API Design Guidelines

To facilitate use as a quick reference, the details of many guidelines can be expanded individually. Details are never hidden when this page is printed.
https://swift.org/documentation/api-design-guidelines/

Table of Contents

Fundamentals(基礎)

- **Clarity at the point of use** is your most important goal. Entities such as methods and properties are declared only once but *used* repeatedly. Design APIs to make those uses clear and concise. When evaluating a design, reading a declaration is seldom sufficient; always examine a use case to make sure it looks clear in context.(在使用上清晰明確是最重要的點.函數跟屬性只被宣告一次但是會一直重複被使用.簡潔地清晰地設計API.當評估一個設計,很少地閱讀宣告;總是檢查內容是明確的的.)

- **Clarity is more important than brevity.** Although Swift code can be compact, it is a *non-goal* to enable the smallest possible code with the fewest characters. Brevity in Swift code, where it occurs, is a side-effect of the strong type system and features that naturally reduce boilerplate.(清晰比簡短還重要.雖然swift語言是緊湊的,他不是以最小最少字數為目標.Swift代碼的簡潔性是因為它的強類型系統以及自然地減少樣板的副作用)

- **Write a documentation comment** for every declaration. Insights gained by writing documentation can have a profound impact on your design, so don't put it off.(為每一個宣告寫文件註解.透過寫文件撰寫可以對你的API設計有深遠的影響,所以不要關閉它)
  If you are having trouble describing your API's functionality in simple terms, **you may have designed the wrong API.(如果你不能簡單的描述你的API,你也許設計了錯誤的API)**

  - **Use Swift's dialect of Markdown.(使用Markdown語言,參考連結)**

  - **Begin with a summary** that describes the entity being declared. Often, an API can be completely understood from its declaration and

its summary.(這個實體被宣告開始地方描述概要,通常從宣告跟描述概要可以完全了解一個API,範例如下)
/// **Returns a "view" of `self` containing the same elements in**
- /// **reverse order.**
- func reversed() -> ReverseCollection
-

- **Focus on the summary**; it's the most important part. Many excellent documentation comments consist of nothing more than a great summary.(重點總結,這是最重要的.大多數優秀的文件註解都是比較'精',少的)

- **Use a single sentence fragment** if possible, ending with a period. Do not use a complete sentence.(使用簡短的片段,不要使用完整的句子)

- **Describe what a function or method** *does* **and what it** *returns*, omitting null effects and Void returns:(敘述函數功能和回傳,忽略沒作用的null和void回傳)
  /// **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
-
- /// **Removes and returns** the first element of `self` if non-empty;
- /// returns `nil` otherwise.
- mutating func popFirst() -> Element?
-

  Note: in rare cases like popFirst above, the summary is formed of multiple sentence fragments separated by semicolons.

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

- **Describe what an initializer** *creates*:
  /// **Creates** an instance containing `n` repetitions of `x`.
- init(count n: Int, repeatedElement x: Element)
-

- For all other declarations, **describe what the declared entity** *is*.
  /// **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?
- ...
-

○

○ **Optionally, continue** with one or more paragraphs and bullet items.
Paragraphs are separated by blank lines and use complete sentences.
/// Writes the textual representation of each ← Summary
○ /// element of `items` to the standard output.
○ /// ← Blank line
○ /// The textual representation for each item `x` ← Additional discussion
○ /// is generated by the expression `String(x)`.
○ ///
○ /// - **Parameter separator**: text to be printed ⎫
○ /// between items. ⎪
○ /// - **Parameter terminator**: text to be printed ⎬ Parameters section
○ /// at the end. ⎪
○ /// ⎭
○ /// - **Note**: To print without a trailing ⎫
○ /// newline, pass `terminator: ""` ⎪
○ /// ⎬ Symbol commands
○ /// - **SeeAlso**: `CustomDebugStringConvertible`, ⎪
○ /// `CustomStringConvertible`, `debugPrint`. ⎭
○ public func print(
○ _ items: Any..., separator: String = " ", terminator: String = "\n")
○

- **Use recognized symbol documentation markup elements** to add information beyond the summary, whenever appropriate.

