Extracted from:

Swift Style

An Opinionated Guide to an Opinionated Language

This PDF file contains pages extracted from *Swift Style*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2017 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.

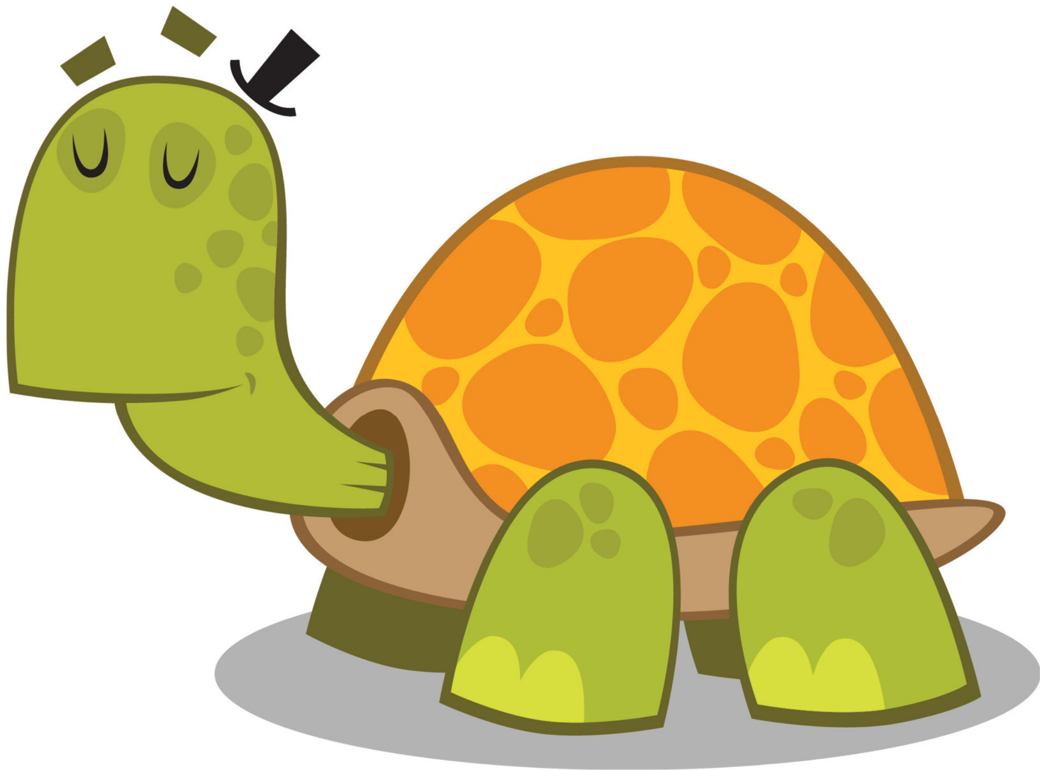
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Swift Style

An Opinionated Guide to an Opinionated Language



Erica Sadun

edited by Brian MacDonald

Swift Style

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*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Brian MacDonald
Indexing: Potomac Indexing, LLC
Copy Editor: Linda Recktenwald
Layout: Gilson Graphics

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

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

Copyright © 2017 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.

Printed in the United States of America.
ISBN-13: 978-1-68050-235-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2017

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. This kind of fragility is counter to good style practices. Robust coding allows you to add, delete, and adapt lines without those changes rippling out to affect other lines of code.

Improving Closure Hygiene

Each closure starts with 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.

• Click HERE to purchase this book now. discuss

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
    ...
}
```

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

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

- Using `inout` parameters almost always requires more detailed declaration lines. The Swift compiler will emit warnings and errors to guide you when you've added insufficient details to your closure signatures.
- Some closures require explicit signatures when the closure performs a transformation on the input types to a new output type. Keep this in mind when debugging closure code.

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 your use of shorthand argument names to the simplest closures, such as those used when mapping, filtering, and sorting. In these cases, your focus is fixed on the call (for example, > in `{ $0 > $1 }`) rather than on the arguments being passed to that call.

In such simple circumstances, you can often pass an operator (>) or function name (min) in place of the closure:

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

Mapping a 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 named function if one is not already available. Don't create special-purpose single-use functions to avoid mapped closures.

When your closure extends beyond a line or two, establish argument names. Names ensure you won't perform mental gymnastics trying to remember what roles `$0` and `$1` play in the closure context. Always promote recognition over recall in code design. Adding names allows you to recognize each parameter in context. This involves a much lower cognitive burden than recall, where you must retrieve the role of each positional argument from memory.

- 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 parameters are worthless.

Weighing Colinear in

Many 1TBS adherents place signature declarations on the same line as the closure's opening brace. It 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
        ...
```

Mitigate over-long lines by moving declarations from the 1TBS opening brace to the following line, as in the following examples. Although less compact, this approach aligns a closure signature with the code that follows.

```
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 creates 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, depending on the degree of generics involved.

Avoid breaking down the in line 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.

A 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. Prefer to separate a closure declaration from its body more conventionally by using 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

return 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 later evaluation when passing the expression as an argument to a function. Its syntactic sugar enables you to omit functional braces.

```
func either(_ test: Bool, or action: @escaping @autoclosure () -> Void) {
    guard !test else { return }
    action()
}

either(count < 10, or: print("Done"))
```

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.
- 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 Kevin (courtesy of Kevin Ballard, one of my technical reviewers and an amazing iOS developer at Postmates) adds parentheses around trailing closures when the argument is functional (that is, it returns a value, avoiding state changes and mutating data), restricting bare braces to procedural calls (updates state and/or has 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.characters.split(separator: " ")
    .filter({ $0.count > 4 })
    .map({ String($0) })
```

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

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

Proactive Compilation Safety

The Rule of Kevin 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
for value in items.map { pow($0, 2) } { // bad
    print(value)
}

// Compiles
for value in items.map({ pow($0, 2) }) { // good
    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 Kevin 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 Kevin, 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 {
        print("Error convolving image")
        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, the latest Swift compilers 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 Kevin 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.

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

// Not okay
UIView.animate(withDuration: 2.0,
    animations: { v.removeFromSuperview() }) {
    _ in postNotification()
}
```