swift offers many ways to distinguish the task setup from its execution. For example, you might use setup clauses or do statements. Here are several examples that demonstrate various approaches.

```
// Do-clause setup
let task = Process() // yes
do {
    task.launchPath = "/usr/bin/mdfind"
    task.arguments = ["kMDItemDisplayName == *.playground"]
    task.standardOutput = pipe
}
task.launch() // execution starts here
task.waitUntilExit()

// Closure setup
let task: Process = { task in // yes
    task.launchPath = "/usr/bin/mdfind"
    task.arguments = ["kMDItemDisplayName == *.playground"]
    task.standardOutput = pipe
    return task
}(Process())
task.launch() // execution starts here
task.waitUntilExit()

// Helper function setup using `with`
let task = with(Process()) { task in  // yes
    task.launchPath = "/usr/bin/mdfind"
    task.arguments = ["kMDItemDisplayName == *.playground"]
    task.standardOutput = pipe
}
task.launch() // execution starts here
task.waitUntilExit()
```

When needed, each approach enables you to scope temporary valuables used only for initialization.

Although each of these approaches introduces task-driven layout, there are recognizable drawbacks to every solution, namely indirection and wordiness. Until Swift adopts a more meaningful way to cascade setup tasks, you must create or adopt what are essentially workaround solutions.

The do clause approach is simple and parsimonious. It differentiates setup from use. The closure setup is more involved. It relies on a clever solution that executes a custom closure. This requires both an extra return statement and a late-mentioned instantiation. It offers just one real advantage over the do clause: it unifies instance creation and setup into a single assignment.

That also applies to the with helper. It moves the instance creation to the start of the call and does not require an extra return. It's my favorite approach of the three. It can be easily modified to work with a value type, allowing customization of a constant before assignment.

The with implementation used here is specific to reference type modification:

```
/// Returns a modified reference type
public func with<T: AnyObject>(_ this: T,
    update: (T) throws -> Void) rethrows -> T
{
    try update(this)
    return this
}
```

And here is a more general version of with that supports value types. This alternate approach allows creating modified versions of constants. As the Swift compiler optimizes constants, this copy/modify/store approach creates more efficient memory use than introducing variables for items that will be modified just once:

```
/// Returns `item` after calling `update` to inspect and possibly
/// modify it. If `T` is a value type, `update` uses an independent
/// copy of `item`. If `T` is a reference type, `update` uses the
/// same instance passed in, but it can substitute a different
/// instance by setting its parameter to a new value.
@discardableResult
public func with<T>(_ item: T,
    update: (inout T) throws -> Void) rethrows -> T {
  var this = item
  try update(&this)
  return this
}
```

- Avoid long columns of setup code at the top level of your current scope.
- Prefer to incorporate a visual initialization style into your preferred practices.
- No matter which style you adopt, use it consistently across your codebase.

Using Call Site Type Inference

Swift's *implicit member expression* allows the compiler to automatically infer types on your behalf. You can use implicit expression when arguments are unambiguously of a single type (T or T?) and there's a static member that returns T. Inferencing tightens enumeration and class method code and makes code easier to read.

For example, in the following code, Swift enables you to drop an explicit UIColor type mention before the .red member. In this sample, the type is fully established by the initializer's signature. Calls don't require extra clarification.

```
// For this initializer
extension UIView {
    public convenience init(backgroundColor color: UIColor) {
        self.init(frame: .zero)
        self.backgroundColor = color
    }
}

// Sample calls
let view = UIView(backgroundColor: UIColor.red) // no
let view = UIView(backgroundColor: .red) // yes
```

Inferred types work hand in hand with argument labels. The initializer's label clarifies the argument role. There's no question of what kind of thing is red. It's the background color for the newly constructed view. Good API design supports type inferencing, allowing explicit type mentions to fade from view in calls. A well-designed call drops the type mention without losing its readability or meaning.

Compare the following assignments. The fully qualified version is wordier and harder to follow. It fails the "don't repeat yourself" (DRY[13]) principle. Type names obscure each enumeration's value. Redundancy (attribute: NSLayoutAttribute, relatedBy: NSLayoutRelation) introduces unnecessary code clutter:

```
let constraint = NSLayoutConstraint( // no
    item: view1,
```

13. https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

```
    attribute: NSLayoutAttribute.leading,
    relatedBy: NSLayoutRelation.equal,
    toItem: view2,
    attribute: NSLayoutAttribute.leading,
    multiplier: 1.0,
    constant: 0.0)
let constraint = NSLayoutConstraint( // yes
    item: view1,
    attribute: .leading,
    relatedBy: .equal,
    toItem: view2,
    attribute: .leading,
    multiplier: 1.0,
    constant: 0.0)
```

Swift's parsimony by inference applies wherever the compiler supports type inferencing. For example, in switch statements, cases reduce to just the most meaningful enumeration elements. In the following example, constraint attributes (NSLayoutAttribute) emphasize their roles (left, leading, top) over their typing:

```
switch attribute {
case .left, .leading, .top:
    actualInset = -inset
default:
    actualInset = inset
}
```

- Swift inference requires fewer explicit type mentions.
- Explicit types add little to readability and a measurable quantity of visual noise.
- Adopt implicit member expression where available.
- Inferencing removes redundant or unneeded type names to create simpler, more readable code.
- Design APIs that support and reinforce type inferencing.
- Always favor succinct code.

Inferring self

Implicit member expressions enable you to omit self when the use of a member is clear and not confusing. If the compiler can unambiguously infer self from context, you don't need it in code. The following example creates two custom computed vector properties, returning the vector's magnitude and unit form.

The first of these two implementations uses explicit self. The second omits self, allowing members to be inferred from context:

```
public extension CGVector { // no
    public var magnitude: CGFloat {
        return hypot(self.dx, self.dy)
    }

    public var unit: CGVector {
        return CGVector(dx: self.dx / self.magnitude,
                        dy: self.dy / self.magnitude)
    }
}

public extension CGVector { // yes
    public var magnitude: CGFloat {
        return hypot(dx, dy)
    }

    public var unit: CGVector {
        return CGVector(dx: dx / magnitude,
                        dy: dy / magnitude)
    }
}
```

Some developers love inferred self, while others hate it. I think the result of inferred self is simpler and cleaner. It provides functionally equivalent implementations of the magnitude and unit properties. Using inferred self also emphasizes your use of self in escaping closures, helping you locate potential problem points when debugging retain cycles.

Required self

Ubiquitous self adds visual noise that can be omitted except under these two circumstances:

Retain explicit self where it's necessary for the code to properly compile or express itself (in escaping closures and initializers). You generally don't need self except in (some) initializers and escaping closures:

- In initializers, self differentiates between local arguments and type members. It's common to pass same-name arguments to initializers, so self makes it clear whether you're referring to the instance member or the argument.
- Marking closures as @escaping forces you to explicitly refer to self to clarify capture semantics. The compiler proactively reminds you when those references are needed, so err on the side of simplicity. Let the compiler

prompt you to restore self reference to ensure you've limited its use to the most necessary cases.

Retain explicit self where self is clearly the subject of a call. For example, consider if self.hasValue(...). Omitting self makes you ask, "What has the value?" Code that makes you stop and ask questions instead of showing you the answer is failing in its job. Prefer to retain self when a method call reflexively refers back to an instance. In the following example's where clause, the contains method call lacks an explicit subject:

```
for (flag, description) in descriptions.enumerated() // no
    where contains(
        Features(rawValue: 1 << flag)) {
        ...
}
```

What is doing the containing? A container isn't included as a method parameter, so it has to be the calling instance. Fully qualifying the call clarifies the subject ambiguity and vastly improves readability:

```
for (flag, description) in descriptions.enumerated() // yes
    where self.contains(
        Features(rawValue: 1 << flag)) {
    ...
}
```

Considering self in Code Review

When reviewing code, it's not unusual to be presented with material that lacks syntax highlighting. Syntax highlighting is an important IDE feature for some developers, and without it, it can be harder to detect the owner of a self-less property. House styles that adopt mandatory self support review without needing the IDE hints that some coders use.

At the same time, the more your group adheres to a "no global anything" rule, the less you need to enforce mandatory self in any situation. If a symbol cannot be global due to house rules, its natural owner will be the containing type.

- Omit self, except where it's needed for clarity or practicality. I think mandatory self adds unneeded code cruft. Its benefits (in readability and code correctness) aren't sufficient to make it worth adopting for most developers.
- Mandatory self has been adopted by a non-negligible number of large and worthy organizations.
- Preferring to limit global symbols can mitigate the need for mandatory self.

Evaluating Case-Binding Syntax

Swift case binding is one of the least straightforward components of the entire language. Case binding enables you to match an enumeration case and then bind that case's associated values to new constants and variables. Where you place those let and var keywords to bind case values involves nontrivial choices.

Safety and consistency play important roles in these decisions. You must choose whether to favor readability or error prevention, even though the likelihood of errors is quite small.

This section explores two distinct styles—external and internal case binding—and details the advantages of each. I recommend you adopt a consistent internal style, which was not my practice prior to writing this book. In researching this topic, I learned that you can safely navigate a variety of pitfalls by preferring this less-attractive but more-reliable style.

External Case Binding

External case binding places a single let or var keyword outside the enumeration case. I warn you that I'm going to recommend against this practice, but before I do, here's how external binding works.

Moving keywords out of their tuples combines binding into a single additional keyword. The results use fewer characters and create a simpler syntax. Here are some examples that showcase both approaches:

```
enum StatusCode {
    case status(code: Int, message: String)
    case error(Error)
}

let fetch: StatusCode =
    .status(code: 418, message: "I'm a teapot")

if case .status(let code, let message) = fetch { // more words
    print(code, message) // 418 I'm a teapot
}

if case let .status(code, message) = fetch { // fewer words
    print(code, message) // 418 I'm a teapot
}
```

The external version produces a consistent code style for every binding, whether or not you ignore individual values. These next examples demonstrate the uniform presentation of prefixed let/var keywords. The style is uncomplicated and won't change based on the number of associated values or bound symbols:

```
enum Value<T> { case one(T), two(T, T), three(T, T, T) }

let example1: Value<Character> = .one("a")
let example2: Value<Character> = .two("a", "b")
let example3: Value<Character> = .three("a", "b", "c")

if case let .one(a) = example1 {}
if case let .two(a, _) = example2 {}
if case let .two(a, b) = example2 {}
if case let .two(_, b) = example2 {}
if case let .three(a, b, c) = example3 {}
if case let .three(_, b, c) = example3 {}
if case let .three(_, _, c) = example3 {}
```

Internal Case Binding

Internal case binding is not as pretty or consistent as its external alternative, but I've come around to adopting this style. While the internal approach involves extra syntax for each bound symbol, it is safer. Consistent internal binding, as in the following example, avoids errors introduced by a variety of uncommon edge cases:

```
if case .three(let a, let b, let c) = example3 {}
```

When pattern matching, it's common to bind a variable or constant and uncommon to use a bound value as an argument. Despite this rarity, adopting an "always explicit, always within the parentheses" rule adds consistency and safety to your code.

The following example showcases an always-internal binding style used with an externally bound symbol. Under this style, binding is limited to each keyword's site. The oldValue constant will not be changed by the if-case statement:

```
let oldValue = "x"
...
// This safely binds and simultaneously matches.
if case .two(let newValue, oldValue) = example2 {
    ...
}
```

Consistent in-place binding avoids the accidental shadowing demonstrated in the following example. Overbinding shadow errors cannot happen when you adopt universal internal binding:

```
// This is an error because the intent is
// to bind newValue and match oldValue
if case let .two(newValue, oldValue) = example2 {
    // Wrongly matches "a", "b".
    //
    // `oldValue` is shadowed here, assigned the
    // value from the second field of the
    // enumeration's associated values.
    ... use newValue ...
}
```

Admittedly, this is an outlier case. Pattern matching rarely uses already bound values. If you've adopted an external binding style, you can express this situation with a separate and explicit where or comma-delimited condition clause, as in the following example. This code introduces additional syntax and adds an extra variable binding (currentValue):

```
// This implements pattern binding and matching
// the given value but the extra condition separates
// these into two distinct goals
if case let .two(newValue, currentValue) = example2,
    currentValue == oldValue
{
    // correctly won't match "a", b"
    ... use newValue ...
}
```

Even here, safety can be problematic. If you inadvertently pass a wrong value to the condition clause, you'll introduce a hard-to-find error. In the following code, I've accidentally typed newValue when I meant to type oldValue. This code will compile, but its logic is flawed. Consistent internal binding avoids this error, too.

```
// This error (`newValue` instead of `oldValue`) will not
// be caught by the compiler and is hard to catch by
// inspection.
if case let .two(newValue, currentValue) = example2,
    currentValue == newValue
{
    // correctly won't match "a", b"
    ... use newValue ...
}
```

Internal binding can be easier for new language adopters to read. Even without binding, if case is confusing. Both if case let and if case var (plus case var and case let) may look like single compound keywords rather than a combination of two distinct actions to developers unfamiliar with this syntax.

There's one final reason to adopt always internal binding. When you need to mix let and var binding, you *must* use internal binding. This example shows how that might look in your code:

```
if case .three(_, let b, var c) = example3 {}
```

- Prefer consistent internal let and var binding. It's safe and simple.
- There is but *one let to rule them all and in the Swiftness bind them.* Use that one let internally and generously.

Ignoring Associated Values

When you want to match only on the case, omit wildcard patterns and just mention the enumeration case. This allows you to ignore the content and structure of the payload and focus strictly on the enumerated conditions provided by the type:

```
switch statusCode {
case .error(_): // no
    ...
case .status(_, _): // extra no
    ...
}
```

```
switch statusCode {
case .error: // yes
    ...
case .status: // yes
    ...
}
```

Using if/guard-case

Both if-case and guard-case are most valuable when used to bind associated values to variables. They allow you to reach inside an enumeration and access those values based on an enumeration's specific cases:

```
enum Either<L,R> { case left(L), right(R) }
guard case .left(let value) = result else { return } // yes
print(value)
```

When pattern matching, prefer the ~= operator to if-case and guard-case. The following statements perform identical tests, checking whether myValue falls within a range:

```
if range ~= myValue { print("success") } // yes
if case range = myValue { print("success") } // no
```

Of these, the first is readable and simple. The second, while logically identical, is esoteric and hard to process. Using the equal sign when there's no assignment or condition binding promotes confusion for most Swift developers, even those experienced with pattern-matching syntax.

While coders can be trained to recognize this rare use, you cannot be sure that anyone reading your code will have that training. This rule falls under the mantle of Brian Kernighan's admonition against writing clever code.[14] Prefer the obvious to the obscure, no matter how cool you consider the alternative to be.

- Reserve if-case and guard-case to bind variables.
- Prefer ~= for non-binding pattern matching.
- "Too clever is dumb."—Ogden Nash

Choosing Capture Modifiers

Prefer weak capture to unowned. Weak captures can be checked at the start of a closure and, if still valid, assigned to a strong reference for the life of the enclosing scope. They provide excellent safety and utility.

An unowned capture is equivalent to Objective-C's unsafe-unretained; it can be used when "the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time." You may use unowned items to refer to global instances whose lifetime extends throughout the application's duration.

An unowned reference guarantees that a reference *always* has a value. If there is any chance whatsoever that the unowned item may deallocate, using unowned items is roughly equivalent to and as desirable as using a weak value with forced unwrapping. The advantage—if that's the right word to use—is that an unowned reference is not an optional value and can be used without further unwrapping.

When capturing weak self in a closure, prefer to create a strong reference at the start of the closure rather than prefix all calls with self?.member optional chaining. Introducing an early strong reference ensures that self cannot deallocate throughout the lifetime of the closure's scope. The guard self construct introduced in Swift 4.2 allows coders to avoid the strongSelf/weakSelf dance of earlier language releases:

```
[weak self] in
```

14. https://en.wikiquote.org/wiki/Brian_Kernighan

```
guard let self = self // yes
    else { ... leave scope ... }
self()
self()
```

```
[weak self] in
self?.f() // no
self?.g()
self?.h()
```

- weak capture is generally safer than unowned capture.
- If you must use unowned, reserve unowned capture for items with guaranteed lifetimes.
- While singletons can be seen as good candidates for unowned capture, since their existence is guaranteed throughout the lifetime of the program, there are no real worries about deallocation.
- Capturing a weak version of self ensures you won't create reference cycles in your code that prevent memory from properly deallocating.

Toggling Boolean Values in Place

Prefer in-place toggle() over writing out a long property path on both sides of an equal sign. The former is wordy and violates the principle of "don't repeat yourself." The latter is succinct and readable:

```
myVar.prop1.prop2.enabled = !myVar.prop1.prop2.enabled // no
```

```
myVar.prop1.prop2.enabled.toggle() // yes
```

Testing Sequences for Boolean Logic

Prefer Sequence.allSatisfy, which was introduced in Swift 4.2, to contains, reduce, or filter. The resulting code is more readable, more natural, and more performant. The allSatisfy method confirms that every sequence member matches the criteria you specify, using positive terms to express those criteria. With allSatisfy, you can say "every member is odd" rather than "does not contain a non-odd number."

Consider the following examples. Using contains and reduce requires mental overhead that distracts from the code's primary goal: to determine whether every member meets a given test. allSatisfy brings these semantics to the foreground and provides a more comprehensible and maintainable form.

```
// every element is 9 // no
!nums.contains(where: { $0 != 9 })
// every element is odd
```

```
!nums.contains(where: { !isOdd($0) })

// every element is 9 // no
nums.reduce(true, { $0 && $1 == 9 })
// every element is odd
nums.reduce(true, { $0 && isOdd($1) })

// every element is 9 // no
nums.filter({ $0 != 9 }).isEmpty
// every element is odd
nums.filter({ !isOdd($0) }).isEmpty

// every element is 9 // yes
nums.allSatisfy({ $0 == 9 })
// every element is odd
nums.allSatisfy(isOdd)
```

For performance, allSatisfy is the winner over repeated tests, with filter being the worst possible choice, as it requires many allocations. These following results represent tests on arrays of 1,000 random members, performed 100,000 times on my creaky old 2012 Mac mini:

```
Running 100000 tests on an array of 1000 random values
Ending Test: contains
Elapsed time: 0.0005732100107707083
Ending Test: reduce
Elapsed time: 0.21395097096683457
Ending Test: allSatisfy
Elapsed time: 0.0002882789704017341
Ending Test: filter
Elapsed time: 1.4889178200392053
```

As for readability, allSatisfy is the only approach that uses a single easy-to-read and easy-to-construct positive logic test. contains requires a double negative, and filter a single negative plus a test for empty. reduce requires constant combinations with existing truth values.

Double-Testing Boolean Logic

Avoid comparing Boolean values with Boolean literals. The test

```
if condition == true { ... }
```

is programmatically identical to

```
if condition { ... }
```

and serves only to add an unnecessary second test.

Some developers argue against this advice. They prefer to add the explicit test to make it more visibly obvious exactly what is being tested. They believe a leading bang/exclamation point (!) can easily get lost when performing code reading.

Boolean Optionals

Avoid typing values as optional Booleans wherever possible. Although there are valid real-world use cases (for example, when working with web-sourced JSON), optional Booleans normally suggest a point where code can be better designed.

For example, consider a dictionary that stores simulator IDs against a Boolean value. The value indicates whether the simulator is currently booted. This dictionary is better represented as a mutable set of booted simulator IDs. You can add and remove IDs from the set as the states change and test whether a simulator is booted with `contains`.

In the rare case that a function or method returns `Bool?`, test with nil coalescing against `false` to treat the `.none` case as false:

```
if functionReturningOptionalBool() ?? false { ... }
```

In all other cases, treat optional Booleans as a warning sign. It indicates code that should be evaluated more closely.

Using Division Logic

It's a rule of thumb in many programming languages to test for even or odd values against zero. You don't want to deal with modulo values that may be negative, which is the case when testing against 1 (or possibly -1). The same rule applies to any test for even division against arbitrary divisors. In Swift 5 and later, prefer to use the `isMultiple(of:)` over explicit modulo tests.

```
let isOdd = (value % 2) == 1  // no! (especially for negatives)
let isEven = (value % 2) == 0 // no longer the better solution

let isEven = value.isMultiple(of: 2) // yes
let isFizz = value.isMultiple(of: 3) // yes
let isBuzz = value.isMultiple(of: 5) // yes
```

Other Practices

Here are some handy ways to help optimize and improve your code:

- Prefer `let` constants to `var` variables unless you truly need mutable data. The compiler can better optimize your data when it's guaranteed not to change. The Swift compiler is very good at detecting unmodified `var` use.

