Extracted from:

Swift Style, Second Edition

An Opinionated Guide to an Opinionated Language

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Swift Style Second Edition

An Opinionated Guide to an Opinionated Language

Erica Sadun

edited by Brian MacDonald

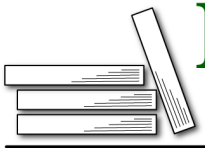
Swift Style, Second Edition

An Opinionated Guide to an Opinionated Language

Erica Sadun

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

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*.

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

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

ISBN-13: 978-1-68050-627-3
Book version: B1.0—August 15, 2018

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 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") }
```

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.