orphan:

Swift Standard Library API Design Guide

> **Note**
>
> This guide documents *current practice* in the Swift standard library as of April 2015. API conventions are expected to evolve in the near future to better harmonize with Cocoa.

The current Swift Standard Library API conventions start with the Cocoa guidelines as discussed on these two wiki pages: [API Guidelines, Properties], and in this WWDC Presentation. Below, we list where and how the standard library's API conventions differ from those of Cocoa

Differences

Points in this section clash in one way or other with the Cocoa guidelines.

The First Parameter

- The first parameter to a function, method, or initializer typically does not have an argument label:

```
alligators.insert(fred)           // yes
if alligators.contains(george) {  // yes
  return
}

alligators.insert(element: fred)           // no
if alligators.contains(element: george) {  // no
  return
}
```

- Typically, no suffix is added to a function or method's base name in order to serve the same purpose as a label:

```
alligators.insertElement(fred)           // no
if alligators.containsElement(george) {  // no
  return
}
```

- A preposition is added to the end of a function name if the role of the first parameter would otherwise be unclear:

```
// origin of measurement is aPosition
aPosition.distanceTo(otherPosition)

// we're not "indexing x"
if let position = aSet.indexOf(x) { ... }
```

- Argument labels are used on first parameters to denote special cases:

```
// Normal case: result has same value as argument (traps on overflow)
Int(aUInt)

// Special: interprets the sign bit as a high bit, changes value
Int(bitPattern: aUInt)

// Special: keeps only the bits that fit, losing information
Int32(truncatingBitPattern: anInt64)
```

Subsequent Parameters

- Argument labels are chosen to clarify the *role* of an argument, rather than its type:

```
x.replaceSubrange(r, with: someElements)

p.initializeFrom(q, count: n)
```

- Second and later parameters are always labeled except in cases where there's no useful distinction of roles:

```
swap(&a, &b)                                                    // OK

let topOfPicture = min(topOfSquare, topOfTriangle, topOfCircle) // OK
```

Other Differences

- We don't use namespace prefixes such as "`NS`", relying instead on the language's own facilities.
- Names of types, protocols and enum cases are `UpperCamelCase`. Everything else is `lowerCamelCase`. When an initialism appears, it is **uniformly upper- or lower-cased to fit the pattern**:

```
let v: String.UTF16View = s.utf16
```

- Protocol names end in `Type`, `able`, or `ible`. Other type names do not.

Additional Conventions

Points in this section place additional constraints on the standard library, but are compatible with the Cocoa guidelines.

- We document the complexity of operations using big-O notation.
- In API design, when deciding between a nullary function and a property for a specific operation, arguments based on performance characteristics and complexity of operations are not considered. Reading and writing properties can have any complexity.
- We prefer methods and properties to free functions. Free functions are used when there's no obvious `self`

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

  when the function is an unconstrained generic

  ```
  print(x)
  ```

  and when function syntax is part of the domain notation

  ```
  -sin(x)
  ```

- Type conversions use initialization syntax whenever possible, with the source of the conversion being the first argument:

  ```
  let s0 = String(anInt)            // yes
  let s1 = String(anInt, radix: 2)  // yes
  let s1 = anInt.toString()         // no
  ```

  The exception is when the type conversion is part of a protocol:

  ```
  protocol IntConvertible {
    func toInt() -> Int // OK
  }
  ```

- Even unlabeled parameter names should be meaningful as they'll be referred to in comments and visible in "generated headers" (cmd-click in Xcode):

  ```
  /// Reserve enough space to store `minimumCapacity` elements.
  ///
  /// PostCondition: `capacity >= minimumCapacity` and the array has
  /// mutable contiguous storage.
  ///
  /// Complexity: O(`count`)
  mutating func reserveCapacity(_ minimumCapacity: Int)
  ```

- Type parameter names of generic types describe the role of the parameter, e.g.

  ```
  struct Dictionary<Key, Value> { // not Dictionary<K, V>
  ```

Acceptable Short or Non-Descriptive Names

- Type parameter names of generic functions may be single characters:

  ```
  func swap<T>(lhs: inout T, rhs: inout T)
  ```

- `lhs` and `rhs` are acceptable names for binary operator or symmetric binary function parameters:

  ```
  func + (lhs: Int, rhs: Int) -> Int

  func swap<T>(lhs: inout T, rhs: inout T)
  ```

- `body` is an acceptable name for a trailing closure argument when the resulting construct is supposed to act like a language extension and is likely to have side-effects:

  ```
  func map<U>(_ transformation: T->U) -> [U] // not this one

  func forEach<S: SequenceType>(_ body: (S.Iterator.Element) -> ())
  ```

Prefixes and Suffixes

- `Any` is used as a prefix to denote "type erasure," e.g. `AnySequence<T>` wraps any sequence with element type `T`, conforms to `SequenceType` itself, and forwards all operations to the wrapped sequence. When handling the wrapper, the specific type of the wrapped sequence is fully hidden.
- `Custom` is used as a prefix for special protocols that will always be dynamically checked for at runtime and don't make good generic constraints, e.g. `CustomStringConvertible`.

- `InPlace` is used as a suffix to denote the mutating member of a pair of related methods:

  ```
  extension Set {
    func union(_ other: Set) -> Set
    mutating func unionInPlace(_ other: Set)
  }
  ```

- `with` is used as a prefix to denote a function that executes a closure within a context, such as a guaranteed lifetime:

  ```
  s.withCString {
    let fd = fopen($0)
    ...
  } // don't use that pointer after the closing brace
  ```

- `Pointer` is used as a suffix to denote a non-class type that acts like a reference, c.f. `ManagedBufferPointer`
- `unsafe` or `Unsafe` is *always* used as a prefix when a function or type allows the user to violate memory or type safety, except on methods of types whose names begin with `Unsafe`, where the type name is assumed to convey that.
- `C` is used as a prefix to denote types corresponding to C language types, e.g. `CChar`.