- Prefer Swift's built-in random number system (for example, `let d20 = Int.random(in: 1 ... 20)`) over external libraries unless you need specific control with regard to seeding, distributions, and random sources. Avoid `rand`, `random`, `arc4random`, `arc4random_uniform`, and so on entirely.
- Marking classes as `final` enables the compiler to introduce performance improvements. These improvements depend on moving away from dynamic dispatch, which requires a runtime decision to select which implementations to call. Removing indirect calls for methods and property access greatly improves your code performance.
- Swift's whole-module optimization can automatically infer many `final` declarations by scanning and compiling an entire module at once. Its inference capabilities apply only to constructs and members marked as `internal`, `fileprivate`, and `private`. Class members with `public` access must explicitly declare `final` to participate in this optimization.
- When working with debug builds, prefer incremental to whole module to increase compilation efficiency. The compiler can process each file separately, leading to quicker compilation time.
- Consider optimizing for size (`-Osize`) when building for release. The code will run slower than speed-optimized builds (`-O`) by about 5%, but you can reduce code size by 10–30%, a huge improvement for App Store deployment.
- Avoid escaping closure arguments unless they must outlive your functions. Using non-escaping closures (the default) introduces performance optimizations and bypasses the need to annotate properties and methods with `self`.
- Default arguments involve a minor check with a small overhead cost. Unless you plan to run tens of millions of calls at once, don't let this overhead sway you away from providing defaults. Using default values, even for arguments typed as closures, involves minimal overhead and significant usability gains.
- Avoid complex string interpolation that performs too much work within the `\()` interpolation delimiter. This is best broken out into separate statements to improve readability and maintainability.
- Prefer `Array(mySequence)` to `mySequence.map({ $0 })` when converting a sequence to an array. The former is easier to read and the intent is clear from the start of the line. Transformation, not coercing a sequence to an array, is `map`'s primary mission statement.

Counting

Don't test a collection's count property. Prefer checking isEmpty over count == 0 or string == "". The isEmpty property returns a Boolean value indicating whether the collection contains no elements. It operates in O(1) time in most (but not all) cases. The standard library recommends using isEmpty over testing count, especially for computed and potentially infinite sequences, which is where count can be particularly expensive.

isEmpty is not always O(1). A lazy filter with no matches must run through its entire sequence before determining isEmpty (which is O(N) complexity). isEmpty should be O(1) for all non-lazy collections and for many lazy collections too, such as lazy maps.

Prefer collection.count(where:) over collection.filter{ ... }.count. The former is simpler and more direct, using a single operation instead of a chain.

Avoid optional collections except where absolutely necessary. Prefer an empty collection to nil. The empty collection can be tested with isEmpty and provide all the normal collection behavior without introducing optional tests. If you must accept an optional, consider nil coalescing to immediately convert the nil case to the empty collection.

Sets

Swift sets are a much overlooked type. They are, to use an American metaphor, the "Jan Brady" of native Swift types—the worthy but humble middle child of collections. Developers often select arrays in preference to sets, even when they're just collecting members and testing membership.

Question any code that looks like this:

```
Array(Set(a).intersection(Set(b)))
// or
a.filter(b.contains)
```

When you're using an array as a set, maybe you should be using a set to begin with. Where possible, prefer more restricted types that offer a better match to your problem domain. If you require ordering or counts, Cocoa Foundation offers NSOrderedSet and NSCountedSet/CFBag, or you can create your own Swift-native implementations of these types.

When using sets, take advantage of the full range of set operators. For example, prefer isDisjoint(with:) to testing whether an intersection is empty:

```
numberSet1.intersection(numberSet2).isEmpty // no
```

```
numberSet1.isDisjoint(with: numberSet2) // yes
```

Many codebases include examples of workarounds that are actually core Set operations already included in the language. If your set code involves loops, check the core documentation to see whether calls like symmetricDifference or subtracting might be what you're looking for.

Avoid using sets to remove duplicates when retaining order is important. Swift sets are inherently unordered, although NSOrderedSet exists in Cocoa.

Moving into Swift

For many Swift developers, there's a real challenge of moving on from other languages. Moving into the Swift ecosystem from a different language means rules, constructs, and conventions that create a break from guidance specific to those other development environments. Coders have adopted styles and practices that don't apply to Swift. Here's a quick review of some habits you'll want to step beyond to create Swiftier code:

camelCase: adopt camelCase for allTheConstants, not ALLCAPS or SCREAMING_SNAKE, not kebab-style, and not lower_snake. Swift minimizes constant use, prefers to wrap them into parent types for namespacing rather than defining them globally, and doesn't offer a macro system.

Native initializers: prefer Swift initializers to legacy ones, even when they still compile. CGPoint(x: 1, y:1) is a Swiftier initializer than CGPointMake(1, 1).

Conditional parentheses: avoid parentheses around if and switch conditions unless the switch pattern is an actual tuple. Conditional parentheses are not required in Swift.

Return parentheses: avoid parentheses around return values. They are not required in Swift and tend to show up as a reflex held over from other programming languages.

Prefixes: do not namespace your types, your methods, your functions, your constants, or anything else with Objective-C style prefixes. NS is a dead prefix walking. Prefer to namespace by embedding constants, functions, and the like into static type members.

Using "call by reference" style with copy-in-copy-out: avoid mixing inout parameters with return values unless you have a strong motivation with an excellent rationale to back up your decision. This pattern often arises from too-literal migration from Objective-C &error-style Cocoa calls. Instead, move to thrown errors or refactor to incorporate a success/failure result enumeration or a tuple to provide a single return style. It's a red flag to return values both within a function's parameters and through its return type.

Terminal semicolons: avoid reflexive terminal semicolons. They imply copy-and-paste coding style even when they're a positive style choice.

Instance Creation

When given the choice of creating instances via a initializer, a stand-alone global function, or a factory method, prefer the initializer in most cases. Factory methods are rarely required in Swift and are often better addressed using convenience initializers. You might require a factory method when you need to create a class cluster, where you build a public class with private subclasses, and have the public class offer a static method to return the appropriate private instance. This is a relatively uncommon pattern in Swift.

Breaking Down Complexities

Where possible, avoid the following:

- Excessive conditional statement nesting (`if-if-if-...`)
- Excessive functional application nesting (`f(g(h(...)))`)
- Giant statements that can be broken down into simple components

Each of these represents a refactoring opportunity. Break code into more sensible elements using smaller names, better condition constructs (such as `switch` statements with tuples), and multiple statements.

Adopting these guidelines and checks into your preferred practices provides opportunities to improve your codebase.

Using Conditions as Results

Keep your eye out for `if` statements that assign truth values in both the positive and negative clauses to the same variable or property. In the following example, the `hidden` property is set by hand, although it ties directly into the evaluation of the initial condition:

```
if i <= visibleCells { // no
    cell.reloadData()
    cell.hidden = false // assignment
} else {
    cell.hidden = true // assignment
}
```

These truth values represent the evaluation of the initial condition but with reversed values. The coder wanted to put the heavier clause on top and so evaluated <= visibleCells and assigned the opposite value. This code is unnecessarily indirect and verbose.

To refactor, assign the condition directly. The following code flips the logic to set the value of hidden. It can then use this value to choose whether to reload data. The result is simpler and more harmonious:

```
cell.hidden = i > visibleCells // yes
if !cell.hidden { cell.reloadData() }
```

Colocating Declarations

Many developers coming from C-like languages want to place their declarations at the start of a scope and then use them throughout. Here's an example where a declaration is used once but is separated from its single use point.

```
// Notification declaration
let note = Notification(name: .textControl, object: entryView)

if let handler = entry.signalHandlers[.focusIn] {
      g_signal_handler_block(widget, handler)
}

Application.shared.firstResponder = entryView

// Notification use
entryView.delegate?
    .controlTextDidBeginEditing(notification: note)
```

Prefer to move single-point declarations to their point of use. Someone reading this code sees the notification's creation and then its consumption as a single thought:

```
...
let note = Notification(name: .textControl, object: entryView)
entryView.delegate?
    .controlTextDidBeginEditing(notification: note)
...
```

Adopting Functional Programming

Look for opportunities to refactor summations to reduce calls. Code that sets a sum variable to zero and then uses a for loop to add items is a likely match for reduction. In a similar vein, prefer to map (map, compactMap, flatMap) or filter (filter) an array to transform its contents. Candidates for refactoring start with an empty array variable and iteratively add transformed or selected items:

```
var totalWidth = 0 // no
for i in 0 ..< columns {
      totalWidth += columns[i].width
}
```

```
let totalWidth = columns.reduce(0, { $0 + $1.width }) // yes

var selectedItems: [ColumnCell] = [] // no
for item in items {
    if item.selected {
        selectedItems.append(item)
    }
}

let selectedItems = items.filter({ $0.selected }) // yes
```

Wrapping Up

You've now read about ways to adopt house approaches for typical coding tasks. You learned how strong recommendations enable organizations to adopt consistent approaches when faced with common challenges like type inference and mandatory `self`. Although it's impossible to offer exhaustive examples, this chapter should inspire you to consider the many opportunities to adopt such rules into your style guide. You should be able to recognize the kinds of repetitive judgment calls that lend themselves to this kind of guidance.

You've explored how to make Swifty choices for getters and setters. You've read ways to initialize types and structure statements. You've considered how to improve safety and the different ways to produce sanity checks through assertions, preconditions, and `guard` statements. You've seen several examples of structure that you may not have thought about before.

The next chapter turns away from implementation concerns to focus on good API design. An API is the public-facing presentation of your code. Read on to discover how to write the best possible interfaces that enable your code to be consumed by a more expansive audience.

CHAPTER 4

Design the Right APIs

Application programming interfaces (APIs) enable you to make use of rich implementations such as graphics, web access, and data manipulation, without the headache of building these from scratch. They grow and support development by offering simple calls to complex functionality.

The API establishes a surface: a boundary between call sites and underlying functionality. It abstracts away implementation details, presenting a limited collection of visible features. An API's scale and scope vary by the code it describes. It can detail a single software component with a distinct limited role, or it can catalog an entire framework or library.

API design is both an art and basic common sense. A well-designed API provides a clear and understandable set of tools with well-chosen names and thoughtful consideration to resilience and long-term code evolution. It establishes an abiding calling contract. An API carefully describes the types, methods, and results offered by its underlying implementation and how these will be expected to behave and perform.

Swift is an opinionated language, and its APIs have opinions too. There are Swift ways to design APIs: presenting functionality in a natural, native, and idiomatic way. Swift guidelines[1] emphasize the principles of clarity, concision, and utility. They promote industry standards of form, simplicity, and affordance. A good API rejects ornamentation and complexity. You should provide solid and well-considered justifications for every element the API exposes and declares.

This chapter introduces topics of API design, helping you refine the way code presents itself for internal and external consumption. From access control to

1. https://swift.org/documentation/api-design-guidelines/

naming, from nesting to defaults, you'll discover how to offer safe, consistent, and meaningful functionality.

Adopting Access Control

Access control restricts type and symbol visibility from parts of your code to code in other types, files, and modules. A symbol can be completely visible or hidden at varying access levels. Using access control hides implementation details and establishes a preferred interface. Introduce access control early in your development process and not as an afterthought.

Access Control

Before diving into specific guidance, here's a quick overview of Swift's access control keywords and related terms:

public

- A public item (including types, members, protocols, and so on) can be accessed outside its defining module.
- A public class can be accessed but not subclassed outside its defining module.
- A subclass cannot override a public member outside its defining module.

open

- An open member can be accessed outside its defining module.
- An open class can be subclassed outside its defining module.
- A subclass can override an open member outside its defining module.
- You can think of the open access level as "more public than public."[2]

final

- A final class cannot be subclassed, regardless of scope.
- When applied to a class (final class), the final keyword marks the entire class as final, including all its members.
- A final method, property, or subscript cannot be overridden by a subclass (for example, final subscript, final func, and final var).

internal

- An internal item can be accessed within its defining module.

2. https://lists.swift.org/pipermail/swift-evolution-announce/2016-July/000268.html

- An internal item will not be visible or usable outside its defining module.

fileprivate
- A fileprivate item restricts use to its defining source file.
- A fileprivate item will not be visible or usable outside its defining source file.
- fileprivate hides implementation details from other files within the module.

private
- A private item restricts use to the enclosing declaration and to type extensions co-located in the same source file.
- A private item will not be visible or usable outside the enclosing declaration and same-file extensions.
- Using private hides implementation details that are specific to a single declaration.
- Using private enables you to add hidden nested types and functions or hide internal members.
- Same-file visibility of private members does not extend to subclasses, only to same-type extensions.

@testable
- The @testable attribute allows a unit test target to access any internal item.
- Without the @testable attribute, access is limited to public and open members.

Subclassing

Think about potential subclassing as you design your classes. Could your class be subclassed at some point in the future? Should it be? Class members that can be safely viewed, modified, and called in a parent class might break if subclasses override elements *required* by the root class. Bring this consideration into your design process to support robust API development.

Consider these points:

- A final class cannot be subclassed. If you can reasonably designate classes as final by default, prefer to do so. You can remove the final designation later if you decide to allow subclassing. Never remove subclassing from published APIs.

- Mark classes as public rather than open unless or until you make an intentional choice to allow consumer subclassing outside your module.
- Swift subclasses can override class members. They cannot override static members. Prefer static members to class members, except when you specifically permit overrides.
- It's uncommon to create pure Swift classes that are intentionally subclassable. Developers more commonly subclass types sourced from Cocoa and open source libraries than ones created natively in Swift. Make your subclassing decisions early and intentionally, taking member visibility, overridability, and other related issues into account. Err on the side of final and public types, which can be deliberately opened up at a later date.
- Disallowing subclassing can interfere with unit testing. Marking imports as @testable won't allow you to bypass this issue. Faced with this reality, you may need to allow subclassing even though Swift purism suggests a class should be final.

Placing Access Modifiers

Access modifier keywords declare the visibility of Swift elements between types, files, and modules. A modifier hides or exposes source implementation details from code that consumes your APIs. You may specify access levels for types and their members as well as freestanding constants, variables, functions, and protocols. Create and place access modifiers with thought and care. Well-considered modifiers support code reading and validation.

Leading with Access

Adopting an access-first style creates a consistent and predictable pattern of declarations. This approach enables you to check for access control by scanning the left-most column of text. When you don't see an access control modifier, you can be sure the symbol either defaults to internal access or represents unaudited code in shops that adopt mandatory control declarations.

The access control token needs to be placed before variable and constant declarations:

```
let fileprivate x = 10 // error
fileprivate let y = 10 // compiles
```

The following lines are legal and will compile. Swift allows you to interchange modifiers with access control tokens for static or class declarations. Prefer to lead with access control:

```
open class MyClass {
    open class func foo() {} // yes
    class open func bar() {} // no
    private static func foo() {} // yes
    static private func bar() {} // no
}
```

The following modifier order is common. As always, adjust the order as needed for your house style:

- at-prefixed modifiers (@objc, @nonobjc, @IBAction, @IBOutlet, and so on)
- access control level (open, public, internal, fileprivate, private)
- setter access control level (for example, private(set))
- override

Setter access control levels (ACLs) normally follow the main ACLs. It's uncommon to break these apart, although some do.

Some developers treat override closer to an at-prefixed modifier, placing it before access control instead of after.

Any class or static modifiers (that is, declaration modifiers rather than class type) can be placed after the access control level (common) or before (less common). They often gravitate to member keywords (like var, let, func, case, and so on). Some developers prefer to place ownership modifiers between them and the member keywords—for example, static unowned let vs. unowned static let.

The following modifiers continue declarations. Here is one ordering you might adopt:

- dynamic
- mutation (mutating, nonmutating)
- lazy
- final
- required
- convenience
- ownership (weak, unowned, and so on)

Declaring Internal Levels

Swift's default access level is internal. Some developers omit explicit internal declarations, although spelled-out modifiers is the generally accepted best practice:

```
public class Window {
    public var frame: CGRect = .zero
    public var title: String = "Untitled Window"
    var backingStore: Data? = nil // omitted `internal`
}
```

Reserve defaulted internal for utility programming and language learning. A simple type like struct Point { var x, y: Double } is sufficient in the classroom *before* teaching access control. If the default were private, you wouldn't be able to engage with these simple examples without first exploring access concepts.

When writing libraries for third-party consumption, explicit declaration offers the following benefits:

Ensuring correctness. The internal keyword expresses that you've made an intentional decision about access. You're not just accepting a default behavior. The comment in the following snippet is overkill. Without mandatory access control mentions, it might have been necessary.

```
internal let symbol = value // Access has been audited
```

Consistency. Include internal when your style guide requires explicit access control for all API members. The internal keyword appears in the column on the left side of the member list like all others, making this check easy to scan and confirm.

```
public struct Point {
    public let x: Double
    public let y: Double
    internal var mirror: (Double, Double) {
        return (-x, -y)
    }
}
```

When excluding the internal keyword, you may end up with some short lines. If column alignment is a realistic concern, you're possibly programming in the wrong language.

Convention. Explicit access control has been adopted by both the standard library and Swift Foundation house styles. There are few better places to draw style inspiration from than Apple itself.

Extensions and Access Control

Swift automatically adopts any access control specified for an extension and applies it to the top-level declarations within that extension. For example:

```
public class Foo { ... }
public extension Foo {
    public class Biffle { // warning: 'public' modifier is redundant
                          // for class declared in a public extension
        public let π =  Double.pi // no warning issued
    }
}
```

Do not annotate extensions with access control levels except when working with trivial utilities. ACL-free extension declarations ensure that you can meaningfully and intentionally add access control to each member declared within those extensions. Each access level will be co-located with the declaration it decorates. This makes your code more easily audited, and its access levels will be immediately apparent as to intent and implementation.

Be aware that following this protocol has drawbacks under the current state of Swift. Consider the following two examples:

```
internal class Bar { ... }

extension Bar {
    public func groot() { ... } // this is still internal
}

// vs.

public extension Bar { // error: extension of internal class cannot be
                       // declared public
    public func groot() { ... } // warning: public modifier is redundant
}
```

- The first code snippet compiles without error or warning. The supposedly `public` member will actually be internal.
- The fully audited second example generates both an error (as you cannot extend an internal class with a public extension) and a warning (that the public modifier is redundant).

While an error helps expose the mismatch between the core type's control level and the extension, neither the missing feedback in the first example nor the warning in the second is germane to the actual issue in the member's control level: a member declared `public` will be `internal`.

For now, use a good linter and code reading support to ensure consistency and avoid the "unintentionally internal" issue. Mark up each declaration and skip extension access levels (except for trivial use cases, in which case, it's easier to type `public` once).

Exposing APIs

You determine what your public API looks like by deciding which types, constants, and other features you want to expose outside your module. You can always expose more details but can never cover them back up. Once you've published, avoid changing access control to a less-visible or non-subclassable level.

When consumers depend on published API details, you will break code by adopting stricter policies. Limit your changes from `private`, `fileprivate`, or `internal` access to `public`, from `final` to non-final, and from `public` to `open`.

When removing an existing API, justify your breaking actions. Consider whether the status quo produces measurable harm or if it's notably contrary to the mission of your library. Document and explain your deprecations and give your API consumers time to respond to deprecations.

Ask yourself: do you want to commit to supporting this API for the next few decades? Are you willing to have these symbols persist with the names you've chosen? Can you safely freeze the API's behavior so it will never have downstream impact?

Prefer to minimize your public surface area. A limited API requires the smallest contract between you and the API consumer. It incurs fewer responsibilities and lower maintenance costs. Once you've published an API, you must support it. Don't think, "What functionality can this offer?" The baseline for API inclusion should never be whether something is nice to have. Prefer to solve real problems, not theoretical ones.

Avoid the "utilities" trap. Many utility libraries offer every bell and whistle possible without being motivated by real-world use cases. It is better to respond to user requests, incrementally expanding your APIs, than to try to pre-guess your audience and include too many. The best utilities libraries are ones you use yourself and have curated over time. Prefer the most valuable routines over the most edge-cased ones. You can always expand your API contract in a future update.

Justify each member. Before you include a feature, make sure it's useful. Express how it provides significant utility *before* exposing it for use. A good feature emphasizes safety, speed, and expressiveness. Forcing yourself to document your ideas resists API clutter and makes you think through the question of "What purpose does this serve?" If you cannot argue coherently for and provide examples of an API's fundamental utility, the feature probably isn't strong enough to expose.

Preparing Public APIs

API design is often a rolling process rather than a static endpoint. Because of this, you should prepare types for public use before you actually mark them public. Swift permits you to incorporate public members into internal types, as you see in the following example. This member will not be visible until the parent type itself becomes public:

```
internal struct InternalType {
    public var publicMember: String
}
```

This approach enables you to consider, evaluate, and refine API design before converting an internal type to a public one. You can evolve your public members more flexibility, with a longer period of review before metaphorically pulling the switch. Changes to your design will not break existing API contracts, where code external to your module relies on a fixed and published API. Instead, you'll have the freedom to incorporate these pending types into live code when you're ready to move forward with your vetted design.