- **Know and use recognized bullet items with symbol command syntax.** Popular development tools such as Xcode give special treatment to bullet items that start with the following keywords:

▪ Attention	▪ Author	▪ Authors	▪ Bug
▪ Complexity	▪ Copyright	▪ Date	▪ Experiment

▪ Important	▪ Invariant	▪ Note	▪ Parameter
▪ Parameters	▪ Postcondition	▪ Precondition	▪ Remark
▪ Requires	▪ Returns	▪ SeeAlso	▪ Since
▪ Throws	▪ Todo	▪ Version	▪ Warning

○ ∘

●

Naming

Promote Clear Usage

- **Include all the words needed to avoid ambiguity** for a person reading code where the name is used.(為了人們閱讀程式碼時必須包含所有的字避免模糊不清範例如下)
  For example, consider a method that removes the element at a given position within a collection.(插入一個元素at位置)
  extension List {
- public mutating func remove(at position: Index) -> Element
- }
- employees.remove(at: x)
-

  If we were to omit the word at from the method signature, it could imply to the reader that the method searches for and removes an element equal to x, rather than using x to indicate the position of the element to remove.(如果沒有at會誤認移除x)
  employees.remove(x) // unclear: are we removing x?
-

- **Omit needless words.** Every word in a name should convey salient information at the use site.
  More words may be needed to clarify intent or disambiguate meaning, but those that are redundant with information the reader already possesses should be omitted. In particular, omit words that *merely repeat* type information.
  public mutating func removeElement(_ member: Element) -> Element?
-
- allViews.removeElement(cancelButton)
-

  In this case, the word Element adds nothing salient at the call site. This API would be better:
  public mutating func remove(_ member: Element) -> Element?

-
- allViews.remove(cancelButton) // clearer
-

  Occasionally, repeating type information is necessary to avoid ambiguity, but in general it is better to use a word that describes a parameter's *role* rather than its type. See the next item for details.

- **Name variables, parameters, and associated types according to their roles,** rather than their type constraints.
  var **string** = "Hello"
- protocol ViewController {
- associatedtype **View**Type : View
- }
- class ProductionLine {
- func restock(from **widgetFactory**: WidgetFactory)
- }
-

  Repurposing a type name in this way fails to optimize clarity and expressivity. Instead, strive to choose a name that expresses the entity's *role*.
  var **greeting** = "Hello"
- protocol ViewController {
- associatedtype **ContentView** : View
- }
- class ProductionLine {
- func restock(from **supplier**: WidgetFactory)
- }
-

  If an associated type is so tightly bound to its protocol constraint that the protocol name *is* the role, avoid collision by appending Type to the associated type name:
  protocol Sequence {
- associatedtype Iterator**Type** : Iterator
- }
-

- **Compensate for weak type information** to clarify a parameter's role. Especially when a parameter type is NSObject, Any, AnyObject, or a fundamental type such Int or String, type information and context at the point of use may not fully convey intent. In this example, the declaration

may be clear, but the use site is vague.
func add(_ observer: NSObject, for keyPath: String)

- grid.add(self, for: graphics) // vague

To restore clarity, **precede each weakly typed parameter with a noun describing its role**:
func add**Observer**(_ observer: NSObject, for**KeyPath** path: String)

- grid.addObserver(self, forKeyPath: graphics) // clear

Strive for Fluent Usage

- **Prefer method and function names that make use sites form grammatical English phrases.**
  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”

x.insert(y, position: z)

- x.subViews(color: y)
- x.nounCapitalize()

It is acceptable for fluency to degrade after the first argument or two when those arguments are not central to the call’s meaning:
AudioUnit.instantiate(

- with: description,
- **options: [.inProcess], completionHandler: stopProgressBar**)

- **Begin names of factory methods with “make”,** e.g. x.makeIterator().

- The first argument to **initializer and factory methods calls** should not form a phrase starting with the base name, e.g. x.makeWidget(cogCount: 47)
  For example, the first arguments to the these calls do not read as part of the same phrase as the base name:
  let foreground = **Color**(red: 32, green: 64, blue: 128)
- let newPart = **factory.makeWidget**(gears: 42, spindles: 14)
- let ref = **Link**(target: destination)

-

  In the following, the API author has tried to create grammatical continuity with the first argument.
  let foreground = **Color(havingRGBValuesRed: 32, green: 64, andBlue: 128)**
- let newPart = **factory.makeWidget(havingGearCount: 42, andSpindleCount: 14)**
- let ref = **Link(to: destination)**
-

  In practice, this guideline along with those for argument labels means the first argument will have a label unless the call is performing a value preserving type conversion.
  let rgbForeground = RGBColor(cmykForeground)
-

- **Name functions and methods according to their side-effects**
  - Those without side-effects should read as noun phrases, e.g. x.distance(to: y), i.successor().

