熊谷さんの

やさしい Swift 勉強会

for YUMEMI

文化系の習い事のように、言語仕様に長く親しむ場所

- プログラミング言語の 存在に意識を傾けるきっかけ にしてほしい
- プログラミング言語に向き合うことで、各自が自身で考える力を養う ことにも繋がってほしい

本日の話題

Swift API Design Guidelines

- Swift のコードを標準化するためのガイドライン

Swift 5.4.2

API デザインガイドライン

目的

- Swift のコードを書くときの公式指標
- 標準化されたコードを提供するのが目的
  - その多くは名前付けや慣用表現に現れてくる
- 書いたコードが大きな Swift エコシステムの一部のように感じられる方法を説明する

基本事項

使用時の明瞭さを最重視

- これが最も重視されている目標
- メソッドやプロパティーは唯 1 度定義され、繰り返し使用される
- 用途を明瞭かつ明確に表現するために API をデザインする
- API のデザインを評価するために
  - 宣言部分を読むだけでは不十分
  - API を使うときに そのコンテキスト内で明確に見える こと

短さよりも明瞭さを意識

- Swift では「少ない文字数で可能な限り短いコードを書くこと」は目標としない
- Swift でコードが短くなるのは
  強力な型システムの副次的な恩恵 ⇨ 定型文が自然に減少

ドキュメントコメントの記載

- 全ての宣言にドキュメントコメントを記載する
- これを書くことが API デザインに大きな影響を与えるので 後回しにしないこと

※ API の機能を簡潔に説明できない場合、デザインが間違っているかもしれない

ドキュメントコメント

概要

- Markdown のような記法 が使える
- 宣言の概要から書き始める
  多くの API は、宣言と概要説明があれば完全に理解できる
- 必要に応じて、文章や箇条書きで説明を加える

概要説明（サマリー）

- Markdown のような記法 が使える
- 宣言の概要説明から書き始める
  多くの API は、宣言と概要説明があれば完全に理解できる
- 必要に応じて、文章や箇条書きで説明を加える

概要説明（サマリー）を最重視

- 概要説明が最重要
- 多くの優れたドキュメントコメントは、優れた概要説明だけで構成される
- 可能なら 断片的な単一文章を使い、句点で終える ようにする
  - 完全な文章は使用しない
  - 稀に 断片的な文章をセミコロンで区切って連ねる場合も ある

```
/// Returns a "view" of `self` containing the same elements in reverse order.
func reversed() -> ReverseCollection

/// Removes and returns the first element if non-empty; returns `nil` otherwise.
mutating func popFirst() -> Element?
```

概要説明（サマリー）での記載事項

- 何をするのか、何を返すのか を記載
- 効果のないことや、戻り値が `Void` であることには触れない

```
/// Inserts `newHead` at the beginning of `self`.
mutating func prepend(_ newHead: Int)

/// Returns a `List` containing `head` followed by the elements of `self`.
func prepending(_ head: Element) -> List
```

添字構文における概要説明

- 添字構文が 何にアクセスするか を記載する

```
/// Accesses the `index`th element.
subscript(index: Int) -> Element { get set }
```

イニシャライザーにおける概要説明

- イニシャライザーが 何を生成するか を記載する

```
/// Creates an instance containing `n` repetitions of `x`.
init(count n: Int, repeatedElement x: Element)
```

そのほかの宣言における概要説明

- 宣言されたエンティティーが 何であるか を記載する

```
/// A collection that supports equally efficient insertion/removal at any position.
struct List {

  /// The element at the beginning of `self`, or `nil` if self is empty.
  var first: Element?
  ...
```

文章や箇条書きによる追加説明

- 概要説明のあとに空行を挟んで、追加の説明を記載しても良い
- 各段落は空行で区切る
- シンボル・ドキュメンテーション・マークアップを使って追加情報を埋め込める
- 箇条書きでは シンボルコマンド構文 が使える

シンボルコマンド構文で使えるキーワード例

Attention	Author	Authors	Bug	Complexity	Copyright
Date	Experiment	Important	Invariant	Note	Parameter
Parameters	Postcondition	Precondition	Remark	Requires	Returns
SeeAlso	Since	Throws	ToDo	Version	Warning