Freezing Enumerations

While it's inconsiderate to change most APIs, it's unforgivable to add new cases to (or, worse, remove existing cases from) enumerations. A published enum should be fixed, with its cases obvious, contractual, and eternal.

For example, insets can be "top, bottom, left, and right." It's a good match to an enum type. Color channels can be "red, green, blue, and alpha" or "hue, saturation, brightness, and alpha." A sign can be "negative, zero, and positive." These canonical cases will never change and you will never add to them. In the case of color channels, you might publish *additional* enumerations, but you're not changing the enumeration cases for existing channel styles.

Try to never publish an enum API that can change over time. Enumeration cases are compiler checked. Client code relies on those cases being complete and exhaustive. If you might ever extend your list of privilege levels, your available polygon styles, or your key encoding strategies, you should not design or implement them using enumerations. For example, adding a new `convertFromKebabCase` strategy to your JSON decoder *should not break code* that exists and depends on your published APIs. If you're writing a type that has any possibility to extend its semantics, avoid enumerations.

When your cases are not fixed at design time, reconsider your approach. What looks like enum and smells like enum might actually be better implemented as an option set or even as a well-designed struct with static members.

Alternate typing allows you to mimic an enum's state-wise cases while providing a design space that can grow and change over time.

The following enumeration does not represent all possible HTTP methods. It does not even represent the set of all "common" methods.[3] If your client may ever need PATCH, OPTIONS, or HEAD (to name a few), encoding an enumeration means you're already looking at an inherently broken API:

```
enum HTTPMethod: String {
   case get = "GET"
   case post = "POST"
   case put = "PUT"
   case delete = "DELETE"
}
```

Don't break your API contracts at the moment of design. If your enumerated cases may change, avoid enumerations.

Privacy Hints

Swift adopts and builds on a set of style hints found in Objective-C. An underscored name indicates a private member or symbol. The Swift Foundation coding style page[4] writes, "Prefix private or internal functions, ivars, and types with an underscore (in addition to either the private or internal qualifier). This makes it very obvious if something is public or private in places where the function or ivar is used."

The rules of underscoring go like this:

- Hide private implementation details from other types.
- An underscore always indicates either limited scoping ("this is privately scoped") or overstated scoping ("this is publicly scoped but not meant for public consumption").
- Avoid underscore prefixes for any public symbol intended for general use.
- Underscores are not limited to properties and constants. Apply them to any symbol that communicates "for internal use only."
- Swift underscore use is distinct from POSIX and GNU underscore rules.
- Underscores lead, not follow.

3. https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
4. https://github.com/apple/swift-corelibs-foundation/blob/master/Docs/Design.md

- Underscores are not used, checked, or mandated by the compiler. The compiler will not emit warnings about them. You can use a linter and custom underscore-checking rules to augment the compiler.

```
internal let _element: Element // yes
internal let element_: Element // no
```

Contrasting Privacy and Hungarian Notation

Although it's common to use underscore prefixing to emphasize private use, conventional Swift does not adopt Hungarian notation[5] for naming symbols. In Hungarian notation, a prefix encodes a variable's data type using cues like "sz" for zero-terminated strings, "k" for constants, and "l" for long integers.

Using specialized type-specific notations complicates variable names and make it harder to read Swift code. Swift prizes clarity and simplicity. Type prefixing is unnecessary in a type-safe language. The compiler already checks types on your behalf.

Swift purists might argue against underscore use for exactly the same reason that Swift abstains from Hungarian notation: it complicates symbol names and reduces readability. Unlike C and Objective-C, Swift has a well-defined notion of access modifiers. To some Swift coders, an underscore is as superfluous as a "k" prefix and as egregious a style sin. In this, the purists miss an important point.

Underscores play a key role in Swift's access control system. They enable the Swift standard library to offer technically public symbols like the `_NSCFString` type, which aren't meant for third-party consumption. Swift developers may use this type indirectly, but they should never instantiate an underscored type by name or call an underscored method they did not themselves create.

Some inherently private protocol members must be marked `public` to enable public use by other protocol members. While these members shouldn't participate in published APIs, they must do so because of Swift technical limitations. Underscores express that "there's no other access control modifier that bypasses the need for a technically public member." Until Swift identifies and enhances its access control to provide for these "must be public to work" use cases, the underscore convention allows technically public but functionally private members to be clearly marked "hands off."

5. https://en.wikipedia.org/wiki/Hungarian_notation

Avoiding Global Symbols

Swift promotes namespacing over freestanding functions and constants. This convention applies even in module APIs where you naturally get a basic level of namespacing. You want to keep your namespace clean and focused, scoping items to types and protocols of natural interest. This practice makes Swift a bit more hierarchical than you might be used to in other languages.

Steer clear of top-level constants and functions that clutter your global namespace. Habits from other programming languages may lead you to create shared symbols and functions like these:

```
public let π = CGFloat.pi // no
public let τ = π * 2.0 // no

public func halt() { // no
    PlaygroundPage.current.finishExecution()
}
```

Prefer to embed globals into types as static members. This provides clean, Swiftier namespacing, with minimal overhead costs.

```
extension CGFloat { // yes
    /// The pi constant
    public static let π = CGFloat.pi

    /// The tau constant
    public static let (tau, τ) = (2 * pi, 2 * pi)
}

extension PlaygroundPage {
    // This is never going to be beautiful
    public static func halt() {
        current.finishExecution()
    }
}
```

Where possible, prefer extending an Apple-supplied type (like CGFloat or PlaygroundPage) over creating new types (like MathConstants or PlaygroundRuntimeSupport). Don’t force constants and functions into classes where they have no natural fit. If you must create a new type to warehouse a namespace, prefer a no-case enumeration, which is guaranteed to be unconstructable.

- Namespace constants into a type where they naturally fit.
- Namespace functions into the types and protocols they service, either as static methods or by removing the first argument and reimplementing the function as an instance method.

- Prefer to nest types rather than add items to the global namespace, especially items that are of interest only within a parent type. This allows the parent and its nested types to reduce complexity when referring to each other, so Shape.Triangle and Shape.Circle see each other as Triangle and Circle.

- Embed functions that will be called by only a single method client into that method, rather than create a second method that exists to serve only one parent.

- Move operator implementations into the types they service, so you reduce the number of operators implemented in the global namespace.

Converting Functions to Type Extensions

When converting a global function to a namespaced member, consider its arguments. The first argument's type is the most likely candidate to host the function. For example, you could easily place the following convertDegreesToRadians function into a CGFloat extension:

```
// Better as a CGFloat method
public func convertDegreesToRadians(_ angle: CGFloat) -> CGFloat {
    return angle * CGFloat(Double.pi) / 180.0
}
```

It's easy to embed functions as static members. However, type-specific functionality may better fit as instance methods. You can often drop a global function's first argument and convert the function to an instance member rather than embed it as a static member:

```
extension CGFloat {
    public static func convertToRadians( // maybe
        degrees angle: CGFloat) -> CGFloat
    {
        return degrees * CGFloat(Double.pi) / 180.0
    }

    public func asRadians() -> CGFloat { // yes
        return self * CGFloat(Double.pi) / 180.0
    }

    public var radianForm: CGFloat { // no
        return self * CGFloat(Double.pi) / 180.0
    }
}
```

If conversion to an instance member removes all arguments, consider whether the function is better represented as a method or property. Methods apply actions. Properties express intrinsic characteristics of the owning instance. Just because a method has no arguments does not mean it should always promote to a property. In the preceding example, floating-point values do not intrinsically have a radian form the way a semantically specific Angle struct would. Prefer the function in this case.

- Where reasonable, move freestanding functions into types.
- The first argument's type is usually its natural home.
- Dropping first arguments enables you to convert static functions to instance methods or properties.
- Properties express instance characteristics.
- Avoid implementing any code that produces side effects as a property, even when those side effects are as simple as printing out a value.

Convenience Types

When creating new types to host global constants and functions, prefer caseless enumerations over structs. Unlike caseless enumerations, structs can be instantiated by an API client. An instance of a convenience type like struct Math is meaningless:

```
struct Math { // no
    public static let hhgttg: Int = 42
}

enum Math { // yes
    public static let hhgttg: Int = 42
}

// For example:
let structVersion = Math() // what would this mean?
let enumVersion = Math() // compiler error
```

Since you'd never want an instance of Math to be created, a caseless enumeration ensures this cannot happen. Prefer caseless enums to workarounds that avoid initialization. This example introduces an unavailable initializer, making the structure unconstructable. This produces the same outcome, but it's ugly and offers no meaningful gains over caseless enumerations:

```
struct Math {
    public static let hhgttg: Int = 42
    @available(*, unavailable) init() {} // no
}
```

Nesting Functions

Swift allows you to embed functions within each other. This encapsulates the embedded function and allows you to refactor otherwise repeating or abstractable code to a single call point. The Swift compiler is smart enough to inline and optimize these calls as needed.

Here's an example of a nested function from the Swift Foundation:

```
open class JSONSerialization : NSObject {
  /// Determines whether the given object can be
  /// converted to JSON.
  open class func isValidJSONObject(_ obj: Any) -> Bool {
    func isValidJSONObjectInternal(_ obj: Any?) -> Bool {
      // Emulate the SE-0140 behavior bridging behavior for nils
      guard let obj = obj else {
                return true
      }
      ...
```

When deciding whether to nest functions, ask yourself these questions:

- Will or could this function ever be called from or required by another function other than its parent? If so, it's a poor candidate for nesting.
- Does embedding the function clean up the grandparent's API in a meaningful way by hiding private implementation details? Think nesting.
- Does your code incorporate overly complex clauses? Will moving some of that code into nested functions clear up your intent and make your code more readable? If so, move the code. Nested functions support staging by abstracting away implementation details at each step of a process.
- Are you overnesting? Avoid cluttering the parent, which reduces clarity and maintainability. When nested items provide variations on the same fundamental functionality, merge them into a more general form. If the functionality is core to the type, consider adding an internal type member.

- Nested calls reduce repetition. They establish a single element to debug and perfect, rather than duplications within a function. This reduces errors, especially when updating code, since you need only apply those updates at a single point.
- Parameterize functional variations. For example, don't embed similar implementations directly into different cases of a `switch` statement. Pull out that code to a reusable function.

- Emphasize code logic over implementation particulars. Nested calls enable you to move code into its own chunk, for better logical flow. In doing so, you simplify readability.
- Nested functions are first-class constructs. Swift enables you to treat them like any other instances, like strings or dates or views. You can invoke inner functions from their parent's implementation, pass them as arguments to other methods or functions, or even return them from the declaring scope. This approach enables you to create little function factories that expose and customize embedded items.

Nesting Types

Type nesting establishes subordinate types within a parent to support a parent's functionality. Nesting is commonly used to embed type-specific `Errors`, to establish client types such as `Card` rank and suit enumerations, and to create namespaced access to common enumeration values (for example, `Action.Status.complete`).

Choose to nest types in these situations:

- When working with closely related types with an obvious parent/child relationship and a natural naming scheme. Adding `AnnotatedGraph.Annotation` to `AnnotatedGraph` creates a more coherent API than using a stand-alone second `AnnotatedGraphAnnotation` type.
- When supporting access control. Prioritize isolation over reuse. Establishing a private nested subtype ensures it cannot be expected to be used elsewhere.
- When simplifying members without adding extra types to the global scope. Nesting allows you to combine member properties into coherent subtypes. You might combine `x`, `y`, and `z` members into their own `Position` struct—for example, `Shape3D.Position`.
- When adding a custom flag-argument type meant for use in a member's arguments. Enumerations and option sets allow you to create meaningful call sites. For example, you might nest `AudioSample.EncodingOptions`.
- When providing `Error` types whose semantics are specific to the context of the parent type, such as `Storage.RetrievalError.documentNotFound(String)` or `Storage.RetrievalError.netAccessUnavailable`.
- When creating namespaced access to structured enumerations, constants, and utility methods rather than cluttering a type at its top level—for example, `Formatter.Style.long`, `Math.Utility.cgPi`, and `Math.Utility.toRadians(degrees:)`.

Nested types can also control and customize type properties. Developer Cezary Wojcik shared the following example with me. He uses nested types to create private collections of weak references. This code builds weak wrappers for the members of an observer array, allowing individual items to deallocate as needed. Using private nesting offers this custom weak-collection functionality without burdening the namespace:

```
class NotifierClass {
    private struct WeakObserver {
        weak var observer: Observer?
    }

    private var observers: [WeakObserver] = []
    ...
}
```

Swift enables you to nest types within functions. As this example demonstrates, the syntax for nesting anything within anything else is consistent across the language:

```
struct Foo {
    func f() {
        class Bar {}
        let bar = Bar()
        print(bar)
    }
}
Foo().f() // prints Foo.(f () -> ()).(Bar #1)
```

Use this approach to nest minimal types. For example, you might create a stub delegate class for unit testing.

Designing Singletons

A singleton restricts type creation to a single instance. It limits access to shared resources and provides coordinated actions across its scope. Apple uses this pattern for a few limited types including UIApplication, NSProcessInfo, and NSWorkspace. Few design patterns are as hotly debated as singletons. Answer the question of "Is it good or is it evil?" for yourself. This book leans agnostic verging on pro.

Singletons are safer and more approachable in Swift than in many other languages, as the following example demonstrates. It's easy to build a restricted controller:

```
/// A singleton pattern struct with a single
/// monotonically increasing state value
public final class Singleton {
    // `shared`
    public static let shared = Singleton()

    // Publicly consumable API
    public private(set) var value: Int = 0
    public func next() -> Int { value += 1; return value }

    // Non-constructable publicly because
    // the initializer is inaccessible
    // due to its `private` protection level
    private init() { }
}
```

This design creates a controlled access point with a shared static member. Using a reference type enables you to store and modify state without having to use mutating functions.

Name your singleton access member well. Provide a shared public API and not sharedInstance. Avoid default. The phrase default suggests that other instances are constructable, offering a launching point for a preconfigured default instance. Using shared communicates the core singleton concept: only a single instance can be accessed.

Some Apple-supplied types offer shared members that would be better named default. For example, URLSession offers a no-configuration shared session, which Apple refers to as a “singleton” but it’s not. The type also allows callers to construct default, ephemeral, and background sessions. The name default reinforces the notion that more than one instance can be created, as with FileManager, TimeZone, NotificationCenter, URLSessionConfiguration, and so forth.

In many cases, singletons control restricted resources, meaning safe access is a must. The following code augments the previous example and introduces atomic memory access. Prefer a safer approach for singletons that incorporate non-thread-safe backing data or those that must block access when busy:

```
public final class Singleton {
    public static let shared = Singleton()
    private init() { }

    // Ensure atomic read of _value
    public var value: Int {
        return accessQueue.sync {
            return _value
        }
    }

    // Ensure atomic change to _value
    public func next() -> Int {
        return _accessQueue.sync(flags: [.barrier]) {
            _value += 1
            return _value
        }
    }

    private let _accessQueue = DispatchQueue(
        label: "org.sadun.singletonIsolation",
        attributes: [.concurrent])
    private var _value: Int = 0
}
```

Two callers attempting to modify the same instance may corrupt a shared memory block. You see this, especially, when using threaded arrays or dictionaries. Prevent this by using synchronous access queues to isolate access. The .barrier attribute in this example's next() method prevents further data reads until any ongoing write operation has completed.

- Limit singletons to resources that should only ever have a single instance.
- Mark your initializer as private. This ensures the type can only be created through its shared instance.
- Using private(set) prevents consumers from modifying otherwise visible properties.
- A singleton's visibility should be well chosen: public, internal, or even fileprivate when appropriate. This ensures it is fully reachable within the scope of its use and no further. Your personal fall into singleton dens of iniquity need never be published beyond the scope of your module, upholding your reputation in the eyes of others.
- A singleton should always be final, ensuring it is never subclassed, even within the same module.
- Use access control to ensure your singleton exposes only those API details you mean to make public. Use private and internal generously. It's always easier to disclose a previously internal implementation detail than remove one from a published API.
- Ensure your singleton is thread-safe by introducing dispatch queues or similar atomic access solutions and abstracting the actual storage away from the published API.
- You can design singleton structs, but dealing with stored state becomes trickier. You can use an embedded reference type to store state and avoid mutability issues, but it's not really worth the effort.
- The most common guidance on singletons, which I do not personally endorse, is "Don't use them." I disagree. They play an arguably valuable if limited role in development. Adopt them cautiously and rarely, and be prepared to defend your choice with strong arguments and generous bribery. Protocol-based dependency injection offers a good alternative. When passed an instance, you effectively create

a singleton that is handed from one object to the next. Your code remains testable, allowing you to inject mocks and fakes when running tests.

Adding Custom Operators

Operators are precious things. They are symbols that behave like functions but adopt more natural syntax—for example, 1 + 2 vs. addPair(1, 2). Operators allow you to check, change, and combine values using a mathematical notation:

```
1 + 2 // operator version
addPair(1, 2) // function version
```

The operator version uses an infix plus sign between its two numbers. It's simple and easy to read. You write the code, just as you'd write an equation. Below it is the functional version, which uses a custom name, and it lists its arguments within parentheses. This form is wordier. It feels more constructed than the operator version. The operator version represents the way you were taught to write this by hand.

Unlike many programming languages, Swift enables you to create custom operators and incorporate them into your APIs. Approach operator declaration carefully, keeping the operator's scope and benefits in mind. Choosing when and where to establish operators is a delicate matter. They're a feature best used sparingly.

I still experience occasional APL[6] flashback nightmares. The language wasn't just a horrible experience in tracking down the right characters to create code. Once written, it was unintelligible even to those who had created the routines. Having experienced write-only code in the classroom, I'm not eager to return there in Swift.

Weighing Operator Recognition and Recall

Operators—especially custom operators—lack natural context and cues that allow you to relate functionality to things that you're seeing in code. They suffer from low discoverability and difficult readability. They are, by nature, symbols and not names.

Operators place a mental cost on users with respect to both recall ("What is the operator that applies the behavior I need?") and recognition ("What does

6. https://en.wikipedia.org/wiki/APL_(programming_language)

the operator in this code do?”). While almost every nontrivial program defines many new identifiers, most projects do not define new operators.

As operators become more self-defined or esoteric, that cost rises. Costs are lowest when symbols are tied to conventional standards like ∪ for union and ⊇ for superset. These are standard mathematical representations. They are widely accepted and widely used. You’ll encounter them in many math textbooks. Costs are highest when you introduce synthetic operators like -->. Chances are that you will not recognize this operator, because it’s completely custom.

Even though the union symbol is familiar to many programmers, recognizing a function name like formUnion(with:) can be simpler than remembering what the ∪, ⊇, and ⊆ operators do. This is especially true for custom operators. Without special training, you’ll find safeBitcast(to:) far more readable and comprehensible than, say, -->. The functional form does not require operator-specific recognition, especially recognition for a symbol that was created entirely ad hoc.

Recall is even harder than recognition. Recall relies on retrieving information from your memory by associating that information with related concepts. It’s easier to *recognize* a correct fact than to *recall* the answer to a question. For example, it’s harder to answer “What are the small bits of paper removed during the creation of a hole punch called?” than “Are chads the small bits of paper removed during the creation of a hole punch?” It might take you a moment or a web search to think of that name. Recognition is instantaneous. You either know this or do not.

When coding, it’s difficult enough to recall a standard function. If you’re working in Xcode, context-sensitive completion may help you to expand the concept *subset* to isSubset(of:). That support is unavailable for operators because they lack component words to guide your search. At best, you can visit a search engine and find the “symbol for subset operator.” For custom operators, you’re entirely at a loss.

Challenging Your Operator Recognition Assumptions

Operators that are intuitively obvious to their authors are often inscrutable to a more general audience. To demonstrate this, consider the following list of mystery operators. These operators represent real-world examples I sourced from various contexts. Can you intuit what each operator means?

```
**, <=?, =?, *?, +?, .?, -->, =>, ==>, <-, >>>
```

The first of these should be relatively easy to guess. It's used (and implemented) within this chapter. It's based off a fairly standard workaround for a mathematical application that appears in Fortran and Python. Its more natural (and obvious) symbol is used in many languages to perform another task.

The first half of the remaining items centers on question marks, which might lead you to presume they have something to do with optionals. Some do, but at least one does not. Even knowing that optionals are involved, is there enough of a clue through the symbol to suggest the exact functionality each implements?

The second half is even more inscrutable. The remaining operators all look like arrows of some kind. There are no hints as to what kind of thing is being directed to what other kind of thing or how the arrows establish a functional relationship. Certainly familiarity and context would support understanding.

Even with explanatory examples like the ones that follow, is it reasonable to expect an unprepared reader to determine what the operator does? Especially in a paper-based code review without Xcode documentation lookup support?

```
let y = x ==> NSString.self
let z = x ==> String.self
```

```
var a = "First"
let y = a <- "Second"
```

Recognition is only part of the problem with custom operators. Recall is the other. After being trained on a handful of operators and then letting a few days elapse, could you remember those operators without looking at a cheat sheet? And then would you be able to use those operators properly in code? Now think of a time lapse of a month or a year. At each stage, you may remember that you implemented your functionality somewhere, but being able to find and use that operator grows more difficult.

