Extracted from:

Swift Style, Second Edition

An Opinionated Guide to an Opinionated Language

This PDF file contains pages extracted from *Swift Style, Second Edition*, 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 © 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.

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

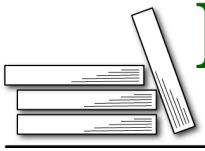
Swift Style, Second Edition

An Opinionated Guide to an Opinionated Language

Erica Sadun

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

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*.

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: B1.0—August 15, 2018

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 uncompli-

cated 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
    ...
}
```