  - Those with side-effects should read as imperative verb phrases, e.g., print(x), x.sort(), x.append(y).

  - **Name Mutating/nonmutating method pairs** consistently. A mutating method will often have a nonmutating variant with similar semantics, but that returns a new value rather than updating an instance in-place.
    - When the operation is **naturally described by a verb**, use the verb's imperative for the mutating method and apply the "ed" or "ing" suffix to name its nonmutating counterpart.

Mutating	Nonmutating
x.sort()	z = x.sorted()
x.append(y)	z = x.appending(y)

      - Prefer to name the nonmutating variant using the verb's past participle (usually appending "ed"):
        /// Reverses `self` in-place.
      - mutating func reverse()
      -
      - /// Returns a reversed copy of `self`.
      - func revers**ed**() -> Self
      - ...
      - x.reverse()
      - let y = x.reversed()
      -

- When adding “ed” is not grammatical because the verb has a direct object, name the nonmutating variant using the verb’s present participle, by appending “ing.”
  /// Strips all the newlines from `self`
- mutating func stripNewlines()
-
- /// Returns a copy of `self` with all the newlines stripped.
- func strip**ping**Newlines() -> String
- ...
- s.stripNewlines()
- let oneLine = t.strippingNewlines()
-

-

- When the operation is **naturally described by a noun**, use the noun for the nonmutating method and apply the “form” prefix to name its mutating counterpart.

▪ **Nonmutating**	▪ **Mutating**
▪ x = y.union(z)	▪ y.formUnion(z)
▪ j = c.successor(i)	▪ c.formSuccessor(&i)

- • **Uses of Boolean methods and properties should read as assertions about the receiver** when the use is nonmutating, e.g. x.isEmpty, line1.intersects(line2).

- **Protocols that describe** *what something is* **should read as nouns** (e.g. Collection).

- **Protocols that describe a** *capability* **should be named using the suffixes able, ible, or ing** (e.g. Equatable, ProgressReporting).

- The names of other **types, properties, variables, and constants should read as nouns.**

Use Terminology Well

Term of Art

noun - a word or phrase that has a precise, specialized meaning within a particular field or profession.

- **Avoid obscure terms** if a more common word conveys meaning just as well. Don’t say “epidermis” if “skin” will serve your purpose. Terms of art are an essential communication tool, but should only be used to capture crucial meaning that would otherwise be lost.

- **Stick to the established meaning** if you do use a term of art.
  The only reason to use a technical term rather than a more common word is that it *precisely* expresses something that would otherwise be ambiguous or unclear. Therefore, an API should use the term strictly in accordance with its accepted meaning.
  - **Don't surprise an expert**: anyone already familiar with the term will be surprised and probably angered if we appear to have invented a new meaning for it.
  - **Don't confuse a beginner**: anyone trying to learn the term is likely to do a web search and find its traditional meaning.
-

- **Avoid abbreviations.** Abbreviations, especially non-standard ones, are effectively terms-of-art, because understanding depends on correctly translating them into their non-abbreviated forms.
  The intended meaning for any abbreviation you use should be easily found by a web search.

- **Embrace precedent.** Don't optimize terms for the total beginner at the expense of conformance to existing culture.
  It is better to name a contiguous data structure Array than to use a simplified term such as List, even though a beginner might grasp of the meaning of List more easily. Arrays are fundamental in modern computing, so every programmer knows—or will soon learn—what an array is. Use a term that most programmers are familiar with, and their web searches and questions will be rewarded.
  Within a particular programming *domain*, such as mathematics, a widely precedented term such as sin(x) is preferable to an explanatory phrase such as verticalPositionOnUnitCircleAtOriginOfEndOfRadiusWithAngle(x).
  Note that in this case, precedent outweighs the guideline to avoid abbreviations: although the complete word is sine, "sin(*x*)" has been in common use among programmers for decades, and among mathematicians for centuries.

Conventions
General Conventions

- **Document the complexity of any computed property that is not O(1).**
  People often assume that property access involves no significant computation, because they have stored properties as a mental model. Be sure to alert them when that assumption may be violated.