Unless operator representations have a strong natural association with their functionality (such as set operators and other mathematical forms), the limits of human memory mean that API users will struggle to be aware of the operator's existence, to recall its symbology for use, and to recognize it in context, especially when reviewing code written by another person. In nearly every case, a function or method is more discoverable than a custom operator and easier to read and understand.

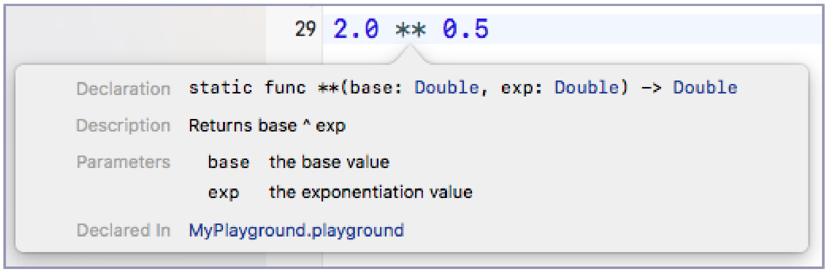
Some might argue that proper documentation will lead you to the correct declaration by incorporating keywords into structured markup. I'd counter that in nearly every case, relying on structured markup to explain, identify, and locate appropriate symbols adds significant overhead to your development costs.

Despite my negativity, a few heavily used, well-considered house operators can be lifesavers. They must be chosen with care and consideration for both value gained and the training costs incurred. Weigh your costs and benefits before adopting new operators into your arsenal, but once they're adopted, promote and use them heavily. A lightly used operator isn't worth the effort. One that is beloved and commonly adopted within the Swift community (for example, !! or ?=, which provide "unwrap or die" and "assign non-nil value" functionality) can be a treasure.

Adding Xcode Support

The human mind's limited ability to carry a load of operator meanings suggests that custom operators should be few, used often (at least passing some threshold of call sites per file, module, or project), and declared close to their use points.

Swift's documentation markup system reduces the burden on recognition in Xcode but not on recall. It's easy to figure out what an operator *does* (assuming you've added structured documentation, as in the following image). It's difficult locating the operator that does what you *intend*.



```
2.0 ** 0.5
```

Declaration	`static func **(base: Double, exp: Double) -> Double`
Description	Returns base ^ exp
Parameters	base the base value
	exp the exponentiation value
Declared In	MyPlayground.playground

Although Swift officially supports a `keyword` documentation comment field, Xcode has not yet been updated to use this feature. At some point in the future, Xcode will hopefully match comment-supplied suggestions to the code-completion engine, allowing you to type phrases like "expon" to search for functions and operator implementations whose names may not include those strings.

Internationalizing Operators

Operators aren't just hard to remember. Outside the core ASCII set, they can be hard to type. Symbols that are easy to type on U.S. keyboards may be difficult to type on international keyboards. For example, the bullet symbol

(Option-8 on U.S. keyboards) is a legal operator symbol. The same symbol on the French keyboard uses Option-@, located where the U.S. keyboard has its tilde (~). While coding, you don't want to continuously cut and paste from declarations or require the system characters picker to complete each line of code. When you must use an operator, prefer common symbols within easy reach on most keyboards.

In theory, you should consider whether operators have regional meanings or do not transport well across international lines. In practice, few symbols will cause problems. To provide a (very) strained example, a U.S. citizen might not recognize this operator to mean "football":

⚽

Admittedly, that's a really bad operator symbol and a really bad operator symbol meaning—plus only one variation of this symbol belongs to the operator character suite.

In most cases, simplicity and ease of lookup should weigh heavily in your decision making. Adhere to the principle of least astonishment. Limit your use of exotic characters (that is, exotic outside your specific development environment) to maximize your expected rewards compared to the costs of typing and discoverability.

Finally, consider sharing costs. Non-ASCII code places a real burden on documentation and websites. It's not that these costs are unreasonable or that some operators aren't natural fits; just consider whether the benefits outweigh those costs when implemented as a whole.

Scoping Operators

It's more common to create utility operators for your own consumption than to export them as public APIs. Ask yourself, "How often can and will this operator be used?" If the answer does not indicate general utility, narrow its access until the operator's benefits outweigh its costs. At that point, count the likely number of times the operator will be called from the code it's scoped to. If the answer is less than a handful (or even a three-fingered Girl Scout salute[7]), replace the operator with a method.

7. https://en.wikipedia.org/wiki/Scout_sign_and_salute

Operator Deglobalization

Swift originally limited operator implementations to global functions. This crowded the global namespace and confused developers who had to, for example, conform to Equatable in a type extension and then implement the == function *outside* that extension. Under the current system, you can move operator methods into the types they support. This is especially helpful when conforming types to protocols that require specific operator implementations.

You may reparent an operator when a type is explicitly mentioned in its implementation. Member operators must include at least one argument of the parent type. You may select either operand as a destination, but you can't move arbitrary operator implementations into types just to get them out of the global namespace.

Prefer to host operator implementations in the type of the left-hand operand. Implement operators as static members of that type. As you see in the following examples, static member implementation simplifies and localizes a conformance to Equatable:

```
extension Foo: Equatable { // yes
    ...type details...
    static func ==(lhs: Foo, rhs: Foo) -> Bool {
        return ...equating logic...
    }
}

// first half of "no"
struct Foo: Equatable {
  ...type details...
}

// second half of "no"
func ==(lhs: Foo, rhs: Foo) -> Bool {
    return ...equating logic...
}
```

Parenting isn't just about protocol conformance. Type-specific operators belong in their natural parent types, too, as you see in this example:

```
/// Establishes a new exponentiation operator
infix operator ** : ExponentiationPrecedence

extension Int {
    /// Builds an integer-specific exponentiation function
    static func **(base: Int, exp: Int) -> Int {
        return repeatElement(base, count: exp).reduce(1, *)
    }
}
```

If a global function is generic but conforms to a protocol, you can move it to a protocol extension, as shown by this example:

```
protocol MyProtocol { ... }

extension MyProtocol { // yes
    static func ***<U>(lhs: Self, rhs: U) -> U {
        ...
    }
}

func ***<T: MyProtocol, U>(lhs: T, rhs: U) -> U { // no
    ...
}
```

If it's fully generic, introduce it at the global scope. There is no natural type or protocol to parent the following operator implementation:

```
func ***<T, U>(lhs: T, rhs: U) -> U { // yes
    ...
}
```

The following operator provides a natural fit for the `Optional` type, but Swift does not offer support for this kind of nuanced association. This operator must be declared globally:

```
/// Converts the preceding expression to an optional value
public postfix func +?<T>(lhs: T) -> T? { return Optional(lhs) }
```

Adopting Common-Sense Operator Practices

When working with operators, keep the following guidelines in mind:

- Operators use a limited character set: /, =, -, +, !, *, %, <, >, &, |, ^, ?, or ~, plus the set of legal Unicode detailed in Apple's documentation.
- Operators can lead with periods (.), but you cannot embed periods into the middle of operators under Swift's current guidelines.
- You can't override operators with reserved uses like =, ->, //, /*, and */.

In addition, adopt the following common-sense practices:

- Overloading the meaning of existing operators is generally more productive than introducing new operators. Always pick consistent meanings. Using > to compare two values is better than using > to pipe the output from one function into another. Using + to add or concatenate is better than using + to apply forced casting. Even "conventional" uses of symbols like division can be confusing. Does (1...20) / 3 divide a range into three sections (1...6, 7...12, 13...18, dropping 19 and 20) or return 1/3, 2/3, ..., 20/3?

- Avoid creating outlandish or contradictory meanings for preexisting operators. `A + B` should not create an A-by-B drawing context, multiply A by B, or perform a debugging dump of A using a print formatter supplied by B. Prefer conventional meanings like "add" and "append."
- Select operators that are reasonably simple to type, whose meaning is easily recognized, and whose use is quickly recalled.
- Limit your use of novel (and especially novelty) operators. If you decide to move ahead, ensure the operator symbology is meaningful to the functionality it implements.
- If you want your Swift code to look like some kind of typeset math paper, you're prioritizing the wrong goals.
- It's common to implement an operator as a thin wrapper around another non-operator function. There may be times you'll want to use the function instead, and you'll have the option to use either call.
- Avoid new operators that establish precedence variations on existing operators, such as a strong coalescing operator. Parentheses are cheap. Operators are expensive.

Speaking of parentheses, include them wherever they make a complex expression easier to read. Parentheses help even when they're not required to determine precedence. Parentheses that enhance the clarity of your intentions are *always* a good thing.

When deciding on precedence,[8] first determine whether your operator follows an existing mathematical pattern (for example, when creating an exponentiation operator). If so, slot your operator in with respect to the existing hierarchy based on normal conventions.

If your operator is not mathematical, weigh how strong the operator needs to be. Consider whether it needs to be left or right associative and whether it performs assignment. For assignment, use `AssignmentPrecedence`. If it does not perform assignment, use your best judgment of strength. The (current) built-in precedence groups are listed in descending order of strength:

```
BitwiseShiftPrecedence
MultiplicationPrecedence (left)
AdditionPrecedence (left)
RangeFormationPrecedence
CastingPrecedence
NilCoalescingPrecedence
```

8. http://ericasadun.com/2016/09/04/optionals-optionals-optionals-introducing-precedence/

```
ComparisonPrecedence
LogicalConjunctionPrecedence (left)
LogicalDisjunctionPrecedence (left)
TernaryPrecedence (right)
AssignmentPrecedence (right, assignment)
FunctionArrowPrecedence (right)
[nothing]
```

Naming Generic Parameters

Name generic type parameters with UpperCamelCase names (like T, U, Element, and Item) to indicate that they are acting as a placeholder for a type and not a value. Prefer words to single letters. Names describe the parameter's role like Key, Value, Pointee, Sequence, and State. Use numbers to distinguish otherwise identical semantics in distinct roles. Sequence1 and Sequence2 refer to two placeholder types, which may not be the same:

```
Dictionary<Key, Value> // yes
Dictionary<K, V> // not as good
Dictionary<T, U> // no

public func +<Pointee>( // yes
    lhs: UnsafeMutablePointer<Pointee>, rhs: Int
) -> UnsafeMutablePointer<Pointee>

public func zip<Sequence1 : Sequence, // yes
  Sequence2 : Sequence>(
  _ sequence1: Sequence1,
  _ sequence2: Sequence2) ->
  Zip2Sequence<Sequence1, Sequence2>

public func sequence<T, State>( // yes
    state: State,
    next: @escaping (inout State) -> T?)
  -> UnfoldSequence<T, State>
```

The Swift standard library reserves single letters, and particularly T and U, to parameters with no associated semantics. Limit one-letter placeholders to situations where all uses of the placeholder are indistinguishable. A single letter means "any type, no specified role":

```
public func !=<T: Equatable>(lhs: T, rhs: T) -> Bool
  where T: RawRepresentable, T.RawValue: Equatable

public func !=<A: Equatable,B: Equatable,C: Equatable>(
  lhs: (A, B, C), rhs: (A, B, C)) -> Bool
```

Originally Swift used shorter type arguments like K for key, V for value, S for sequence, C for collection, and T for type. Some developers have retained the older, single-letter style. This compact style easily distinguishes generic parameters from type names and compresses code. Although adopting single-letter type arguments reduces code width, prefer full words to single letters whenever reasonable.

Even old-style coders use meaningful words for multiple generic parameters with distinct roles. In these cases, the benefits of descriptive names outweigh any issues caused by their overhead. If space is tight, you may want to compromise between `Sequence1` and `S1` with `Seq1` to integrate at least some role information.

- Always prefer words to single letters when generic arguments are constrained with associated types. Otherwise, prepare to struggle with understanding what V.A does not conform to S.T is trying to communicate in your error messages.
- When you do choose to use single-letter placeholders, prefer distinct parameter names like `T` and `U` for completely unrelated types and `T1` and `T2` when you expect there to be two types occupying similar or interchangeable roles.
- Avoid reusing obvious associated type names like `Value` when it's likely that a single type may conform to two protocols. Multiple conformances using the same associated type fields can produce undesired and hard-to-debug behavior.

Naming Symbols

Swift adopts naming style conventions that reduce the effort required to read and understand source code. In adopting these conventions, the language establishes a baseline for standard and expressive code. These styles enable coders to ignore the way people spell, select, and capitalize symbols and focus instead on code intent. In turn, the conventions add meaning and context to a symbol's role and utility, making code more self-documenting.

As an opinionated language, Swift has a common center of gravity with strong design principles. Becoming familiar with the various conventions and stylings can be a significant learning challenge, but once mastered, they reduce language friction. They enable you to incorporate nuance and meaning into your API choices. Apple's official API design guidance can be found at the swift.org website.[9] In the following sections, you'll read through an exploration of these style rules and other community conventions.

9. https://swift.org/documentation/api-design-guidelines/

Casing

Standard Swift uses camelCase: lowerCamelCase for variables, constants, and type members (including enumerations) and UpperCamelCase for types and protocols. Swift does not normally use snake_case (whether common_snake or SCREAMING_SNAKE) or kebab-case (or SCREAMING-KEBAB) symbols in any context, although it permits you to create some of these as legal symbols. Prefer standard Swift casing unless you're working directly in the context of non-Swift libraries.

```
enum Either<L, R> { case Left(L), Right(R) } // no
enum Either<L, R> { case left(L), right(R) } // yes
protocol DataSource {} // yes
protocol dataSource {} // no
```

Using nonstandard casing won't affect compilation, but it is unSwifty, making your code stand out from conventional standards.

Concision

Software engineer Li Haoyi established a set of rules[10] for symbol names. His writing centered on a Scala context, but the lessons are universal. I was pointed to his write-up by Ash Furrow, who in turn adapted these strategies[11] to Swift.

Here are the rules:

- Prefer longer, more explanatory names as access extends beyond types to files, modules, and global scope.
- Leverage context and typing to shorten names.
- Prefer shorter names for symbols you use often and those limited to a narrow scope.
- Mark dangerous methods and functions with long names.

Here is how you apply these rules to your code:

Scoping: as the scope of a symbol grows, longer names add meaningful context. While you can use i and j in a for loop without a significant loss of meaning, these names are poor choices for protocols, types, members, functions, and so forth. A well-chosen, broadly scoped name defines a *role* and allows a code reader to understand how the item works.

10. http://www.lihaoyi.com/post/StrategicScalaStyleConcisenessNames.html
11. https://ashfurrow.com/blog/naming-things-in-swift/

```
{
    ...
    for i in list { ... } // yes
    ...
}

struct I { // no
    var i: T // no
    func i() -> T { ... } // no
    ...
}

struct MemberQueue: ... { // yes
    var latency: T // yes
    func makeIterator() -> T { ... } // yes
}

var count = 0 // no, broadly scoped
var databaseAccessCount = 0 // yes, broadly scoped
```

Names like i and j provide little clue as to how their values will be used with regard to specific semantics. As such, they should be scoped closely to their use point with limited visibility.

Context: context allows you to shorten names by building on the semantics of a parent. The preceding MemberQueue example uses latency, a fairly short name. Its context is established by the type. Together MemberQueue plus latency make sense in a way that latency alone does not.

In a similar vein, you can shorten names with contextual nesting. For example, you might define a FileOperation.Error rather than a stand-alone FileOperationError type. An Error subtype uses a significantly shorter name than a stand-alone type while retaining context-based clarity.

Access: marking an item with a more restricted access level supports shorter names. In Swift, using private or fileprivate access control ensures that the distance between a short name and its definition is small. If you use a shortcut name, a restricted access means it only has to be seen from short distances: from the type declaration or within the same file.

Less-restrictive access weakens context. A code reader has to perform more work to match a symbol to its intended use. Expanding the name with more descriptive words supports understanding as you move away from its point of declaration. A well-expressed symbol carries a level of self-documentation with it.

Naming something this works in a very narrow context. Longer names like comparisonSelector can carry meaning across the more demanding gap introduced by an internal, public, or open scope. Fuller names match a symbol to its intended use.

Frequency: frequency elevates a symbol's status within your project. According to Haoyi, short names follow the principles of Huffman coding.[12] By representing common items with simpler names, you reduce the overhead associated with easily contextualized high-use symbols. Familiarity and recognition reduce symbol bandwidth.

A typical coder will be familiar with simple names like print and open because they're so widely used across many languages. In a similar fashion, a project-native symbol used many places within your code can sustain significant name shortening. For example, you might get away with update or updateDisplay vs. reloadDataRefreshQueueAndUpdateDisplay so long as that function is foundational to your project's design and used at many call sites.

This rule takes advantage of working memory.[13] This is the part of your brain that provides a short-term memory buffer for stored information. Working memory has limited capacity. Favoring a small set of frequently used terms makes sense. It ensures each short-word member can be retained and used without needing the additional cues provided by longer names.

When you'll perform an action only once or twice, prefer a longer and more explanatory name, even when it feels heavy handed. For example, you might create a loadDataAndEstablishQueue method that is called only when the application launches. Extra words explain the expected behavior at the one point (or few points) where that understanding is needed. In doing so, your code redirects the effort required for symbol comprehension from working memory and recall to simple reading and recognition.

Danger: prefer long names for destructive and dangerous functionality. A long name provides a meaningful slowdown when entering and reading code. This extra burden is appropriate for any symbol that should be used with caution. Methods named deleteStoredKeychainCredentials() or resetDataStoreToInitialConfiguration() provide more warning and context than delete() or reset(). As Ash Furrow pointed out in his write-up[14] on this subject, well-written code forces code readers to acknowledge things they *don't know* but *want and need to know*.

12. https://en.wikipedia.org/wiki/Huffman_coding
13. https://en.wikipedia.org/wiki/Working_memory
14. https://ashfurrow.com/blog/naming-things-in-swift/

Categorizations

Avoid incorporating the word Type and other categorizations into your symbol names unless it is unavoidable, such as designing an APIClientProtocol conformed to by concrete APIClient and MockAPIClient types. Prefer names that describe the construct's role rather than its implementation details:

```
enum EitherType<L,R> { case left(L), right(R) } // no
enum Either<L,R> { case left(L), right(R) } // yes
```

Derived types should reflect their ancestry and inherited capabilities. Blend the narrowed role of a derived type (for example, handling logins) with the parent role name (such as view controller):

```
class LoginViewController: UIViewController // yes
class Login: UIViewController // no
```

In the preceding example, there's no context for the Login type. It could be an abstract manager, a data store, or something else. Combining Login with the ViewController role ensures the type name is clear: it is a view controller that handles a login process.

Preserving Concepts

When using a type name to categorize constants and variables, use the type correctly. Avoid dropping key concepts that make items look like they store types that they actually do not. A string is not a URL, and an image is not a view:

```
let url = "http://apple.com" // no
let urlString = "http://apple.com" // yes
let personImage = UIImageView() // no
let personImageView = UIImageView() // yes
```

Acronyms and Initialisms

Development is rife with acronym use. Prefer capitalizing acronyms unless the acronym appears at the start of a lowerCamel symbol:

```
let htmlString = ... // yes
let HTMLString = ... // no

let urlString = ... // yes
let gifImage = ... // yes
struct URLRequest { ... } // yes
class GIFGenerator { ... } // yes

let userid = ... // no
let userId = ... // generally no (except referring to Freudian tripartite)
let userID = ... // generally yes
let dataFromHtml = ... // no
let dataFromHTML = ... // yes
```

Lowercase all letters of a mixed-case brand name or acronym at the start of a lowerCamel symbol and capitalize the first letter for an UpperCamel symbol:

```
let ipadIcon = ... // yes
let latexData = ... // yes
let githubConnection = ... // preferred

protocol IPadRepresentable { ... } // preferred
enum LaTeXStyle { ... } // preferred
class GitHubService { ... } // preferred
```

Labeling Enumeration Associated Values

Prefer labels that modify concepts distinct from an enumeration case name or type:

```
case success(T) // yes
case success(value: T) // yes
case success(successValue: T) // no

case monochrome(Double, Double) // okay
case monochrome(white: Double, alpha: Double) // better
```

Plurality

Plurality plays an interesting role in Swift naming. Most Swift types should be single—but not option sets. Most Swift instances should be single, but many collection instances work better as plurals. This section overviews how plurality affects naming across a number of Swift elements you encounter when designing your APIs.

Protocols

Protocols, from Hashable to SetAlgebra to Sequence, are universally singular. Avoid plurals like Numbers or Incrementables. A protocol describes a single semantic quality. Using plural words incorrectly describes the items that conform to the protocol instead of describing the semantics that conformance guarantees.

