Extracted from:

Swift Style

An Opinionated Guide to an Opinionated Language

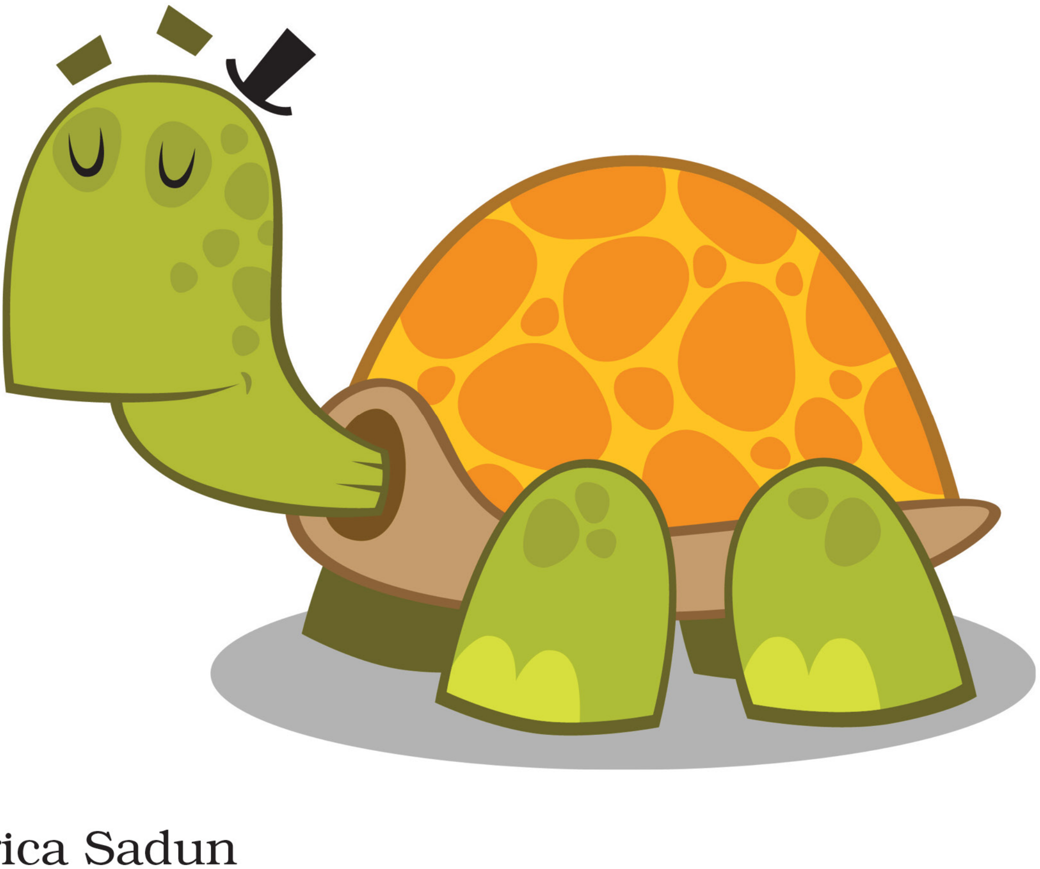
The Pragmatic Bookshelf

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

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

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

Placing Attributes

Placing attributes like @objc, @testable, @available, and @discardableResult on their own lines before a member declaration has become a widely adopted Swift style:

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

@available(*, unavailable, renamed: "Iterator") // yes
public typealias Generator = Iterator

@discardableResult // yes
func _copyContents(initializing ptr:
  UnsafeMutablePointer<Iterator.Element>)
  -> UnsafeMutablePointer<Iterator.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 leftmost column.

Many developers mix and match styles for short Swift attributes like @objc. It's common to 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
```

Some prefer to break attributes out universally, regardless of length. This approach is consistent, and it promotes consistent leftmost access modifiers:

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

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

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

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 some kind of 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 spacing, 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 (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.

```
let test1 = 1_0_0_1 // no
let test2 = 1_0_0__1 // especially no
let test3 = 0118_999_881_99_9119_725 // very very no
```

There's one exception to the "only use decimal/hex chunking" rule. When working with number literals that represent structured information such as social security numbers or phone numbers, it is *occasionally* acceptable ("But, oh god, never ever do this, please!") to use nonstandard chunking.

```
let social = 045_68_4425
let phone = 202_456_1111 // US style
```

Strings are almost always a better solution unless you have a more structured value type that breaks items into shorter subcomponents. Remember, half of phone numbers overflow 32-bit integers.

- 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.

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.

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. 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 any 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 to same-type arguments 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) // your call
```