- **Prefer methods and properties to free functions.** Free functions are

used only in special cases:

1. When there's no obvious self:
   min(x, y, z)
2.

3. When the function is an unconstrained generic:
   print(x)
4.

5. When function syntax is part of the established domain notation:
   sin(x)
6.

-

- **Follow case conventions.** Names of types and protocols are UpperCamelCase. Everything else is lowerCamelCase.
  Acronyms and initialisms that commonly appear as all upper case in American English should be uniformly up- or down-cased according to case conventions:
  var **utf8**Bytes: [**UTF8**.CodeUnit]
- var isRepresentableAs**ASCII** = true
- var user**SMTP**Server: Secure**SMTP**Server
-

  Other acronyms should be treated as ordinary words:
  var **radar**Detector: **Radar**Scanner
- var enjoys**Scuba**Diving = true
-

- **Methods can share a base name** when they share the same basic meaning or when they operate in distinct domains.
  For example, the following is encouraged, since the methods do essentially the same things:
  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 { ... }
-

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

  And since geometric types and collections are separate domains, this is also fine in the same program:
  extension Collection where Element : Equatable {
- /// Returns `true` iff `self` contains an element equal to
- /// `sought`.
- func **contains**(_ sought: Element) -> Bool { ... }
- }
-

  However, these index methods have different semantics, and should have been named differently:
  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 { ... }
- }
-

  Lastly, avoid “overloading on return type” because it causes ambiguities in the presence of type inference.
  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? { ... }
- }
-

Parameters

func move(from **start**: Point, to **end**: Point)

- **Choose parameter names to serve documentation**. Even though parameter names do not appear at a function or method’s point of use, they play an important explanatory role.
  Choose these names to make documentation easy to read. For example,

these names make documentation read naturally:
/// Return an `Array` containing the elements of `self`
- /// that satisfy `**predicate**`.
- func filter(_ **predicate**: (Element) -> Bool) -> [Generator.Element]
-
- /// Replace the given `**subRange**` of elements with `**newElements**`.
- mutating func replaceRange(_ **subRange**: Range, with **newElements**: [E])
-

These, however, make the documentation awkward and ungrammatical:
/// Return an `Array` containing the elements of `self`
- /// that satisfy `**includedInResult**`.
- func filter(_ **includedInResult**: (Element) -> Bool) -> [Generator.Element]
-
- /// Replace the **range of elements indicated by `r`** with
- /// the contents of `**with**`.
- mutating func replaceRange(_ **r**: Range, **with**: [E])
-

- **Take advantage of defaulted parameters** when it simplifies common uses. Any parameter with a single commonly-used value is a candidate for a default.
  Default arguments improve readability by hiding irrelevant information. For example:
  let order = lastName.compare(
- royalFamilyName**, options: [], range: nil, locale: nil**)
-

can become the much simpler:
let order = lastName.**compare(royalFamilyName)**
-

Default arguments are generally preferable to the use of method families, because they impose a lower cognitive burden on anyone trying to understand the API.
extension String {
- /// *...description...*
- public func compare(
- _ other: String, options: CompareOptions **= []**,
- range: Range? **= nil**, locale: Locale? **= nil**
-) -> Ordering
- }
-

The above may not be simple, but it is much simpler than:
extension String {

- /// ...*description 1*...
- public func **compare**(_ other: String) -> Ordering
- /// ...*description 2*...
- public func **compare**(_ other: String, options: CompareOptions) -> Ordering
- /// ...*description 3*...
- public func **compare**(
- _ other: String, options: CompareOptions, range: Range) -> Ordering
- /// ...*description 4*...
- public func **compare**(
- _ other: String, options: StringCompareOptions,
- range: Range, locale: Locale) -> Ordering
- }
-

Every member of a method family needs to be separately documented and understood by users. To decide among them, a user needs to understand all of them, and occasional surprising relationships—for example, foo(bar: nil) and foo() aren't always synonyms—make this a tedious process of ferreting out minor differences in mostly identical documentation. Using a single method with defaults provides a vastly superior programmer experience.

- **Prefer to locate parameters with defaults toward the end** of the parameter list. Parameters without defaults are usually more essential to the semantics of a method, and provide a stable initial pattern of use where methods are invoked.

Argument Labels

func move(**from** start: Point, **to** end: Point)
x.move(**from:** x, **to:** y)

- **Omit all labels when arguments can't be usefully distinguished**, e.g. min(number1, number2), zip(sequence1, sequence2).