Enumerations

An enumeration reads as a singular *thing*: PaperSize, HTTPResponse, DayOfTheWeek, and so on. Don't pluralize enumerations (for example, Colors). Enumeration types should be single (Color) with the possible exception of caseless enumerations used as umbrella types. Each case modifies the core name:

```
// Enumeration type names are singular and UpperCamel
// Enumeration cases are lowerCamel

enum Fingers { case thumb, pointer, middle, ring, pinky } // no
enum Finger { case thumb, pointer, middle, ring, pinky } // yes

enum Planets { case mercury, venus, earth, mars, jupiter } // no
enum Planet { case mercury, venus, earth, mars, jupiter } // yes

enum FailureCondition { // preferred
    static let unreachableCode = ...
    static let unimplementedMethod = ...
}
```

Reserve single-line enumeration declarations, like the ones used in the preceding examples, for the simplest possible groups. Otherwise, assume the case list will grow during development, and prefer multiline declarations. For example, a list of errors might grow as functionality expands. Multiline declarations enable you to comment each case and read the cases as a column of declarations.

Classes and Structures

Both class and structure type names should be single. Instances should also be singular unless there's a compelling reason (such as creating a collection or sequence) that makes instances inherently plural.

```
class View: UIView {} // yes
let view = View() // yes

struct Point { let (x, y): (Double, Double) } // yes
let point = Point(x: 5, y: 3) // yes

let values = [1.2, 1.0, 0.9, 1.3, 1.2] // okay, plural
```

Collections

Prefer singular words (Array, Dictionary, Box, Bag) for collection types and plural words for collection instances. The plural word should reflect the collection's accessible elements, such as an array's or set's items or a dictionary's values:

```
// Collection and sequence instance names represent
// a plural version of the thing they contain or produce.

let colors: Set<UIColor> = [ // yes
    .orange,
    .purple,
    .green,
]

// Prefer `sounds` to `animals`. The former refers to
// values and the latter to keys:

let sounds: [String: String] = [ // yes
```

```
    "dog": "woof",
    "cow": "moo",
    "dogcow": "moof",
]
```

When groupings are not obvious from the type signature, you can extend the point by adding a further clause that brings in the role of the key:

```
let soundsByAnimal: [String: String] = [ // yes
    "dog": "woof",
    "cow": "moo",
    "dogcow": "moof",
]

let usersByIdentifier: [TokenID: User] // yes
```

Pluralize collection labels in methods and functions when the member name does not do the plural lifting:

```
students.append(contentsOf: ["Ella", "William"]) // yes
func cache(images: [Image]) // yes
func cacheImages(named names: [String]) // yes, the method is clearly plural
```

Avoid singular names based on the item *type*:

```
let mixedItems: [AnyObject] = [s, v] // yes
let mixedArray: [AnyObject] = [s, v] // less desirable
let array: [AnyObject] = [s, v] // much less desirable
let numbers: [Int] = [1, 5] // yes
let numberArray: [Int] = [1, 5] // less desirable
```

(The obvious exception to this is pedagogy, where calling something a nameArray describes what you're building. This book skews toward production over teaching and utility coding, so keep that in mind when producing type-driven symbol names.)

When working with attribute-heavy APIs like AV Foundation, prefer plurals (attributes) to singular forms (format). The latter suggests you are passing a structure rather than a collection.

Sequences

Follow collection rules when working with sequences. Prefer singular names for types (Zip2Sequence, UnfoldSequence, WordSequence) and plural names for instances and argument labels.

```
let words = WordSequence(length: .random) // yes
let wordSequence = WordSequence(length: .random) // less desirable
let sequence = WordSequence(length: .random) // even less desirable
```

Option Sets

When working with option sets, prefer plural type names and single component names, even for grouped components. Option sets always represent a plural concept—a collection of bit flags—and their type, and instance names should reflect that plurality. In the following example, express and all combine individual options into ready-to-use presets:

```
struct ShippingOptions: OptionSet { // yes
    let rawValue: Int

    static let nextDay    = ShippingOptions(rawValue: 1 << 0)
    static let secondDay  = ShippingOptions(rawValue: 1 << 1)
    static let priority   = ShippingOptions(rawValue: 1 << 2)
    static let standard   = ShippingOptions(rawValue: 1 << 3)

    static let express: ShippingOptions = [.nextDay, .secondDay]
    static let all: ShippingOptions = [.express, .priority, .standard]
}

let desiredOptions: ShippingOptions = [.express] // use the brackets

struct Directions: OptionSet { // yes
    let rawValue: UInt8
    static let up    = Directions(rawValue: 1 << 0)
    static let down  = Directions(rawValue: 1 << 1)
    static let left  = Directions(rawValue: 1 << 2)
    static let right = Directions(rawValue: 1 << 3)
}
```

Tuples

Tuples are essentially anonymous structures and their instance names should be singular unless there's a compelling reason that unifies the roles of its fields:

```
let color = (r: 0.5, g: 0.2, b: 1.0) // yes

let colors = ( // no
    orange: UIColor.orange,
    purple: UIColor.purple,
    green:  UIColor.green
)

let colors = ( // okay
    baseColor: UIColor.black,
    highlightColor: UIColor.darkGray
)
```

Tuple typealiases should also be singular and capitalized, just as you'd name a structure:

```
typealias Color = (r: Double, g: Double, b: Double)
```

Choosing Label Names

External labels describe the way consumers call your functions, methods, initializers, and (some) subscripts. Each label should be clear. It should express the argument's purpose with respect to the overall call. In contrast, internal parameter names express the way your implementation assigns semantics. These two may not match, and in fact, often *should not* match.

The following example uses the external label from and the internal parameter name image. The external label supports call sites by *introducing* the parameter. The internal argument describes the item's *role* for its implementation:

```
// The vImage_Buffer type is part of the Accelerate network
// and, as such, does not use Swift style naming
extension vImage_Buffer {
    /// Extract bytes from image
    public static func bytes(from image: UIImage ) -> Data? {
    ...
    }
}
```

Selectors that introduce parameters should not name parameter types. Prefer bytes(from:) to bytes(fromImage:). For example, during the Swift 3 migration, Apple updated documentForURL(_:) to document(for:):

```
bytes(fromImage: UIImage) // no
bytes(from: UIImage) // yes

documentForURL(_ url: NSURL) // old
document(for url: URL) // new
```

In the byte-extraction example, the consumer knows it is passing an image, based on the method's type. It's unnecessary to mention that detail in the call. It's vital that the internal code works on an image, so image is the right argument name compared to the from label name. This API choice establishes the correct semantics for both the call site and the implementation.

Generalized Types

When working with type-erased types (Any, AnyHashable, AnyObject) or root types (NSObject), add nouns that describe the argument's role using verbNoun form. For example, in addObserver(_ observer: NSObject), observer supplements add. This rule prioritizes roles over types. It allows you to describe an expected argument without specifically naming an otherwise weakly specified type. Don't add a Hashable or Any component to the name. Use real, meaningful role names. These role names compensate for weak type information.

Objective-C Naming

Swift allows you to follow old-style Objective-C naming style, with dropped first argument labels subsumed into the method name. Avoid this. Prefer animate(withDuration:completion:) to animateWithDuration(_:, completion:), giving co-equal status to both labels. Swift prefers explicit first labels that describe the argument role. Follow the guidance presented in Swift's API Design Guidelines[15] and SE-0005[16] when choosing method names and argument labels.

Indistinguishable Arguments

Omit labels for indistinguishable arguments with identical roles. Labels aren't required for min(number1, number2), zip(sequence1, sequence2), or sum(value1, value2, value3, ...). The arguments are order dependent for zip, but there's no real difference in the way the function treats each sequence.

The base function or method name for these particular calls is unambiguous in its intent. min, zip, and sum are all terms of art. That is, they have a precise and known meaning within the scope of computer science and mathematics. The names are short and clear.

Method and function names are rarely this precise. Names grow larger in real code when there are no well-established terms of art. You could imagine using pickSmallest or selectLowest if min wasn't already well defined.

Indistinguishable Variadics

Use variadics when accepting an arbitrary number of indistinguishable arguments. Because a variadic list accepts zero or more items, you must split the call to ensure a minimum number of arguments—in this case, one. The rest parameter picks up any arguments beyond the minimum:

```
public func pickSmallest<T: Comparable>(_ first: T, _ rest: T...) -> T
public func minValue<T: Comparable>(_ first: T, _ second: T, _ rest: T...) -> T

pickSmallest(1, 2, 3, 4, 5) // okay
pickSmallest(5) // compiles, rest is `[]`
pickSmallest() // won't compile
```

Here, pickSmallest(_:_:) accepts one or more arguments, and minValue(_:_:_:) takes two or more. Both examples skip first argument labels, and their arguments remain indistinguishable from the caller's perspective. The adapted declaration

15. https://swift.org/documentation/api-design-guidelines/
16. https://github.com/apple/swift-evolution/blob/master/proposals/0005-objective-c-name-translation.md

structure guarantees a minimum argument count that must be passed by the caller.

Variadic Placement and Argument Names

Place variadic arguments wherever they are needed. They do not have to be placed last, but you can use only one variadic parameter per member declaration.

```
// This variadic version has three non-variadic arguments so it does not
// conflict with non-variadic two-argument `max(_:, _:)` implementation
public func max<T : Comparable>(_ x: T, _ y: T, _ z: T, _ rest: T...) -> T

public func print(_ items: Any...,
    separator: String = default,
    terminator: String = default)
```

Most variadic arguments *want* to be last, and call sites read better when variadic elements trail. Where possible, let variadic arguments float to the right, all the way up to your defaulted arguments. When variadics do not trail, all subsequent arguments *must* be named.

Prepositions

As a method name grows complex, its core functional description may contain a preposition. Apple's current guidance prefers to move prepositions to the first argument label (removeBoxes(havingLength:) vs. removeBoxesHaving(length:)) except when multiple arguments share an abstraction, as in moveTo(x:, y:). A shared abstraction (x and y) is always stronger than a preposition.

Some prepositions describe the call instead of the target. In such cases, do not move the preposition. For example, prefer sendSubviewToBack(_ view:) to sendSubview(toBack view:). The view and the back share no common semantics. Unlike the boxes that have intrinsic length, splitting on the preposition makes no sense. (I would personally prefer sendToBack(_ subview:) or even subview.sendToBack(), but I don't get a vote on these Apple-sourced examples.)

Move prepositions out from calls rather than using them to balance the call, as in the following example:

```
moveTo(x:, y:) // yes
move(toX:, y:) // unbalanced
move(toX:, toY:) // no
```

Some developers replace a preposition (like "with," "of," "by," "for," or "to") with the method's opening label when the results remain meaningful. Ask yourself if the preposition plays a critical naming role and whether it can be omitted without harm to the name's meaning. Dropping "to" in `moveTo(x:, y:)` completely changes the call's message. `move(x:, y:)` suggests the instance is moving a point, not that an instance is moving *to* a point. In such cases, prefer to retain the preposition and preserve the calling intent.

Avoid moving the preposition to the first argument label when this breaks a shared abstraction. In `move(toX:, y:)`, one related argument has an extra word glommed onto it and the other does not. Prefer `moveTo(x:, y:)`. This preserves both the (x, y) abstraction and the preposition. Neither can be discarded without losing key API information:

- `move(to:,_:)` fails to explain argument roles. This is problematic if you also support polar coordinates.
- `move(x:, y:)` reads like an offset, not a destination.
- `move(destinationX:, destinationY:)` is overly wordy.

Many developers might also prefer passing a point (`move(to point:)`) over separate `x` and `y` coordinates.

Grammatical Phrases

Apple omits first argument labels to avoid splitting grammatical `verbNoun` phrases, as in `addSubview(_ :)`. The guidelines prefer `verbNoun()` to `verb(noun:)`. I'm not completely on board with Apple's advice for all situations. For example, `add(subview view:)` may better represent an intended goal and allow the creation of related members such as `add(shadow:)` and `add(title:)`, but consider Apple's reasoning in how they decided on `addSubview(_:)` over `add(subview view:)`.

Dave Abrahams wrote on the Swift Evolution mailing list[17] that Apple included `subview` as part of its base name for two reasons. First, a base name should express a method's core semantics. As an example, he compared the difference between adding a subview and adding a gesture recognizer or shadow. Although the naming process suggests similarities, the two methods express notably divergent side effects. The first represents a core behavior, while the second extends semantics beyond the view hierarchy.

Second, adopting consistent and simple naming rules is important. According to Abrahams, the Apple team could not converge on a rule that allowed them to break "add" away from "subview" without creating inferior APIs and/or complicating the naming rules in such a way that would damage other APIs.

17. https://lists.swift.org/pipermail/swift-evolution/Week-of-Mon-20161010/028078.html

In this case, addSubview allowed both focused semantics and consistent naming rules.

Apple's Tony Parker explained[18] how important the addSubview method's *operation* was to the type's core purpose. An array stores instances so functions with add and remove base names make more sense in an array context than they do for a view, whose primary purpose is to display content.

Because an array can store other arrays (for example, Array<Array<T>>), the append(contentsOf:) method addresses a special case that disambiguates appending new members and adding an array as a member. Similar core-purpose challenges applied to URL, which fell out as more view-like than array-like. Apple chose appendingPathComponent(_:) over appending(pathComponent:) for this reason.

Other First-Argument Labels

When the preceding guidelines do not apply to your use case, prefer to incorporate first-argument labels. Labels are especially important when their absence conveys incoherent thoughts. For example, words.split(12) communicates less effectively than words.split(maxSplits: 12), as does dismiss(true) vs. dismiss(animated: true).

Subscripts and Label Names

Although most subscripts do not incorporate labels, open yourself to their possibilities. Label names can describe a subscript's role. For collections, lists, and sequences, this role should distinguish a labeled call from standard label-free usage. In this example, a labeled argument allows array indices to wrap, so an index past the end of the array begins again from the start (or end for negative indices). Although these lookups return optionals (because of an empty array check), this approach provides a handy and safe alternative to standard label-less array lookups:

```
extension Array {
    public subscript(wrap index: Int) -> Element? {
        guard !isEmpty else { return nil }
        // Support negative indices via modulo and
        // adding `count`
        return self[(index % count + count) % count]
    }
}

// For example, using a labeled subscript:
```

18. https://lists.swift.org/pipermail/swift-evolution/Week-of-Mon-20161010/027976.html

```
let x = myArray[wrap: 10]
```

One can argue that the preceding snippet should not return an optional for the same reason that normal arrays do not return optionals. If you'd prefer, omit that layer of safety and avoid nil checks. Remove the isEmpty guard and trap on empty array indexing just as you would in normal arrays.

I've written labeled subscripts to ignore case for dictionary lookups (caseInsensitive:), safely collect values (safe:), and perform other utility subscripting. Adopting subscript labels enlarges the range of possible lookup styles for your house libraries.

Types that fall outside the realm of collections, lists, and sequences may also implement subscripts. While subscripting is more common with types wrapping an underlying collection, any type that supports co-equal members can provide a good candidate. For example, you might extend NSColor to add subscripted access to each color channel.

```
extension NSColor {
    /// RGBA color channels
    public enum RGBAChannel: Int { case red, green, blue, alpha }

    /// Return the value stored in a given channel. Each access
    /// calls `getComponents`. Channel values are not stored
    /// between accesses.
    public subscript(channel: RGBAChannel) -> CGFloat {
        guard self.canVendRGBChannels // defined elsewhere
            else { fatalError(Complaints.nonRGBColor) }
        var rgba: [CGFloat] = [0, 0, 0, 0]
        self.getComponents(&rgba)
        return rgba[channel.rawValue]
    }
}
```

Avoid subscript syntax for non-lookup behavior. The following example creates a struct that generates a repeated string using subscripting. This is a bad use of subscripts. Subscripts should access member elements or member components, not replace general method calls with a "cool" syntax:

```
struct StringMultiplier {
    var string: String
    subscript(times count: Int) -> String { // no
        return String(repeating: string, count: count)
    }
}

let instance = StringMultiplier(string: "xo")
instance[times: 5] // "xoxoxoxoxo"
```

- Labeled subscripts highlight special indexing behaviors.
- Subscripts are lookups. They should access member elements or components inherent to the type.
- Avoid labels when implementing conventional subscripting lookups. Prefer customCollection["key"] to customCollection[key: "key"].
- Ensure that labeled subscripts don't duplicate or replace normal and customary behavior.

Initializers

Initializer labels allow you to announce how the information you pass will be treated and the role each value will play. Initializers aren't sentences: avoid making initializers sentence-like. Don't try to form grammatical phrases, regardless of the initializer's style (standard, required, or convenience). Initializers focus on types and arguments:

```
// yes, `Type(thing1:, thing2:)`
init(thing1: T, thing2: U)

// no, `Type(with:, thing2:)`
init(with thing1: T, thing2: U)
```

Omit first-argument labels for value-preserving, type-converting initializers. When the initialization is lossless and type-converting, skip the first label Int64(myUInt32Value).

Include first-argument labels for type-narrowing converting initializers. When the initialization narrows the type, use a label that alerts the user to possible dangers: Int32(truncating: myUInt64value).

Include exact member names for member-wise initialization. When there's a one-to-one relationship between parameters and members, even when the initializer does not include an exhaustive set of members, use member names for labels: init(member1:, member2:, member4:, ...).

Describe roles in convenience initializers. In a convenience initializer, each parameter should describe the *role* of the value being passed to it, offering a suggestion of how that value will be used to initialize the instance. This example initializes a Bezier curve from a character and a font:

```
// Bezier convenience initializer
public convenience init? (
    character: Character,
    font: Font = Font.boldSystemFont(ofSize: 32.0)) {
```

```
    ...
}
```

Avoid inconsistent role labels. Avoid unbalanced labels that explain roles for more than one argument:

```
init(translationX: CGFloat, y: CGFloat) // no, bad Apple!
init(scaleX: CGFloat, y: CGFloat) // no
```

Improve initializer APIs by passing more appropriate types or breaking down arguments to named components:

```
init(translate: CGVector) // yes
init(translate: CGSize)   // less desirable, width and height offsets
init(translate: CGPoint)  // no, x and y are locations not offsets

init(translate: (CGFloat, CGFloat)) // yes, values
init(translate: (tx: CGFloat, ty: CGFloat)) // yes, affine term of art
init(translate: (dx: CGFloat, dy: CGFloat)) // yes, geometry
```

Think carefully about similar types with distinct roles. The canonically correct offset type for any initializer is a vector. Until CGVector was introduced, CGSize offered an acceptable alternative. Width and height can be read as offset extents. Although CGPoint stores two CGFloats, it represents a location, not an offset, and should not be used to initialize a translation. If you must use a point, create an initializer withRespectTo the point rather than using the point location values as absolute offsets.

You can break down compound types into individual arguments. Spread role descriptions between arguments, as demonstrated in the following examples:

```
init(tx: CGFloat, ty: CGFloat) // yes
init(dx: CGFloat, dx: CGFloat) // yes
init(xTranslation: CGFloat, yTranslation: CGFloat) // yes
init(xScale: CGFloat, yScale: CGFloat) // yes
init(scaleX: CGFloat, scaleY: CGFloat) // yes
```

In the preceding example, tx and ty are not camelCased. They are terms of art for affine matrices; dx and dy are terms of art for vectors. A term of art, or a word with a precise and specialized meaning, always trumps normal casing rules.

Convenience Initializers

Prefer convenience initializers whenever there's a simpler, easier, or more convenient way to create new instances than laying out member-wise components. Convenience initializers create type shortcuts that illuminate your coding intent and enhance the readability of instance creation.

If your convenience initializer hides details, suggests incorrect intent, or otherwise makes your code more confusing, you're doing it wrong. Good convenience initializers elevate readability by preferring a semantics-driven API over implementation-driven details.

For example, consider a label view type. Its default initializer may rely on laying out a view frame. This is logical from a windowing system's priorities and meaningless for how the view will be used. From a coder's point of view, an instance should be created from an attributed string. The string can drive layout from its fonts and paragraph styles, all of which can be used to infer a default geometry. This is a perfect opportunity to create a convenience entry point.

Prefer convenience initializers that change the abstraction used to establish an instance rather than ones that just incorporate presets. When the presets are many and varied, establish an enumeration that covers common presets and build a convenience initializer to process those cases. When presets are uniform, prefer instead to default arguments in the primary initializer.

- Convenience initializers prioritize semantics over implementation.
- Create a single primary initializer instead of writing many similar initializers to reflect what are essentially defaulted values.
- A convenience initializer should not take the place of static factory methods. When a convenience initializer's only role is to grind out a single preset, consider moving that initializer to a static method.

Factory Methods

A 2D affine matrix consists of six fields: `a`, `b`, `c`, `d`, `tx`, and `ty`. This one structure supports three distinct modes in addition to the six-field member-wise initializer: scale, translation, and rotation. It's not unreasonable to create factory methods to build these three styles instead of building convenience initializers.