文章や箇条書きによる追加説明（記載例）

```
/// Writes the textual representation of each
/// element of `items` to the standard output.
///
/// The textual representation for each item `x`
/// is generated by the expression `String(x)`.
///
/// - Parameter separator: text to be printed between items.
/// - Parameter terminator: text to be printed at the end.
///
/// - Note: To print without a trailing
///   newline, pass `terminator: ""`
func print(_ items: Any..., separator: String = " ", terminator: String = "\n")
```

← 概要説明

← 空行

← 追加説明

← シンボルコマンド構文

命名規則

明瞭な使用を促していく

必要なすべての語句を含める

- コードを読む人が解釈し間違えないようにするのが狙い

■ 指定されたインデックスの要素を削除するメソッド

インデックスで指定した要素を削除することが明確に伝わるデザイン

```
employees.remove(at: x)
```

要素から`x`で指定された要素を削除するように読めるデザイン

```
employees.remove(x)
```

必要のない語句は省く

- すべての語句が それぞれの場所で際立って意味を伝える 必要がある
- 読み手が既に知っている情報は重複させないで省略する
- 特に "型の情報" を繰り返すだけの語句は省略する

この例では`Element`が際立った意味を提供しない

```
mutating func removeElement(_ member: Element) -> Element?
allViews.removeElement(cancelButton)
```

意味を提供しない言葉を省略すると扱いがより明瞭になる

```
mutating func remove(_ member: Element) -> Element?
allViews.remove(cancelButton)
```

変数・引数・関連型の名前

- 型の制約ではなく 役割に応じて 名前をつける

型名を使っても明瞭さや表現力は高まらない

```
var string = "Hello"
func restock(from widgetFactory: WidgetFactory)
```

```
protocol ViewController {
  associatedtype ViewType : View
```

エンティティーの役割を表現する名前をつけると明瞭になる

```
var greeting = "Hello"
func restock(from supplier: WidgetFactory)
```

```
protocol ViewController {
  associatedtype ContentView : View
```

関連型とプロトコルの関係

- 関連型はプロトコルの制約と密接に結びついていたりする
- プロトコル名が役割になっているときは 接尾辞として `Protocol` を付与

```
protocol Sequence {
  associatedtype Iterator : IteratorProtocol
}

protocol IteratorProtocol { ··· }
```

自由度のある型情報を補う

- 制約の少ない型の場合は 型情報と文脈だけでは意図が伝わらないことがある
- 特に `Int` や `String` などの基礎型や、`NSObject`, `Any`, `AnyObject` などの汎用型には注意

定義は明瞭だが、使用時に不明瞭となる例

```
func add(_ observer: NSObject, for keyPath: String)
grid.add(self, for: graphics)
```

型情報の弱いパラメーターに、その役割を説明する名詞を添えると明瞭になる

```
func addObserver(_ observer: NSObject, forKeyPath path: String)
grid.addObserver(self, forKeyPath: graphics)
```

命名規則

流暢な表現を目指して

関数名・メソッド名

- 関数名は英文法として正しい名前を選ぶ
- 外部引数名も含めて表現する

英文法として流暢な表現

```
x.insert(y, at: z)            // x, insert y at z
x.subViews(havingColor: y)    // x's subviews having color y
x.capitalizingNouns()         // x, capitalizing nouns
```

英文法として違和感のある表現

```
x.insert(y, position: z)
x.subViews(color: y)
x.nounCapitalize()
```

意味の中心にはない引数ラベルの扱い

- 関数呼出の中で主体にならない引数は、英文法としての流れが崩れても良い

```
AudioUnit.instantiate(
                with: description,
                options: [.inProcess],
                completionHandler: stopProgressBar)
```

ファクトリーメソッド

- ファクトリーメソッドの名前は make で始める

```
x.makeIterator()
```

イニシャライザーとファクトリーメソッド

- 第１引数の名前は、そのベース名から始まる英文にしない
- このルールにより原則として、最初の引数はラベルを持つことになる

引数ラベルは、ベース名が作る英文として構成されていない

```
let foreground = Color(red: 32, green: 64, blue: 128)
let newPart = factory.makeWidget(gears: 42, spindles: 14)
let ref = Link(target: destination)
```

引数ラベルを英文の一部として含めようとした例

```
let foreground = Color(havingRGBValuesRed: 32, green: 64, andBlue: 128)
let newPart = factory.makeWidget(havingGearCount: 42, andSpindleCount: 14)
let ref = Link(to: destination)
```