- **In initializers that perform value preserving type conversions, omit the first argument label**, e.g. Int64(someUInt32)
The first argument should always be the source of the conversion.
extension String {
- // Convert `x` into its textual representation in the given radix
- init(_ x: BigInt, radix: Int = 10) ← Note the initial underscore
- }
-
- text = "The value is: "

- text += **String(veryLargeNumber)**
- text += " and in hexadecimal, it's"
- text += **String(veryLargeNumber, radix: 16)**
-

  In “narrowing” type conversions, though, a label that describes the narrowing is recommended.
  extension UInt32 {
- /// Creates an instance having the specified `value`.
- init(_ value: Int16) ← Widening, so no label
- /// 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)
- }
-

  A value preserving type conversion is a monomorphism, i.e. every difference in the value of the source results in a difference in the value of the result. For example, conversion from Int8 to Int64 is value preserving because every distinct Int8 value is converted to a distinct Int64 value. Conversion in the other direction, however, cannot be value preserving: Int64 has more possible values than can be represented in an Int8.
  Note: the ability to retrieve the original value has no bearing on whether a conversion is value preserving.

- **When the first argument forms part of a prepositional phrase, give it an argument label**. The argument label should normally begin at the preposition, e.g. x.removeBoxes(havingLength: 12).
  An exception arises when the first two arguments represent parts of a single abstraction.
  a.move(**toX:** b, **y:** c)
- a.fade(**fromRed:** b, **green:** c, **blue:** d)
-

  In such cases, begin the argument label *after* the preposition, to keep the abstraction clear.
  a.moveTo(**x:** b, **y:** c)
- a.fadeFrom(**red:** b, **green:** c, **blue:** d)
-

- **Otherwise, if the first argument forms part of a grammatical phrase, omit its label**, appending any preceding words to the base name, e.g.
  x.addSubview(y)
  This guideline implies that if the first argument *doesn't* form part of a grammatical phrase, it should have a label.
  view.dismiss(**animated:** false)
- let text = words.split(**maxSplits:** 12)
- let studentsByName = students.sorted(**isOrderedBefore:** Student.namePrecedes)
-

  Note that it's important that the phrase convey the correct meaning. The following would be grammatical but would express the wrong thing.
  view.dismiss(false) Don't dismiss? Dismiss a Bool?
- words.split(12) Split the number 12?
-

  Note also that arguments with default values can be omitted, and in that case do not form part of a grammatical phrase, so they should always have labels.

- **Label all other arguments**.

Special Instructions

- **Label tuple members and name closure parameters** where they appear in your API.
  These names have explanatory power, can be referenced from documentation comments, and provide expressive access to tuple members.
  /// 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.
- ///
- /// - Returns:
- /// - **reallocated**: `true` iff a new block of memory
- /// was allocated.
- /// - **capacityChanged**: `true` iff `capacity` was updated.
- mutating func ensureUniqueStorage(
- minimumCapacity requestedCapacity: Int,
- allocate: (_ **byteCount**: Int) -> UnsafePointer<Void>
-) -> (**reallocated:** Bool, **capacityChanged:** Bool)
-

Names used for closure parameters should be chosen like parameter names for top-level functions. Labels for closure arguments that appear at the call site are not supported.

- **Take extra care with unconstrained polymorphism** (e.g. Any, AnyObject, and unconstrained generic parameters) to avoid ambiguities in overload sets.
  For example, consider this overload set:
  struct Array {
- /// Inserts `newElement` at `self.endIndex`.
- public mutating func append(_ newElement: Element)
-
- /// Inserts the contents of `newElements`, in order, at
- /// `self.endIndex`.
- public mutating func append(_ newElements: S)
- where S.Generator.Element == Element
- }
-

  These methods form a semantic family, and the argument types appear at first to be sharply distinct. However, when Element is Any, a single element can have the same type as a sequence of elements.
  var values: [Any] = [1, "a"]
- values.append([2, 3, 4]) // [1, "a", [2, 3, 4]] or [1, "a", 2, 3, 4]?
-

  To eliminate the ambiguity, name the second overload more explicitly.
  struct Array {
- /// Inserts `newElement` at `self.endIndex`.
- public mutating func append(_ newElement: Element)
-
- /// Inserts the contents of `newElements`, in order, at
- /// `self.endIndex`.
- public mutating func append(**contentsOf** newElements: S)
- where S.Generator.Element == Element
- }
-

  Notice how the new name better matches the documentation comment. In this case, the act of writing the documentation comment actually brought the issue to the API author's attention.