In this use case, the initializer modes don't just change the calling abstraction. They essentially create three different abstractions. The semantic distance between "a rotation matrix" and "a scaling matrix" establishes a kind of pseudo-inheritance from the core type to the intended use.

Pulling each use into a static factory allows you to rearchitect the APIs to decouple the abstraction from the arguments. This allows the labels to better describe their actual arguments.

Consider the following example:

```
// Standard translation initializer
init(translationX: CGFloat, y: CGFloat)

// Redesigned initializer labels, allowing equal weights
init(tx: CGFloat, ty: CGFloat)
init(xTranslation: CGFloat, yTranslation: CGFloat)

// Separating abstraction from labels with a tuple
init(translation: (x: CGFloat, y: CGFloat))

// Introducing a static factory member
static func translation(x: CGFloat, y: CGFloat) -> CGAffineTransform
```

This example starts with the standard unbalanced Core Graphics initializer. It is followed by a pair of initializers using more balanced labels. In each case, the initializer labels must do double duty. They specify the desired outcome (translation, scale, or rotation) as well as the arguments supplied for that outcome.

Adding a tuple argument allows you to pull `translation` out to a single label. Decoupling enables you to split the requested transform style from its arguments. The result is a trade-off. This change balances a slightly awkward call site (`CGAffineTransform(translation: (x: 2, y: 5))`) with better argument labels (`x` and `y`) and a dedicated style label (`translation:`).

Using a static factory member moves this deconstruction one step further, as you see in the following examples:

```
CGAffineTransform.translation(x: 2, y: 5)
CGAffineTransform.rotation(degrees: 180)
CGAffineTransform.rotation(radians: .pi)
CGAffineTransform.scale(x: 2, y: 2)
```

A factory establishes an easy-to-read, easy-to-call initializer that emphasizes each intended use (`translation`) and provides simple argument labels (`x` and `y`) without needing workarounds like a tuple. The resulting calls are clear. They are free to use those simple labels because they don't have to cram all their information into initializer structures.

Naming Methods and Functions

Now that you've read about naming symbols, labels, and initializers, consider the nuances intrinsic to naming methods and functions themselves. Swift adopts a holistic approach to method names that weighs the readability of the entire call site in addition to the design of the base name and its parameters.

Avoid ambiguity. Include all words required to avoid confusion. Prefer remove(at index:) to remove(_ index:), since the call site might appear to request the removal of the index itself, not the element *at* the index. The base name of a function or method should describe its core purpose.

Be succinct. When the caller already knows specific information about the call, such as the parameter's type, remove needless words. Prefer bytes(from image:) to bytes(fromImage image:). As Apple writes, "Omit words that merely repeat type information."

Be grammatical. Prefer subviews(withColor:) to subviews(color:), and capitalizingNouns() to nounsCapitalized(). A well-named Swift call can be read out loud and sound like a fluent part of a conversation. The dot operator that appears between an instance or type and a method normally acts as a comma or conjunction. thisSentence.capitalizingNouns() can be read as "This sentence by capitalizing nouns." There's no fluent equivalent for thisSentence.nounsCapitalized. The closest, "This sentence where nouns are capitalized," requires you to insert an extra word that does not appear in the API.

Remove types from the start of member names when the type refers to the receiver. Prefer withAlphaComponent() to colorWithAlphaComponent() because the receiver is a color. When you're using an unmatched-type receiver, retain the type name. For example, grayDoubleValue.colorWithAlphaComponent() explains that you're building a translucent color from a Double.

Avoid using nounBy construction. Prefer multiplyingMask() to productByMultiplyingMask().

Clarify factory methods and Boolean properties. Add make to factory methods (makeIterator()) and is or has to Boolean properties (isAlive, hasTitle).

Avoid abbreviations. Prefer establishErrorCondition() to establishErrCondition() and fadeBackground() to fadeBG(). Apple offers a list of acceptable abbreviations,[19] which I thoroughly encourage you to avoid in Swift. Limit your abbreviations to conventionally universal expressions like max and min.

Avoid ambiguity. Consider whether a function or method name has multiple interpretations. For example, in displayName, is the word *display* a noun or a verb? Does this refer to a command "display this name" or a quality "the name that is displayed"? When it's unclear, rework the name to eliminate that confusion.

19. https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/APIAbbreviations.html#//apple_ref/doc/uid/20001285-BCIHCGAE

Be consistent. Use the same terms to describe concepts. For example, avoid using fetchBezierElements() for one method and listPathComponents() for another.

Avoid and. Don't call or initialize with and—for example, (view:, andPosition:). Allow exceptions to this rule when a method describes two distinct actions, such as openFile(withApplication:, andDeactivate:), and you want to emphasize that differentiation to callers.

Distinguish purely functional methods from ones with side effects. Use noun phrases to describe methods without side effects, such as distance(to:), successor(). Use imperative verb phrases for methods with side effects, such as print(_:), sort(), append(_:).

Tips for Naming

The more you consider and refine your code, the better it communicates about the functionality and behavior you're building instead of the person who was tasked with preparing it.

When choosing names for public APIs, follow these rules:

- *Prefer clarity to concision.* Don't eliminate words for the sake of short method names.
- *Avoid obscure terms.* Prefer simple, common words (both in your spoken language and in your field) to complex ones. Use terms of art with precise, specialized meanings sparingly to capture exact crucial meaning.
- *Avoid jargon.* This is especially true if your code may be consumed outside the context of your immediate working environment. When in doubt, prefer simpler and more common words to project-specific names and acronyms. As Apple puts it, "Don't surprise an expert. Don't confuse a beginner."
- *Adopt American English spelling.* Prefer initialize to initialise and color to colour because these words are Apple supplied. Feel free to accessoriseAgeingDataCentreStores or honourColourFidelity when you're not providing public APIs intended for general consumption.
- *Spell-check.* It may sound odd to apply fussy preparation checks to source code and its documentation, but your code communicates better when it avoids colections and cloors. Flag spelling mistakes and typos in your code review feedback.

Mutating Variations

Pair mutating and nonmutating method variations. A mutating method alters an instance in place; a nonmutating variation returns a copy with changes applied to it.

Mutating, verb-style name

Use imperative form.

`sort()`, `append(_:)`

Nonmutating, verb-style name

Apply the `ed` or `ing` suffix. Prefer the past participle[20] (-ed). If that's not possible, fall back to the present participle (-ing).

`sorted()`, `appending()`

Mutating, noun-style name

Prefix the name with `form`.

`formUnion()`, `formSuccessor()`

Nonmutating, noun-style name

Use the noun name.

`union()`, `successor()`

At some point, Xcode will catch up with SE-0047[21] to support two new document comment fields: `MutatingCounterpart` and `NonmutatingCounterpart`, which will cross-reference related members through QuickHelp. This feature has not yet been implemented in Xcode.

Computed Properties vs. Methods

Properties and methods play distinct roles in Swift. A property expresses an inherent *quality* of an instance, while a method performs an *action*. If you're unsure whether to design a property or method, consider the following guidelines:

- Methods can have parameters; properties can't.
- Prefer methods for any call with side effects. If a method does something (for example, it loads, parses, toggles, or prints) or has a verb name, it should not be a property.

20. https://en.wikipedia.org/wiki/Participle
21. https://github.com/apple/swift-evolution/blob/fa75ca35911a8d20f5f38d0f19e3ab38457b4ab9/proposals/0047-nonvoid-warn.md

- Prefer properties for simple values that you can get and/or set.
- Properties should express a semantic intrinsic quality of a type instance.
- Properties allow you to add observers via willSet and didSet.
- Unlike stored instance properties, stored type properties must always be given a default value.

Adding Defaults

Swift philosophy focuses on concision and clarity. Defaulted parameters can simplify common use cases. Any parameter associated with a single commonly used value represents a good candidate for a default value.

Defaulted APIs create clean, focused call sites. Aim to default any argument that's typical, obvious, or potentially irrelevant in a majority of use cases. In the following example, a convenience initializer establishes a new view with everyday attributes. These attributes enable users to avoid blocks of uninteresting and repetitive code, packing common setup tasks into a single simple call:

```
extension UIView {
    /// Builds a simple view, prepared for Auto Layout
    /// and ready for playground use.
    public convenience init(
        _ w: CGFloat,
        _ h: CGFloat,
        position: CGPoint = .zero,
        backgroundColor: UIColor = .white,
        translucency alpha: CGFloat = 1.0,
        borderWidth: CGFloat = 0.0,
        borderColor: UIColor = .black,
        cornerRadius: CGFloat = 0.0,
        postSetup: (UIView) -> Void = {}
        ){
        ...
    }
}
```

```
UIView(300, 200, backgroundColor: .blue) // for example
```

Forcing users to supply typical arguments complicates call sites. Defaults shrink calls, allowing callers to discard any conventional component that does not require specialization:

```
let order = lastName.compare( // no
  otherName, options: [], range: nil, locale: nil)
```

```
let order = lastName.compare(otherName) // yes
```

Provide default arguments for the following:

- Conventional settings such as delay: 0.0, fps: 30, repeats: false, and the like
- Nullable items, which can default to nil
- Trailing closures such as completion handlers, defaulting to {}
- Option sets that may be empty (typically options, exclusions, features, and so on), defaulting to []
- Dictionaries (typically options, attributes, info), defaulting to [:]
- Arrays (typically options, attributes), defaulting to []

Prefer default arguments to establishing method families. Defaults simplify APIs and reduce the API surface area.

Move defaulted parameters toward the end of your function and method declarations. Apple writes, “Parameters without defaults are usually more essential to the semantics of a method, and provide a stable initial pattern of use where methods are invoked”:[22]

```
func transform(around anchor: Anchor = .topleft, // no
    _ t: CGAffineTransform)
func transform(_ t: CGAffineTransform, // yes
    around anchor: Anchor = .topleft)
```

When you incorporate a trailing closure argument, place that parameter *after* your defaults. Swift will not allow a parameter declared before defaulted arguments to act as a trailing closure:

```
public static func this<T>(
    file: String = #file,
    line: Int = #line,
    block: () -> T?) throws -> T // last
{
    guard let result = block()
        else { throw NilError(file: file, line: line) }
    return result
}
```

Protocols

Swift protocols aren’t just bags of common syntax. Protocols incorporate well-defined, focused, testable semantics that grow out of the existence of real-world use cases. There must be a sufficient number of potential use cases to

22. https://swift.org/documentation/api-design-guidelines/

make their existence worthwhile. A good protocol should be applicable to many situations, not just one or two.

Consider a new initializer similar to `init(repeating:, count:)` that uses a generator (`() -> Element`) instead of a fixed value (`Element`) for its repeating value. Implementations that fall under such shared semantics may apply to all conforming types, not just the one you're working on at the moment. By placing this initializer into `RangeReplaceableCollection` instead of `Array`, you provide functionality across all range replaceable conforming types.

You describe protocols using nouns (Error, OptionSet, Integer) and adjectives (typically ending in -ble, like Hashable, Strideable, Indexable). These names express what a conforming type *is* and what it *does*. A protocol's name reads as a meaningful role for its conforming types and reflects its underlying semantics.

Look for opportunities to refactor code into protocols. Watch for repeated or similar code across different types and code that is applicable beyond its host type. Redundant code segments with just a few type-specific changes hint at patterns that lend themselves to generic and protocol implementations. Focus on the commonalities of design to find targets of opportunity.

Instead of adding separate conformances for, for example, `Int` and `String`, or `Array` and `String`, check whether there is a unifying semantic concept at a common protocol both types already adopt. The former pair are both `Hashable`, adding unique identity expression. The latter pair are both `Sequence`, capable of producing sequential, iterated access to their elements.

When conforming to protocols and adding default implementations, do so at the highest possible abstraction. At the same time, don't force implementations into protocols where they don't fit and the implementation does not relate to the protocol's unifying concept. Instead, create a new protocol. Conform the types that apply, and maybe even add an extension to implement whatever generic behavior you're looking for.

- Semantics. Not syntax.
- One protocol does not rule them all. Keep protocols short, sweet, and meaningful.
- Avoid overloading protocols. Too many semantics distract from a protocol's one true calling.
- Refactor functions with extensive angle-bracket clauses to use protocol extensions, producing generic beautification. Swift's protocol extensions use `where` clauses to

constrain where methods apply, enabling you to move clauses away from overloaded angle brackets and into a more meaningful context.

- It's never too late to refactor. While it's great to write generics and protocols right out of the box, it's extremely common to develop code and then later consider how to retrofit.
- With -ble (specifically -able and -ible) word endings comes a world of pain. I've written about -ble spelling at my blog[23] in great detail.
- "'Spider-Man, Spider-Man, does whatever a spider can' is a clear expression of protocol-oriented programming."—Nate Cook

Generic Beautification

Generic beautification converts complex generic functions into less-complex protocol extensions. Adopt this approach where you can. In the following example, the removingDuplicates(of:) global function beautifies to a removingDuplicates() protocol method:

```
/// Returns array of sequence elements removing duplicates.
func removingDuplicates<S: Sequence>(of sequence: S)
    -> [S.Element] where S.Element: Hashable {
    var seen: Set<S.Element> = []
    return sequence.filter { seen.insert($0).inserted }
}

extension Sequence where Element: Hashable {
    /// Returns array of sequence elements removing duplicates.
    func removingDuplicates() -> [Element] {
        var seen: Set<Element> = []
        return filter { seen.insert($0).inserted }
    }
}
```

The restrictions on the function are associated with hashable set elements, used to keep track of already seen values. A protocol extension creates a better and simpler implementation. The where clause moves from the function (now method) to the extension declaration, and the first argument is dropped to provide a simpler call on any sequence.

Adding Typealiases

A typealias creates a new name for an existing type, simplifying code. Adopt typealiases into your APIs when the alias establishes a more concrete reusable expression. Like custom operators, a typealias should be used multiple times

23. http://ericasadun.com/2015/08/21/swift-protocol-names-a-vital-lesson-in-able-vs-ible-swiftlang/

throughout your project or have the potential to be used repeatedly by API clients. Justify your introduction of an extra layer of abstraction and an extra symbol.

Typealiases should be shorter and cleaner than the types they replace. Adding a typealias for complex or boilerplate functions is a good thing. Adding a typealias for a common type usually isn't, unless you're trying to differentiate a specific *role* for its use.

```
typealias Completion = (Result) -> Void // yes
typealias VoidClosure = () -> Void // yes
typealias IntegerType = Int // no, just a rename
typealias Magnitude = Double // okay, role
```

Typealiases do not prevent passing an instance of the original type. The compiler will not check, for example, for CelsiusDegrees declarations and differentiate them from Double.

Typealiases mustn't obscure normal use. Avoid generic typealiases that distort or hide an underlying type. The name should be semantically meaningful in the context of the alias:

```
typealias MyType<Value> = Dictionary<String, Value> // no
typealias StringKeyedDictionary<Value> = Dictionary<String, Value> // yes
```

Use typealiases to simplify referenced elements in protocol declarations:

```
public typealias Index = Base.Index // yes
public typealias IndexDistance = Base.IndexDistance // yes
```

These aliases allow you to refer to T.Index and T.IndexDistance rather than T.Base.Index and T.Base.IndexDistance.

Typealiases also allow you to simplify and enhance type-specific uses of complex protocols. For example, the following type allows the protocol to be subsumed into a type-specific use:

```
class MyViewController: UIViewController {
    typealias DataSource = FormPresentingDataSource
        // now MyViewController.DataSource exists
    weak var dataSource: DataSource?
}
```

Choosing Value vs. Reference Types

When crafting APIs, carefully consider whether to present consumers with value types or reference types. Many words have been dedicated to this topic, so I'll keep this short. When designing types, remember these rules:

- If it's basically a bag of data, use a value type.
- If it has a lifecycle, use a reference type.

Don't let your training with object-oriented languages keep you from using and enjoying value types and (especially) adding extensions. Structs and enumerations are first-class types in Swift, along with classes.

A few more thoughts:

- If you'd conform it to `Equatable`, think value type; for example, `Point(2, 3)` and `Point(2, 3)` are the same thing.
- If you'd distinguish it from another version with exactly the same data (for example, `View1` and `View2` sharing the same frame), use a reference type. For example, `View()` and `View()` are two different things.

Here are clues that indicate a reference type is appropriate:

- If you'd implement `deinit` to clean things up before deallocation, think reference type.
- If you'd persist an instance through a coder and rehydrate it at a later point, lean toward reference type.
- If you're working with pointers for trees or linked lists or the like, use reference types.
- If more than one thing will attempt to modify the instance, and the changes must propagate to all interested parties, use reference types.
- If you need variations on a core type through inheritance, use reference types.

For Apple-provided APIs:

- Most Cocoa APIs are reference types. This shouldn't be a surprise, given Objective-C and `NSObject`.
- Most Swift Foundation refactors are value types: `Date`, `URL`, `String`, `Array`, `Dictionary`, and so on. If I were writing a basic color type from scratch, I'd use a struct to store red, green, blue, and alpha. Two red colors (0xFF000000 and 0xFF000000) are the same thing.

All bets are off when it comes to Core Foundation C-based APIs. Although there's a lot of value type in Core Foundation, much of the library is based on reference types, with a few value types like `CFRange` thrown into the mix. Core Foundation implements a limited object model in C, using calls like `CFRetain` and `CFRelease` to handle object memory management.

Writing Good Errors

Errors exist at the intersection of information and flow control. Most early error guides for NSError, the original Cocoa error type, direct you to consider the end user in crafting your content. Today, developers are the primary consumer of errors, and advice has shifted accordingly. The following points discuss how you might best create error content for developer consumption:

- *Be clear.* A good error message should establish what the issue is, the cause of the issue, the source of the issue, and how to resolve the issue. Take inspiration from Foundation and offer both failure reasons and recovery suggestions in your error feedback.

- *Be precise.* The more your error traces back to a specific fail point, the better able an end programmer will be able to use it to fix the code or respond with runtime workarounds.

- *Incorporate details.* Swift errors enable you to create structures, associate values, and provide vital context about where and why things went wrong. Create more informative errors by supporting workarounds and recovery through your Error type design.

- *Explain yourself.* Don't eliminate words for the sake of shorter error messages. "Unable to access uninitialized data store" is preferable to "Uninitialized."

- *Focus.* Limit your explanations to the issue your error is dealing with. Avoid unneeded extras.

- *Add support.* When you incorporate API and documentation references, you further explain the condition and support recovery. Links are good. Snippets can be good. Full documentation is unnecessary. Allow features like QuickHelp to properly fill their role without attempting to usurp them.

Unlike other types and protocols, you may retain the word Error in your error types. Throwing FileError.fileNotFound more sensibly establishes an error context than throwing File.fileNotFound or Reason.fileNotFound. Many Swift developers now prefer Swift native errors to NSError, but there are still many community members who stick with the more traditional Cocoa Touch class to better support cross-language development.

Wrapping Up

You've now learned about ways to declare and refine APIs. APIs may be consumed by your own types or they may play a role in a shared module with nuanced visibility. The quality and expressiveness of your shared functionality and the readability of call sites accessing that functionality play important roles in Swift design.

In reading this chapter, you've explored the notion of Swift design hierarchy and how nested types and functions play a large role in limiting an API's visible surface area. In Swift, a well-considered smaller API surface is preferred to a more expansive one. Access control and careful selection minimize development and maintenance costs to provide a highly curated API experience.

You've discovered why custom operators should be very few and very meaningful, since they place a huge burden on discoverability and recognition. Operator design, too, plays into the overall theme of deglobalization and tightening APIs.

You've read about Swift's strong design principles and how naming plays an important role in creating proper APIs. From label names to call site phrases, Swift incorporates fundamental and opinionated conventions.

Now that you've spent time exploring type design, symbols, protocols, and operators, the next chapter turns its attention to resilience. You'll look to the future to incorporate best practices for documenting and future-proofing your code. Read on to learn more about commenting, structured documentation, well-considered bookmarks, and more.

CHAPTER 5

Look to the Past and the Future

Where have you come from and where are you going? Time plays a critical role in Swift style. Always consider the future. How will you prepare code to ensure a robust, well-documented, and long-lived codebase? Seeing Swift in this larger scope is important in both adopting new language practices and supporting code beyond its moment of creation. This short chapter explores time, guiding you into routines that enhance, document, and support the Swift code you're now writing.

Reconciling Past You vs. Future You

It's a basic truth in programming. Coding is difficult when you get interrupted before coming to a really good stopping point. Life happens and reality is cold and harsh. Sure you intend to comment about what needs fixing and where you left off, but past you always provides less reminding than future you desperately needs. Future you can be really dense.

The worst moments happen when you break off in the middle of refactoring. That grand design in your head, the one where every piece fit together like a stunning mosaic? It's gone. Duct tape, BAND-AIDs, and a bit of WD-40 are all you have to look forward to. Past you and future you are never friends. If you've ever written a comment similar to "The methods in this class probably don't work the way you'd expect based on their name," you know what I'm talking about.