副作用の有無に応じた関数名

- 副作用を持たない関数は 名詞 で名付ける
- 副作用を持つ関数は 命令形の動詞 で名付ける

副作用を持たない関数

```
x.distance(to: y)
i.successor()
```

副作用を持つ関数

```
print(x)
x.sort()
x.append(y)
```

副作用の有無で対になっている関数

- 内容を更新する関数には、計算結果として内容を返す関数も併存することがある
- このような同じ役割を持ち副作用の有無が異なる関数ペアには 一貫した名前 をつける
- 名前を 動詞 で自然に表現できるとき
  - 副作用ありは、動詞の命令形
  - 副作用なしは、動詞の過去分詞形または現在分詞形（一般に接尾辞 `-ed` または `-ing` を付与）
- 名前を 名詞 で自然に表現できるとき
  - 副作用ありは、接頭辞 `form-` を付与
  - 副作用なしは、名詞そのまま

副作用を伴わない関数を動詞で表現するとき

- 名前を 過去分詞形 にする（一般に -ed を付与）
- 直接目的語を伴うような英文法に不自然なときは、名前を 現在分詞形 にする（-ing を付与）

過去分詞形で名付けるとき

```
mutating func reverse()
func reversed() -> Self

x.reverse();              let y = x.reversed()
```

現在分詞形で名付けるとき

```
mutating func stripNewlines()
func strippingNewlines() -> String

s.stripNewlines();        let oneLine = t.strippingNewlines()
```

副作用を伴う関数を名詞で表現するとき

- 名前に接頭辞 form- を付与

過去分詞形で名付けるとき

```
mutating func formUnion(_ value: Self)
func union(_ value: Self) -> Self
```

```
y.formUnion(z);
let x = y.union(z)
```

現在分詞形で名付けるとき

```
mutating func formSuccessor(_ i: inout Index)
func successor(_ i: Index) -> String
```

```
c.formSuccessor(&i);
let j = c.successor(i)
```

真偽値型のメソッドやプロパティー

- 内容変化を伴わない場合は、
  レシーバーについての 主張として読み取れるように する

```
x.isEmpty
line1.intersects(line2)
```

プロトコルの名称

- "それが何か" を表すものは 名詞 にする
- "特性" や "能力" を表現するものは 接尾辞 -able や -ible、-ing を付けた名前 にする

```
protocol Collection

protocol Equatable
protocol ProgressReporting
```

これまでのルールに合致しないもの

- これまでのルールでは決められていない型やプロパティーの名前は 名詞 にする

命名規則

専門用語を上手に使う

専門用語

- 特定の分野や職業において
  精密で専門的な意味を持つ語句のこと

難しい用語

- 一般的な言葉で意味が通じるなら、難しい用語は使わないようにする
- たとえば「Skin」で通じるなら、敢えて「epidermis」とは言わない
- 専門用語は 他の言葉では重要な意味が失われるような場合に限って 使う

専門用語の意味にこだわる

- 専門用語を使うときは、それが 確立している意味を曲げずに 使うこと
- 専門用語を使う唯一の理由は、
  他の方法では意味が曖昧になる場面で、意味を正確に表現するため
- 専門用語は意味に厳格であること
  - その用語をよく知る 専門家 が、厳密な意味と違う使われ方に驚かないように
  - その用語を知らない 初学者 が、調べて見つけた伝統的な意味と違って混乱しないように

略語は使用しない

- 特に一般的ではない略語は 専門用語 のようなもの
- 略語の意味を正確に汲むには、略されていない表現に正しく翻訳できることが求められる

既存の文化は大切に

- 用語を初学者目線で選ぶあまりに、既存の文化から逸脱しないように心がけること

連続するデータを `Array` と表現する

- `List` という語を当てる方が初学者にはわかりやすいかもしれない
- ただし、現代のソフトウェア理論で一般的な "Array" を当てた方が、
  その意味を既に知っていたり、知らなくても調べればすぐに理解できる

正弦関数を `sin(x)` と表現する

- 数学のような特定のプログラミング領域では `sin(x)` という語が広く使われている
- 説明的な `verticalPositionOnUnitCircleAtOriginOfEndOfRadiusWithAngle(x)` より好ましい
- 正弦を厳密に表現すると "sine" だが、数学的分野では "sin" が既に広く浸透している

表現法

全般的な表現法

プロパティーの計算量がデータ量に依存するとき

- 計算型プロパティーの 計算量が O(1) でないとき は、ドキュメントコメントに明記する
- プロパティーアクセスは O(1) であると思って使われがちのため 注意を促す のが目的