Never underestimate future you's lack of sense. Add code and doc comments to explain your current thoughts, and while you're at it, write tests. Tests can save you the whole "What I was doing here?" because you can just look at what is broken and what you expected to work. Sadly, your code will pass every test you never got around to writing. This is especially true when your code has inherent bugs tests could have caught. It's better to write less code and make up that time by explaining and testing the code you did write.

Sure, it helps to leave in a to-do where it counts ("the performance is really bad here"), but while you're at it, try to leave a few ideas about what exactly is going wrong and what hypotheses past you have rolling around your soon-to-be-extinct neurons. Past you understands things. Future you is clueless.

It always costs less to fix things in the past when you've already invested in uploading the full design into your brain. Re-upping that design and getting back to speed involves huge penalties in efficiency, in understanding, and in coherent and holistic understanding of your goals.

- Future you is thick as a plank.
- Past you can be extremely irritating.
- Comment as if you are explaining key design decisions to a particularly dimwitted stranger, especially when that stranger is you.
- More tests. Less code.
- Fill in those structured document fields. The more you comment now, the less grief future you will experience.
- "Code that makes clear its own thoughtlessness, while not ideal, is still more maintainable than code whose thoughtlessness must be determined through painstaking analysis."—Paul Cantrell

Documenting in Real Time

Prepare for elapsed time by commenting and building documentation as you write your code. No matter how much you have going on in your head right now, you won't be the same you in a year when you look back at this code. You'll be future you, and future you may not be able to read, understand, and process the code you wrote a year ago without a significant investment of time.

The same time-gap loss applies when you intend to write comments just a few days, weeks, or months down the line. If you don't have the details in your head as you do at the time you write your code, you're not producing your best documentation. It's far less expensive to document in real time than it is to return to code and attempt to document without those special insights that design-time markup afford you.

Documenting at the peak of your understanding provides more insightful explanations and better examples. You can retain and document key points about the limits, intent, and complexity of the code you're writing. Documenting and writing tests isn't always fun like coding is. That's one reason so many people put it off. But in putting it off, you're doing yourself and your code a disservice. You have the best understanding about the mechanics of your code when you're creating it.

Developers who design algorithms and types before implementing them, especially those who follow test-driven development,[1] have a distinct advantage. If you're someone who develops through exploration, letting the code grow to meet a goal, it is most valuable to document early and thoroughly.

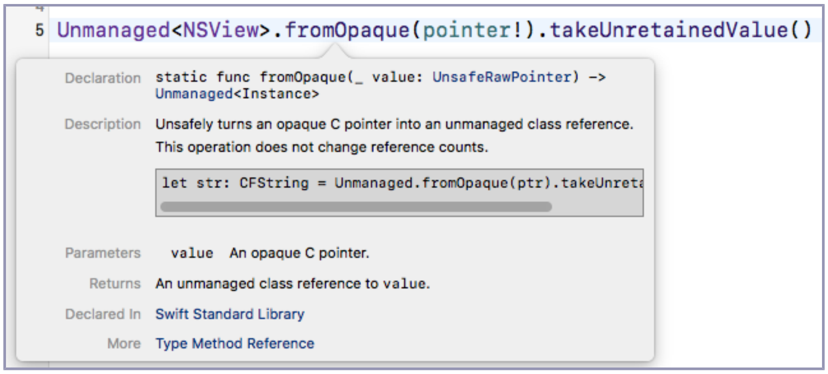
Adding Structured Markup

Swift structured markup uses a standardized annotation template to support automatic documentation generation. Similar to other structured documentation systems like Doxygen, reStructuredText, Javadoc, Pydoc, and so forth, you embed API details into triple-slash (///) or double-asterisk (/** */) comment blocks:

```
/// Unsafely turns an opaque C pointer into an
/// unmanaged class reference.
///
/// This operation does not change reference counts.
///
///     let str: CFString = Unmanaged.fromOpaque(ptr)
///                              .takeUnretainedValue()
///
/// - Parameter value: An opaque C pointer.
/// - Returns: An unmanaged class reference to `value`.
```

Swift markup includes keywords like Parameter, Returns, and Throws, along with support for freeform descriptions and code examples. Adhering to format standards enables automatically parsing for presentation in your IDE. Xcode's QuickHelp system integrates this markup, creating well-formatted reference material for anyone accessing your APIs, as you see in the following image:

1. https://en.wikipedia.org/wiki/Test-driven_development



```
Unmanaged<NSView>.fromOpaque(pointer!).takeUnretainedValue()
```

Declaration `static func fromOpaque(_ value: UnsafeRawPointer) -> Unmanaged<Instance>`

Description Unsafely turns an opaque C pointer into an unmanaged class reference.
This operation does not change reference counts.

```
let str: CFString = Unmanaged.fromOpaque(ptr).takeUnret
```

Parameters `value` An opaque C pointer.

Returns An unmanaged class reference to `value`.

Declared In Swift Standard Library

More Type Method Reference

Swift markup incorporates CommonMark[2] annotation. Use standard Markdown-style features like bolding, italics, and so forth. You can embed code examples with either four-space indentation or triple-backtick code fencing.

Using triple backticks enables you to specify the language in question for smarter code styling on platforms that support language-specific code fencing, like in the following example:

~~~
```swift
// swift-styled code here
```
~~~

New features introduced by the open source Swift Evolution community[3] are still being implemented by Apple's Developer Tools group.

Structured documentation enables you to do the following:

Annotate your code using a standard Swift schema. Describe parameters, return values, possible errors, and the like using well-defined tokens.

Use structural elements tags. Incorporate explanatory, functional, and bibliographic information into your code. Explanatory elements specify what your code does, why you wrote it, who you are, and when you wrote it. Functional elements annotate the details of how your code works and how routines are called. Bibliographic elements offer related definitions, documentation, and sources.

Embed markup hints. Markup adds style, including monospacing, italics, and bolding to your comments. Rich text elements expand the expressive range of comment text. Use markup sparingly, with a focus on communicating information:

2. http://commonmark.org
3. https://swift.org

- Monospace language-specific elements like keywords and method names.
- An italicized word establishes an important technical phrase, signaling that consumers should pay attention to the use. Prefer italics to highlight technical terms ("A *normal* value is a finite number using full precision") rather than to emphasize intent ("The capacity is *underestimated*").
- Use bolding to add prominence to important concepts that emphasize potential dangers ("The `Element` array is suitably aligned **raw memory**").

Support QuickHelp. Prepare documentation that's compatible with Xcode's inline rendering and presentation tools.

Add documentation comments for every method and property. Apple writes in its API guidelines, "Ideally, the API's meaning can be understood from its signature and its one- or two-sentence summary.…Insights gained by writing documentation can have such a profound impact on your API's design that it's a good idea to do it early on."[4] The guidelines conclude with this key bit of advice: "If you are having trouble describing your API's functionality in simple terms, you may have designed the wrong API."

Basic Markup

Here a few tips on adding structured documentation to your code:

Locate. A comment's location establishes context. Markup placed just before a declaration annotates that declaration, even when the comment does not mention a symbol by name.

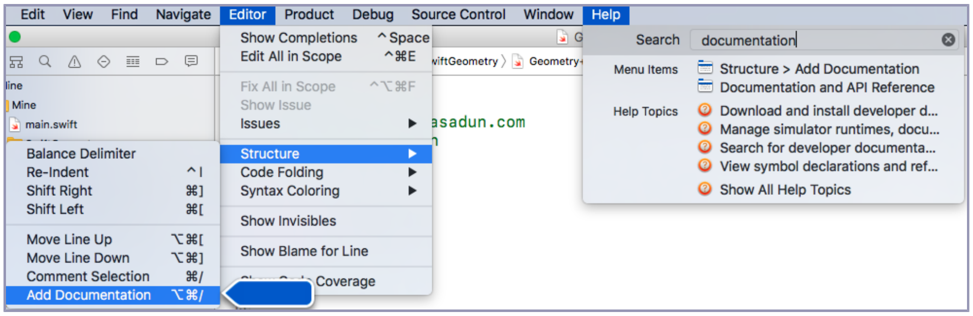
Summarize. Summarize the entity you're declaring. Provide a succinct, clear description of its role. Your summary should be a sentence fragment that omits the leading symbol name. Swift's system automatically picks up and prepends symbol names to descriptions. Use third person: *returns* and not *return* as if the symbol is the actor. For example, "Accesses the subsequence bounded by the given range." Capitalize the first letter of your summary and prefer a terminating period.

Describe. Explain the item's role as fully or succinctly as needed. Document any known issues or limitations associated with the API. Offer examples if the item's use isn't obvious. Add notes, if needed, about the API's attribution (such as its authors and copyright), version availability, admonitions about potential usage pitfalls, its performance characteristics, functional semantics, and related ("see also") information.

4. https://swift.org/documentation/api-design-guidelines/

Annotate. Document the complexity of any computed property or method that is not O(1). Apple writes, "People often assume that property access involves no significant computation, because they have stored properties as a mental model. Be sure to alert them when that assumption may be violated."

Template. Xcode scans symbols and creates a basic markup template when you select Editor > Structure > Add Documentation (Command-Option-/), as shown here.



```
/// Returns a Boolean value that indicates whether
/// the two arguments have unequal values.
///
/// - SeeAlso: `Equatable`, `Comparable`
public func !=(lhs: Int64, rhs: Int64) -> Bool
```

The kind of symbol you're documenting guides its description. Here's a rundown of the most common symbol types you'll encounter when crafting markup.

- *Types.* For classes, enums, and structs, describe the construct's role, such as "A signed integer value type" or "Represents a set of Unicode-compliant characters." Prefer descriptions that start with an article, although it's not uncommon to see a verb.
- *Protocols, associated types, typealiases.* Describe what the symbol is, such as "A type that can be initialized with a string literal" or "The Objective-C bridged type of `Data`." Start the description with an article.
- *Subscripts.* Describe what the subscript indexes, such as "Accesses the subsequence bounded by the given range" or "Sets or returns the byte at the specified index." Start the description with a verb.
- *Functions and methods (including operator implementations).* Describe what a function does and what it returns, such as "Performs a logical NOT operation on a Boolean value" or "Returns the position immediately before the given index." Start the description with a verb.

- *Properties, variables, and constants.* Whether stored or computed, global or local, describe the item's role, such as "The path to the directory containing shared data" or "The end `Index` into the data." Start the description with an article.
- *Initializers.* Describe the instance being created, such as "Creates a new, empty collection" or "Returns a `Date` initialized to the current date and time." Start the description with a verb.

Other comments should actively describe item creation and the outcome being established.

Learning More

Want to learn more about documentation markup? For further coverage of this topic, check out these resources:

- Apple's Markup Formatting Reference.[5] This document offers a general overview of supported features.
- My stand-alone *Swift Documentation Markup* book dives into exhaustive coverage of doc comments. The book is currently available at iTunes[6] and Leanpub.[7]

The open source Jazzy project[8] offers a command-line utility that generates documentation from Swift documentation markup. It's an excellent way to create output that matches the look and feel of official Apple documentation.

Commenting Well

Traditional comments have gone out of fashion in some circles. Comments aren't compiled or executed. They don't naturally change and evolve along with the code they decorate. Because of this, some coders have stepped back from their use or they hesitate to incorporate what might be a frivolous use, believing less is more.

To get a sense of this philosophy, developer Andrew Warner writes in his "Beware the Siren Song of Comments" blog post[9] that comments are naturally subject to decay. Warner believes comments are always a failure on the part

5. https://developer.apple.com/library/content/documentation/Xcode/Reference/xcode_markup_formatting_ref/
6. https://itunes.apple.com/us/book/swift-documentation-markup/id1049010423?mt=11
7. https://leanpub.com/swiftdocumentationmarkup
8. https://github.com/realm/jazzy
9. http://www.strongopinionsweaklytyped.com/blog/2014/08/27/beware-the-siren-song-of-comments/

of the coder, suggesting that comments compensate for a coder's failure to express specific intent through code.

Warner writes that as a noncompiled code component, a comment has no checks and balances placed upon its use. When comments grow out of date, they cannot be automatically flagged. They are not testable and present no user-facing data, limiting their ability to resolve as code evolves.

I don't claim that comments should counterbalance bad design decisions like poor naming or flawed algorithms. I do disagree with an overall "comments are failures" philosophy. Comments play an important role in good coding practices—whether your code is meant strictly for internal use or to be consumed as APIs. Here are my arguments in support of comments:

Comments explain the non-obvious. They describe the tricky, the gotchas, the workarounds, and the counterintuitive to a reader. *This is intentional. This thing affects that thing. This data is not validated at this stage. Responsibility for this process now passes to this delegate.* Comments establish what your assumptions are at a specific point in code, and they express design qualities in words that are separate from the code that uses those assumptions.

Comments create a record of intent. They discuss how design decisions came to be—what paths were explored and why some of those paths weren't chosen in the end. Comments enumerate the approaches that were tried and explain why they were abandoned.

Comments allow you to colocate thoughts with the code they refer to. Commit messages are helpful, but a commit message isn't the proper place to document workarounds or unexpected behavior. Version control history doesn't travel with source code the way comments do when the code is repurposed or reused in another project.

Comments provide a bread crumb trail of design decisions. Design decisions cannot be intuited just from reading code or scanning tests. Comments add an historical record of why choices were made and what motivated them.

Comments support the principle of least astonishment. Commenting on the unexpected is important because the alternative is essentially a bug. Unexpected complexity is one of the things you'll want to comment about. So are edge cases. The consumer of your code does not have the time or resources to re-engineer from scratch. Document those things you know people likely won't discover on their own without a serious commitment of effort.

Comments don't paper over bad design and coding. Comments document the process of making code right. This supports future code reading and updates.

Good comments reduce the mental effort for readers each time they review your source. Comments enable readers to focus more on specific tasks like “how to add this feature” rather than “what the hell was going on here.” While good code *can* be self-documenting, that doesn’t mean it always is (or even usually is) self-documenting.

Comments support formal boilerplate. There’s really no place in code other than comments to add required content like copyright, authorship, and confidentiality statements, all of which represent a coding reality for a substantial portion of the developer community. (Many automatically snip away the Xcode-produced boilerplate and that’s okay too. It’s freeing and makes your files smaller and cleaner.)

As stepping stones to past you, good comments document what you were thinking and why you did things the way you did. Your past design decisions should never be a mystery or a burden placed before future readers of your code no matter how brilliant or insightful you expect those readers to be.

I agree with Marin Benčević, who wrote about abandoning “comments are only for bad code” in “Why code without comments wastes my time.”[10] He asserts that reading code is difficult, no matter *how well written the code itself is*. There’s a high cost to not only understanding code at a line-by-line level but also explaining how code pieces fit together into a greater gestalt. Like any other kind of content creation, introductory and explanatory text in code provides a jump start for readers, lowering wasted time costs and minimizing bugs resulting from poor code comprehension.

Laying Out Comments

Good Swift code combines markup with traditional comments. Reserve structured markup for API documentation, regardless of scale. It can be as valuable to explain a variable’s role as to explain a method’s interface. Use traditional comments to annotate your development. Your code expresses what is being done. Traditional comments explain why it is being done, using terms of implementation detail.

When combining traditional comments with structured markup, place markup before the item being described. This layout is required for proper QuickHelp parsing. When using traditional comments on their own, place them above or to the right as you normally would without consideration for markup:

```
/// Returns the lesser of two comparable values.
```

10. https://medium.cobeisfresh.com/why-code-without-comments-wastes-my-time-c7251479975f#.g0jujzf7f

```
///
/// - Parameters:
///   - x: A value to compare.
///   - y: Another value to compare.
/// - Returns: The lesser of `x` and `y`.
///   If `x` is equal to `y`, returns `x`.
public func min<T : Comparable>(_ x: T, _ y: T) -> T {
  // In case `x == y` we pick `x`.
  // This preserves any pre-existing order in case `T` has identity,
  // which is important for e.g. the stability of sorting algorithms.
  // `(min(x, y), max(x, y))` should return `(x, y)` in case `x == y`.
  return y < x ? y : x
}

/// A hook for playgrounds to print through.
public var _playgroundPrintHook :
    ((String) -> Void)? = {_ in () } // Non-nil starting value
```

Factoring Out "We" Voice

The preceding Apple-sourced example demonstrates "we" voice: "In case x == y, we pick x." Let the code or algorithm be the actor ("In case x == y, pick x.") rather than the developer. The following examples refactor away "we" voice instances sourced from versions of the standard library. The updated comments are simpler and code-specific:

Original	Revised
unit0 is a low-surrogate. We have an ill-formed sequence. (Unicode.swift)	unit0 is a low-surrogate. This sequence is ill formed.
We perform the operation on UInts to get faster unsigned math (shifts). (UnsafeBitMap.swift)	Perform the operation on UInts for faster unsigned math (shifts).
Can we get Cocoa to tell us quickly that an opaque string is ASCII? Do we care much about that edge case? (StringCore.swift)	Can Cocoa provide quick detection of opaque ASCII strings? Check whether the edge case is important.
This is a class—we require reference semantics to keep track of how many elements we've already dropped from the underlying sequence. (Sequence.swift)	This class's reference semantics track how many elements were already dropped from the underlying sequence.
Returns the actual number of Elements we can possibly store. (HeapBuffer.swift)	Returns the actual number (or *capacity*) of Elements that can be stored within the buffer.

Phrasing Comments

Explain *what* using code. Explain *why* using comments. When tempted to add *what* comments to your code, consider whether the comments can be mitigated with better symbol naming, formatting, and code organization. Comments don't fix bad code names:

```
// no
var x = q + j // add shoe height to body height

// yes
var fullHeight = bodyHeight + shoeLiftHeight
```

Reserve comments for information that code cannot explain. The notion of self-documenting code stops when your code tries to express the reasoning behind its design. To discuss a non-obvious design point, lay out the rationale and context of your decisions:

```
// Thank you, Paul Cantrell for this example

// no

func process(typeFooItemToAvoidBarMaterializationCost: Foo) {
    ...
}

/// Processes the item.
///
/// - Note: In this example, `item` is typed `Foo` because Library X sends `Foo`
/// instances. `Bar` conversions incur significant materialization costs.
func process(item: Foo) { ... } // yes
```

In the first example, the code attempts to be the documentation. The result is unwieldy and clunky. The second example moves the design issues into comments and out of the code.

When writing these comments, follow these guidelines:

- Describe actions and roles in present tense ("initializes the private store," "functions as the operation's cache").
- Describe resolution in past tense ("fixed the overflow issue"). Resolution comments can mirror commit messages but provide insight into why and how the change took place. They travel with the code in a way that commit messages often cannot.
- Describe state in present tense ("data is now guaranteed to be non-empty").

Organizing with Bookmarks

Swift offers special bookmarking keywords to support code organization:

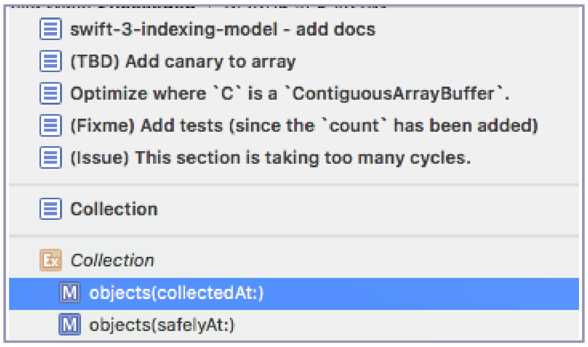
- Document code issues by adding `TODO:` and `FIXME:` comments.
- Use `MARK:` to subdivide your code's named sections.

Here are examples of what these bookmarks look like in code, sourced from the standard library. Xcode 10.2 and later supports custom icons for FIXME (BAND-AIDs) and TODO (bullet list), allowing you to differentiate the three kinds by sight. Prefixes remain helpful, though. They help distinguish, for example, what kind of fix is required:

```
// TODO: swift-3-indexing-model - add docs

// TODO: (TBD) Add canary to array
//       Putting a canary at the end of the array
//       in checked builds might be a good idea

// TODO: Optimize where `C` is a `ContiguousArrayBuffer`.

// FIXME: (Fixme) Add tests (since the `count` has been added)

// FIXME: (Issue) This section is taking too many cycles.
//        (Does this need `redraw` content mode?)

// MARK: - Collection -
```

Xcode automatically incorporates bookmarked items into a file's jump bar. The jump bar is the path control located just below the toolbar in the editor. Click and hold the final item in the file navigation trail to view a source code file's overview, which includes its top-level declarations.