フリーな関数を使う場面

- フリーな関数は 特別な場面に限って使う

明確な主体が存在しないとき

```
min(x, y, z)
```

何にも制約されないジェネリック関数とき

```
print(x)
```

関数の構文が、確立された分野の表記法の一部であるとき

```
sin(x)
```

大文字小文字の区別

- 型とプロトコルの名前には Upper Camel Case を使う
- それ以外には Lower Camel Case を使う

頭字語の扱い

- アメリカ英語で全てを大文字にする語は、登場場所に応じて一律に大文字または小文字 で表記
- そのほかの頭字語は、一般の語と同じルールで表記

全てが大文字で表現される頭字語

```
var utf8Bytes: [UTF8.CodeUnit]
var isRepresentableAsASCII = true
var userSMTPServer: SecureSMTPServer
```

普通の語のように表現される頭字語

```
var radarDetector: RadarScanner
var enjoysScubaDiving = true
```

メソッドのベース名

- 基本的に同じ意味合いで使われるときは、複数のメソッドで同じベース名を使って良い

本質的に同じ機能を担っている例

```
extension Shape {
  /// Returns `true` iff `other` is within the area of `self`.
  func contains(_ other: Point) -> Bool { ... }

  /// Returns `true` iff `other` is entirely within the area of `self`.
  func contains(_ other: Shape) -> Bool { ... }
```

機能の本質が異なる例

```
extension Database {
  /// Rebuilds the database's search index
  func index() { ... }

  /// Returns the `n`th row in the given table.
  func index(_ n: Int, inTable: TableID) -> TableRow { ... }
```

メソッドのベース名

- 異なるドメインで使われるときは、複数のメソッドで同じベース名を使って良い

あるドメインに同じ名前の機能があっても・・・

```
extension Shape {
  /// Returns `true` iff `other` is within the area of `self`.
  func contains(_ other: Point) -> Bool { ... }
}
```

異なるドメインであれば違う機能に同じ名前を当てられる

```
extension Collection where Element : Equatable {
  /// Returns `true` iff `self` contains an element equal to `sought`.
  func contains(_ sought: Element) -> Bool { ... }
}
```

戻り値の型でのオーバーロード

- 同じ名前で 戻り値だけが異なるオーバーロード も可能
- ただし、型推論のある文脈で曖昧になるので 避けるのが良い

戻り値の型だけが異なるオーバーロード

```
extension Box {

  /// Returns the `Int` stored in `self`, if any, and `nil` otherwise.
  func value() -> Int? { ... }

  /// Returns the `String` stored in `self`, if any, and `nil` otherwise.
  func value() -> String? { ... }
}
```

表現法

引数の表現法

ドキュメントになる引数名を選ぶ

- 引数名は使用時には表に現れなくても、説明として大事な役割を果たす
- ドキュメントを自然に読ませる名前にする

ドキュメントを自然に読ませる引数名

```
/// Return an `Array` containing the elements of `self`
/// that satisfy `predicate`.
func filter(_ predicate: (Element) -> Bool) -> [Generator.Element]
```

ドキュメントが文章としておかしくなる引数名

```
/// Return an `Array` containing the elements of `self`
/// that satisfy `includedInResult`.
func filter(_ includedInResult: (Element) -> Bool) -> [Generator.Element]
```

ドキュメントになる引数名を選ぶ

- 引数名は使用時には表に現れなくても、説明として大事な役割を果たす
- ドキュメントを自然に読ませる名前にする

ドキュメントを自然に読ませる引数名

```
/// Replace the given `subRange` of elements with `newElements`.
mutating func replaceRange(_ subRange: Range, with newElements: [E])
```

ドキュメントがぎこちない文章になる引数名

```
/// Replace the range of elements indicated by `r` with the contents of `with`.
mutating func replaceRange(_ r: Range, with: [E])
```

引数の既定値を活用する

- 一般的にひとつの値が指定される引数で既定値が活かせる
- メソッドの使用を簡略化できる

```
let order = lastName.compare(royalFamilyName, options: [], range: nil, locale: nil)
let order = lastName.compare(royalFamilyName)
```

引数の既定値で明瞭化する

- メソッドファミリーで定義するより、既定値を使う方がコードを読む負担が軽くなる
- ドキュメントが分断されず、同等の機能であることが明確に伝わる

メソッドを１つ理解すれば済む（単一の機能であることが明瞭）

```
func compare(_ other: String, options: CompareOptions = [],
                  range: Range? = nil, locale: Locale? = nil) -> Ordering
```

それぞれのメソッドを理解しなければいけない

```
func compare(_ other: String) -> Ordering
func compare(_ other: String, options: CompareOptions) -> Ordering
func compare(_ other: String, options: CompareOptions, range: Range) -> Ordering
func compare(_ other: String, options: CompareOptions,
                  range: Range, locale: Locale) -> Ordering
```

既定値のある引数の配置

- 既定値を持つ引数は、後方に置くのが望ましい
  - 既定値を持たない引数は、重要な意味を持つことが多い
  - メソッドを使うとき、その基本的な使い方を提供できる

表現法

引数ラベル

引数を区別する必要がないとき

- 複数の引数を取る場合で、それらに区別のないときはラベルを省略する

```
min(number1, number2)
zip(sequence1, sequence2)
```

変換イニシャライザーの引数

- 値を保全した型変換 を行うときは、最初の引数 ラベルを省略 する
- 値の再解釈を伴う型変換 は、最初の ラベルで変換方法を説明 する

```
extension UInt32 {
  /// Creates an instance having the specified `value`.
  init(_ value: Int16)

  /// Creates an instance having the lowest 32 bits of `source`.
  init(truncating source: UInt64)

  /// Creates an instance having the nearest representable
  /// approximation of `valueToApproximate`.
  init(saturating valueToApproximate: UInt64)
}
```

前置詞句を構成する最初の引数

- 前置詞を 引数ラベル に含める
- ただし、後続の引数と合わせてひとつのものを表す場合 は ベース名 に含める

```
x.removeBoxes(havingLength: 12)
```

複数の引数でひとつのものを表しているのが明瞭

```
a.moveTo(x: b, y: c)
a.fadeFrom(red: b, green: c, blue: d)
```

複数の引数でひとつのものを表していることが不明瞭

```
a.move(toX: b, y: c)
a.fade(fromRed: b, green: c, blue: d)
```

前置詞句を構成しない最初の引数

- 文法的な句を構成するとき は ラベル名を省略してベース名で説明 する
- 文法的な句を構成しないとき は その引数をラベルで説明 する

```
x.addSubview(y)
```

文法的な句を構成しないときに、ラベルでの説明がある例

```
view.dismiss(animated: false)
let text = words.split(maxSplits: 12)
```

文法的な句を構成しないときに、ラベルでの説明がない例

```
view.dismiss(false)              // Don't dismiss? Dismiss a Bool?
let text = words.split(12)       // Split the number 12?
```

それ以外の引数

- 常にラベルをつける
- 既定値を持つ引数も文法的な句を構成しないため、常にラベルをつける

特記事項

API 定義内におけるタプルの要素

- 各要素にラベルを付ける
- タプルの要素にアクセスするコードの可読性が高まる

```
/// Ensure that we hold uniquely-referenced storage for at least
/// `requestedCapacity` elements.
///
/// - Returns:
///   - reallocated: `true` iff a new block of memory was allocated.
///   - capacityChanged: `true` iff `capacity` was updated.
mutating func ensureUniqueStorage(
    minimumCapacity requestedCapacity: Int)
                            -> (reallocated: Bool, capacityChanged: Bool)
```

特記事項

API 定義内におけるクロージャーの引数

- 名前で用途を説明する（ラベルは付けられないので _ を明記）
- 名前は ドキュメントコメントで役立つ名前 にする

```
/// Ensure that we hold uniquely-referenced storage for at least
/// `requestedCapacity` elements.
///
/// If more storage is needed, `allocate` is called with
/// `byteCount` equal to the number of maximally-aligned bytes to allocate.
mutating func ensureUniqueStorage(
    minimumCapacity requestedCapacity: Int),
    allocate: (_ byteCount: Int) -> UnsafePointer<Void>)
```

制約のないポリモーフィズムに注意を払う

- オーバーロードされたメソッドの役割が曖昧になることがある
- Any や AnyObject、制約のない型パラメーターを使うときに注意

どちらのメソッドを呼び出したいかが不明瞭になる定義例

```
struct Array {
    mutating func append(_ newElement: Element)       // appends a new element.
    mutating func append(_ newElements: [Element])   // appends the contents of new elements.
}
```

曖昧さを考慮した定義例

```
struct Array {
    mutating func append(_ newElement: Element)
    mutating func append(contentsOf newElements: [Element])
}
```

Enjoy! Swift