Your MARKs, FIXMEs, and TODOs help structure your file overview. Add a hyphen before and/or after text to incorporate horizontal line separators into the jump table. The location of the hyphen corresponds to the location of the divider, as shown here.


swift-3-indexing-model - add docs
(TBD) Add canary to array
Optimize where `C` is a `ContiguousArrayBuffer`.
(Fixme) Add tests (since the `count` has been added)
(Issue) This section is taking too many cycles.
Collection
Collection
objects(collectedAt:)
objects(safelyAt:)

Here are some guidelines when working with bookmarks:

- Prefer TODO: when there's a known solution.
- Prefer FIXME: when an issue is identified but a fix is not yet planned out.
- Summarize each FIXME and TODO on a single line. Add supporting descriptions to successive lines so they don't appear in the jump list.
- For each TODO, describe the required task and, where appropriate, add a reason motivating the change. Include bug numbers where work is planned but broken up into several tickets or user stories.
- For each FIXME, incorporate a description of what is wrong, your explanation (or guess) of why it is wrong, and known or potential avenues for resolving the issue.
- Leave an empty line before and after MARK comments. The whitespace helps establish clear sections in your code as you move from one bookmarked section to the next.

Removing Boilerplate Code

Xcode auto-generates skeletons to support novice users who may not be aware of core methods that support particular Cocoa and Cocoa Touch APIs. Prefer to remove unused implementations that linger in code and form unused rot that detracts from the code document, or carefully annotate them with TODO marks indicating the work left to be performed:

```
// No
func applicationWillResignActive(_ application: UIApplication) {
    // Sent when the application is about to move from active to inactive
    // state. This can occur for certain types of temporary interruptions
    // (such as an incoming phone call or SMS message) or when the user
    // quits the application and it begins the transition to the background
    // state. Use this method to pause ongoing tasks, disable timers, and
    // invalidate graphics rendering callbacks. Games should use this method
    // to pause the game.
}

// Yes
func applicationWillResignActive(_ application: UIApplication) {
    // TODO: disable timer before delegating control
}
```

Addressing Consistent Fatal Outcome

Many developers push beyond simple TODO marks to use active runtime checks that ensure unfinished code does not get passed to production. Developer libraries may implement custom exit points like the following example to provide that functionality:

```
/// Handles unimplemented functionality with site-specific information
public static func unimplemented(
  _ action: String,
  _ function: String = #function,
  _ file: String = #file
) -> Never {
  fatalError("\(function) in \(file) has not been implemented. TODO: \(action)")
}

func fireUpDatabaseStore {
    unimplemented("must establish data store")
}
```

Universal Code Outcomes

Four universal core language concepts can be better represented in Swift. They are:

- This code is unreachable.
- This code is not yet implemented.
- This method must be overridden in a subtype.
- This code should never be called.

These items represent common scenarios that are fundamental to development. Consider adding uniform outcome methods of some design to your house toolbox. I personally like extending fatalError, as in the following example.

```
fatalError(because: .unreachable)
fatalError(because: .unimplemented("disable timer before delegating control"))
fatalError(because: .mustBeOverridden)
fatalError(because: .doNotCallThis)
```

Using old C-style global functions is not to everyone's taste. Equivalent Abort.because(...) or Trap.because(...) types and methods are easily constructed. They allow extensible uniform feedback about both universal fatal outcomes and in-house-specific ones, ensuring that common patterns that pop up over and over are coded and executed uniformly.

```
extension FatalReason {
  static func unconstructableURL(_ urlString: String) -> FatalReason {
    return FatalReason("Cannot construct URL from \"\(urlString)\"")
  }
}
```

Unreachable

The Swift compiler cannot detect that the following code is exhaustive. It emits an error and recommends a default clause.

```
switch Int.random(in: 0...2) {
case 0: print("got 0")
case 1: print("got 1")
case 2: print("got 2")
}
```

```
// error: switch must be exhaustive
// note: do you want to add a default clause?
```

That default clause should annotate the code so it's clear that the clause is unreachable and will never run. If it does run, there is a serious breach in logic that should result in a fatal outcome.

Not Yet Implemented

In the TODO scenario, a type member is reachable, but calling it should trap at runtime because the logic has not yet been implemented.

```
public func semanticallyResonantMethodName() {
    // TODO: implement this
}
```

This call should establish why the code cannot yet be run. Swift's compile-time diagnostics are insufficient as they don't provide runtime support:

- Using #error prevents compilation and debugging.
- Using #warning is insufficient. It can be missed at compile time and error in "warn-as-errors" development houses.
- Using TODO may similarly be too easily overlooked.

There should be a way to trap at runtime to indicate a currently unsafe avenue of execution.

Must Override

It's not uncommon to build an abstract supertype that requires methods and properties to be implemented in concrete subtypes:

```
class AbstractSupertype {
    func someKeyBehavior() {
        // Must override in subclass
    }
}
```

This method should never be called, as the type is abstract.

Must Not Be Called

Silencing warnings should not contradict Swift safety best practices. Some required members may add functionality that should never be called:

```
required init?(coder: NSCoder) {
  super.init(coder: coder)
  // Xcode requires unimplemented initializer
}
```

A method that is left *deliberately* unimplemented rather than as a TODO item should be annotated as such. Consider adding a fatal response to catch errant calls during your design process.

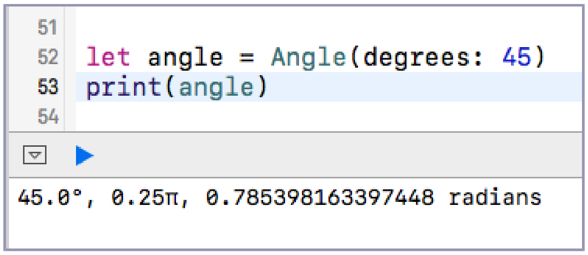
Improving Code Descriptions

Inspectable always trumps opaque. You enhance a custom type by conforming it to the CustomStringConvertible and CustomDebugStringConvertible protocols and/or providing custom reflection. As the names suggest, these protocols describe behaviors that convert instances to string presentations. Each uses a custom text representation property: description for print and debugDescription for debugPrint.

Like string convertibles, custom mirrors enable you to expand debugger and dump() output. You implement CustomReflectable by adding a customMirror property. Return a dictionary that lists the property names you want to present and their values. A mirrored value can be native, such as a member property, or computed, such as a rectangle's area or a vector's magnitude.

It's worth investing time to create better representations for testing and debugging. For example, you might create a custom Angle type that reports itself in degrees, number of pi, and radians. From an implementation viewpoint, it's necessary to store only a single number, perhaps a floating-point radians value. During debugging, it's valuable to add developer-facing descriptions that present type semantics over implementation structure. These semantics provide more meaningful feedback, as in the following example and screen shot.

```
public struct Angle: CustomStringConvertible {
    public var description: String {
        return "\(degrees)°, \(piCount)π, \(radians) radians"
    }
   ...
}
```



```
let angle = Angle(degrees: 45)
print(angle)
```

```
45.0°, 0.25π, 0.785398163397448 radians
```

Swift's class reflection is particularly limited. Unlike value types, instances show their type name without exposing details of underlying type members:

```
enum Enum { case one(Int, String) }
print(Enum.one(1, "two")) // one(1, "two")

struct Struct {
    let a = 1
    let b = "two"
}
print(Struct()) // Struct(a: 1, b: "two")

class Class {
    let a = 1
    let b = "two"
}
print(Class()) // Class
```

Developer Mike Ash created a simple way to provide a better default representation for reference types. My implementation of his approach establishes a simple protocol with an extension that adds automatic reflection for conforming members:

```
public protocol DefaultReflectable: CustomStringConvertible {}

/// Enable class members to display their values
extension DefaultReflectable {

    /// Constructs a better representation using reflection
    internal func DefaultDescription<T>(instance: T) -> String {

        let mirror = Mirror(reflecting: instance)
        let chunks = mirror.children.map({
            (label: String?, value: Any) -> String in
            guard let label = label else { return "\(value)" }
            return value is String ?
                "\(label): \"\(value)\"" : "\(label): \(value)"
        })

        if chunks.isEmpty { return "\(instance)" }
        let chunksString = chunks.joined(separator: ", ")
        return "\(mirror.subjectType)(\(chunksString))"
    }

    /// Conforms to CustomStringConvertible
    public var description: String {
        return DefaultDescription(instance: self)
    }
}

extension Class: DefaultReflectable {}
print(Class()) // Class(a: 1, b: "two")
```

Although this topic may seem out of scope at first glance, establishing robust representations for your debugging output plays an essential role in good Swift style. Prefer to enhance the way your code supports state presentation, inspection, and exploration.

- Support type descriptions with custom string output.
- Support debugger output with custom mirrors.
- Reflect the semantics as well as the underlying structure of the types you create. This extra information is invaluable for debugging.

Avoiding Clever

No one likes clever code or clever comments—not even the person who was so pleased when she first wrote that material. As Brian Kernighan wrote in *The Elements of Programming Style*,

> “Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?”

Developer Justin Etheredge wrote a wonderful essay about this called “Don’t be clever.”[11] He writes that simplicity and understandability triumphs over cleverness. Good code qualities include loose coupling, testability, exception handling, and generally good design. It’s better, in his opinion, to be a good software engineer than to win the International Obfuscated C Code Contest.[12] When writing code, emphasize the practical over showy. Simple, durable, and reliable code is your primary goal.

- “Programming is not like being in the CIA, you don’t get credit for being sneaky.” —Steve McConnell (*Code Complete*, Microsoft Press, 2004).

11. http://www.codethinked.com/dont-be-clever
12. https://www.ioccc.org

- Saying “I’m a one-person shop so I don’t have to document” doesn’t take into account the fact that future you is a primary client of both your code and your comments.
- Minimize humor because humor travels poorly. A symbol or comment that’s meant to be self-deprecating may misfire at some future point.
- Use symbol names and comment phrasing that does not insult your cubicle mate, your manager, or the persons who developed the API you’re struggling with. Get out of the habit of using profanities or inside jokes in code. You may not *plan* to ever check a snarky log message into your repo but, life being what it is, those inappropriate strings often end up in production code.
- Use whitespace liberally to avoid “blocky info-dump of doom” scenarios. Let carefully placed line breaks chunk information into more manageable concepts.
- Respect the time of the person reading your code. Keep your comments short, to the point, and valuable to the reader.
- Use a linter. Linters do more than remove trailing whitespace on lines and in files. They allow you to automate common scanning tasks so you can focus your energies on writing and debugging code, not proofing its structure. The SwiftLint[13] project is the most prevalent community linter at this time.

Wrapping Up

You’ve now read about ways to future-proof your code. No matter how hard you try to stay current on your codebase, time and memory work against you. The realities of human cognition mean that investing time in documenting, structuring, and refining code will pay off as your code ages and matures and its recency diminishes.

You’ve discovered how Swift implements a custom markup system to support inline documentation and how to mix and match doc markup with traditional comment systems. You’ve read about bookmark organization and how to improve the quality of words that make up your code. You’ve also considered how clever code can weigh down a project by limiting its maintainability and flexibility.

Now that you’ve spent time reading about designing for resilience, turn your attention to the qualities that make code great. This book concludes with a short meditation on good code. Read on for a few inspiring words.

13. https://github.com/realm/SwiftLint

CHAPTER 6

Good Code

Good software code doesn't just compile; it communicates. It talks to you as a developer here and now and to future you, who will be maintaining this code. It is comprehensible to your team and to anyone who will look at it externally.

It always chooses reliable and testable over clever, and it prefers to take an extra step or ten if needed to make its intent and outcome clear.

It's code that thinks about scale, about living in a forest as well as being a single tree, and about being run a million times at once as well as once.

It's code that believes in being broken down into sensible units instead of massive single implementations.

It's code that exposes carefully considered and easily consumed APIs without sharing unnecessary internal implementation details.

It's code that understands that it will be implemented not once or twice but reviewed and refactored to the best and most robust implementation possible.

It's code that thinks globally and provides hooks for many cultures and multi-abled users, which is a real code-level thing—how you add internationalization and accessibility is not an afterthought.

It's code that thinks about not just the specifics of the destinations it will run on today but also how to adapt for when those destinations change over time.

It's code that ships. Good code works, does what is needed, and also is delivered on time.

> "*Great code leaves you in awe and inspired to go write your own great code.*"
>
> —Dave DeLong

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line "Book Feedback."

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2019 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

And thank you for your continued support,

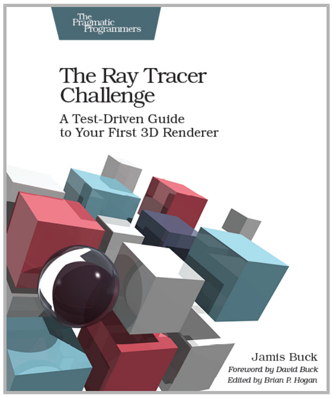
Andy Hunt, Publisher

Pragmatic Bookshelf

SAVE 30%!
Use coupon code
BUYANOTHER2019

The Ray Tracer Challenge

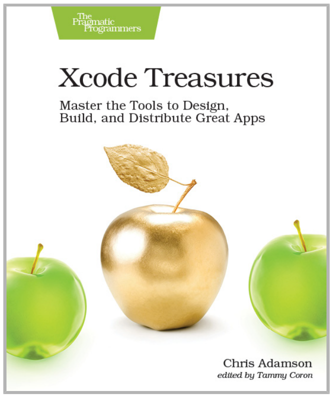
Brace yourself for a fun challenge: build a photorealistic 3D renderer from scratch! It's easier than you think. In just a couple of weeks, build a ray tracer that renders beautiful scenes with shadows, reflections, brilliant refraction effects, and subjects composed of various graphics primitives: spheres, cubes, cylinders, triangles, and more. With each chapter, implement another piece of the puzzle and move the renderer that much further forward. Do all of this in whichever language and environment you prefer, and do it entirely test-first, so you know it's correct. Recharge yourself with this project's immense potential for personal exploration, experimentation, and discovery.



Jamis Buck
(290 pages) ISBN: 9781680502718. $45.95
https://pragprog.com/book/jbtracer

Xcode Treasures

Learn the critical tips and techniques to make using Xcode for the iPhone, iPad, or Mac easier, and even fun. Explore the features and functionality of Xcode you may not have heard of. Go under the hood to discover how projects really work, so when they stop working, you'll know how to fix them. Explore the common problems developers face when using Xcode, and find out how to get the most out of your IDE. Dig into Xcode, and you'll discover it's richer and more powerful than you might have thought.

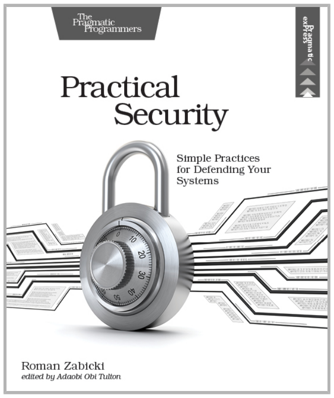

The Pragmatic Programmers
Xcode Treasures
Master the Tools to Design, Build, and Distribute Great Apps
Chris Adamson
edited by Tammy Coron

Chris Adamson
(274 pages) ISBN: 9781680505863. $45.95
https://pragprog.com/book/caxcode

Practical Security

Most security professionals don't have the words "security" or "hacker" in their job title. Instead, as a developer or admin you often have to fit in security alongside your official responsibilities — building and maintaining computer systems. Implement the basics of good security now, and you'll have a solid foundation if you bring in a dedicated security staff later. Identify the weaknesses in your system, and defend against the attacks most likely to compromise your organization, without needing to become a trained security professional.

Roman Zabicki
(132 pages) ISBN: 9781680506341. $26.95
https://pragprog.com/book/rzsecur

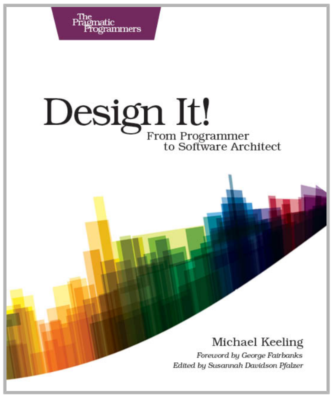


Release It! Second Edition

A single dramatic software failure can cost a company millions of dollars—but can be avoided with simple changes to design and architecture. This new edition of the best-selling industry standard shows you how to create systems that run longer, with fewer failures, and recover better when bad things happen. New coverage includes DevOps, microservices, and cloud-native architecture. Stability antipatterns have grown to include systemic problems in large-scale systems. This is a must-have pragmatic guide to engineering for production systems.

Michael Nygard
(376 pages) ISBN: 9781680502398. $47.95
https://pragprog.com/book/mnee2

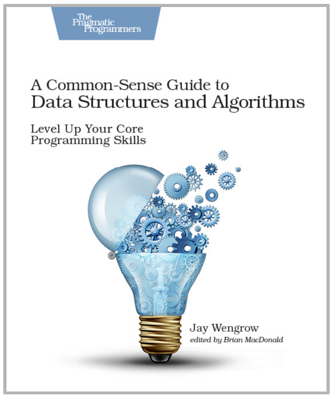
Design It!


The Pragmatic Programmers
Design It!
From Programmer to Software Architect
Michael Keeling
Foreword by George Fairbanks
Edited by Susannah Davidson Pfalzer

Don’t engineer by coincidence—design it like you mean it! Grounded by fundamentals and filled with practical design methods, this is the perfect introduction to software architecture for programmers who are ready to grow their design skills. Ask the right stakeholders the right questions, explore design options, share your design decisions, and facilitate collaborative workshops that are fast, effective, and fun. Become a better programmer, leader, and designer. Use your new skills to lead your team in implementing software with the right capabilities—and develop awesome software!

Michael Keeling
(358 pages) ISBN: 9781680502091. $41.95
https://pragprog.com/book/mkdsa

A Common-Sense Guide to Data Structures and Algorithms

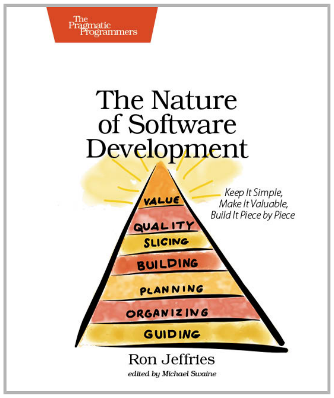

The Pragmatic Programmers
A Common-Sense Guide to Data Structures and Algorithms
Level Up Your Core Programming Skills
Jay Wengrow
edited by Brian MacDonald

If you last saw algorithms in a university course or at a job interview, you’re missing out on what they can do for your code. Learn different sorting and searching techniques, and when to use each. Find out how to use recursion effectively. Discover structures for specialized applications, such as trees and graphs. Use Big O notation to decide which algorithms are best for your production environment. Beginners will learn how to use these techniques from the start, and experienced developers will rediscover approaches they may have forgotten.

Jay Wengrow
(220 pages) ISBN: 9781680502442. $45.95
https://pragprog.com/book/jwdsal

The Nature of Software Development

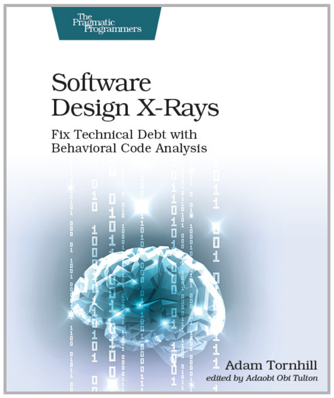
You need to get value from your software project. You need it "free, now, and perfect." We can't get you there, but we can help you get to "cheaper, sooner, and better." This book leads you from the desire for value down to the specific activities that help good Agile projects deliver better software sooner, and at a lower cost. Using simple sketches and a few words, the author invites you to follow his path of learning and understanding from a half century of software development and from his engagement with Agile methods from their very beginning.


The Pragmatic Programmers
The Nature of Software Development
Keep It Simple, Make It Valuable, Build It Piece by Piece
VALUE
QUALITY
SLICING
BUILDING
PLANNING
ORGANIZING
GUIDING
Ron Jeffries
edited by Michael Swaine

Ron Jeffries
(176 pages) ISBN: 9781941222379. $24
https://pragprog.com/book/rjnsd

Software Design X-Rays

Are you working on a codebase where cost overruns, death marches, and heroic fights with legacy code monsters are the norm? Battle these adversaries with novel ways to identify and prioritize technical debt, based on behavioral data from how developers work with code. And that's just for starters. Because good code involves social design, as well as technical design, you can find surprising dependencies between people and code to resolve coordination bottlenecks among teams. Best of all, the techniques build on behavioral data that you already have: your version-control system. Join the fight for better code!


The Pragmatic Programmers
Software Design X-Rays
Fix Technical Debt with Behavioral Code Analysis
Adam Tornhill
edited by Adaobi Obi Tulton

Adam Tornhill
(274 pages) ISBN: 9781680502725. $45.95
https://pragprog.com/book/atevol

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

https://pragprog.com/book/esswift2

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date

https://pragprog.com

Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

https://pragprog.com/news

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/esswift2*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764