



# API Design Guideline

---

## Fundamentals(기본 사항)

- 사용 지점에서의 명확성이 가장 중요한 목표입니다.

메서드와 속성과 같은 개체는 한 번만 선언되지만 반복해서 사용됩니다.

API를 사용이 명확하고 간결하게 만들도록 설계하세요.

설계를 평가할 때 선언만 읽는 것은 거의 충분하지 않습니다.

항상 사용 사례를 검토하여 컨텍스트(문맥)에서 명확해 보이는지 확인하세요.

- 명확성이 간결함보다 더 중요합니다.

스위프트 코드는 간결할 수 있지만, 가능한 가장 작은 코드를 가능하게 하는 것은 목표가 아닙니다.

스위프트 코드에서 간결함이 발생하면, 이는 강력한 타입 시스템과 보일러플레이트\*(상용 문)를 자연스럽게 줄이는 기능의 부작용입니다.

- 모든 선언에 대해 문서 주석을 작성하세요.

문서 작성을 통해 얻은 통찰력은 설계에 큰 영향을 미칠 수 있으므로 미루지 마세요.



API의 기능을 간단함 용어로 설명하는 데 어려움이 있다면 잘못된 API를 설계했을 수 있습니다.

---

## More Detail

- 스위프트의 **Markdown 통용어(dialect of Markdown)**를 사용합니다.
- 선언된 개체(entity)를 설명하는 **요약으로 시작**하세요. **API**는 선언과 요약을 통해 완전히 이해되는 경우가 많습니다.

```
/// Return a "view" of 'self' containing the same elements in
/// reverse order.
/// 동일한 요소를 역순으로 포함하는 'self'의 "view"를 반환합니다.
func reversed() -> ReverseCollection
```

- More detail
  - **요약에 초점을 맞추세요.** 요약은 매우 중요한 부분입니다. 많은 우수한 코드 주석은 훌륭한 요약문을 가지고 있습니다
  - 가능하면 **한 개의 절을 사용하고**, 마침표로 끝내세요. 완전한 문장을 사용하지 마세요.
  - **함수 또는 메소드가 어떤 일을 하는지, 어떤 것을 반환하는지 설명하고**, null 효과와 Void 반환은 설명을 생략하세요

```
/// Inserts 'newHead' at the beginning of `self`.
/// `self` 시작 부분에 'newHead'를 삽입합니다.
mutating func prepend(_ newHead: Int)

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

/// Removes and returns the first element of `self` if non-empty;
/// returns `nil` otherwise.
/// 비어 있지 않으면 `self`의 첫 번째 요소를 제거하고 반환합니다. 그렇지 않으면 `nil`을 반환합니다.
mutating func popFirst() -> Element?
```



**NOTE :** 자주 사용되지는 않지만, popFirst의 경우처럼 세미콜론을 사용해 여러 절로 이루어진 요약문을 작성할 수도 있습니다.

- **subscript가 어떤 것에 접근하는지 설명합니다.**

```
/// Accesses the `index`th element.
/// `인덱스`번째 요소에 액세스(접근)합니다.
```

```
subscript(index: Int) -> Element { get set }
```

◦ 이니셜라이저가 생성하는 것을 설명하십시오.

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

◦ 그 외의 경우, 선언된 개체가 무엇인지 설명합니다.

```
/// A collection that supports equally efficient insertion/removal
/// at any position.
/// 어떤 위치에서든 똑같이 효율적으로 삽입/제거할 수 있는 컬렉션.
struct List {

    /// The element at the beginning of `self`, or `nil` if self is
    /// empty.
    /// `self`의 첫 번째 요소, 또는 self가 비어있는 경우 `nil`입니다.
    var first: Element?

    ...
}
```

- 경우에 따라, 여러 절과 글 머리 기호를 사용할 수 있습니다. 공백 줄로 절을 나누고 완전한 문장을 사용합니다.

```
/// 표준 출력에 `items` 각 요소의 텍스트 표현을 작성합니다. ← 요약
///                                     ← 빈 줄
/// 각 요소인 `x`의 텍스트 표현은 `String(x)` 표현식으로 ← 추가 설명
/// 제공됩니다.
///
/// - 매개변수 separator: 요소 사이에 출력되는 텍스트      }
/// - 매개변수 terminator: 끝부분에 출력되는 텍스트      } 매개변수 부분
///
/// - 주의: 끝부분에 줄 바꿈을 출력하지 않으려면      }
///   `terminator: ""`를 전달하세요.                  |
///                                                     } 기타 참고사항
/// - 참조: `CustomDebugStringConvertible`,            |
///   `CustomStringConvertible`, `debugPrint`.         |
public func print(
    items: Any..., separator: String = " ", terminator: String = "\n")
```

- more detail
  - 요약문 외에 추가 정보를 제공할 때에는 많은 사람들이 이해할 수 있게 **symbol 문서 마크업 요소들을 사용하세요.**
  - **symbol 커맨드 구문을 익히고 활용하세요.** Xcode와 같은 개발 도구는 다음 키워드로 시작하는 글 머리 기호 (예 - Note)를 특별하게 취급합니다:

<u>Attention</u>	<u>Author</u>	<u>Authors</u>	<u>Bug</u>
<u>Complexity</u>	<u>Copyright</u>	<u>Date</u>	<u>Experiment</u>
<u>Important</u>	<u>Invariant</u>	<u>Note</u>	<u>Parameter</u>
<u>Requires</u>	<u>Returns</u>	<u>SeeAlso</u>	<u>Since</u>
<u>Throws</u>	<u>Todo</u>	<u>Version</u>	<u>Warning</u>

## 이름 지정

### 명확한 사용 활성화하기(Promote Clear Usage)

- 이름이 사용된 코드를 읽는 사람을 위해 **모호성을 피하고자 필요한 모든 단어를 포함합니다.**

예를 들어, 컬렉션 내에 주어진 위치에 있는 요소를 제거하는 메서드를 고려합니다.

✓

```
extension List {
  public mutating func remove(at position: Index) -> Element
}
employees.remove(at:x)
```

메소드 서명에서 **at** 단어를 생략한다면, 삭제하려는 요소의 위치를 나타내기 위해 **x**를 사용하는 것보다는 **x**와 같은 요소를 검색하고 제거한다고 독자에게 암시할 수 있습니다.

✗

```
employees.remove(x) // 불분명: x를 제거하는가?
```

- **불필요한 단어는 생략합니다.** 이름 내 모든 단어는 사용 장소에서 두드러진 정보를 전달해야 합니다.

명확한 의도 또는 애매하지 않은 의미를 위해 더 많은 단어가 필요할 수 있지만, 이는 독자가 이미 알고 있어 생략한 불필요한 정보입니다. 특히, 단순히 타입 정보를 반복하는 단어는 생략합니다.



```
public mutating func removeElement(member: Element) -> Element?

allViews.removeElement(cancelButton)
```

이 경우엔, **Element** 단어는 호출 장소에서 두드러지는 것을 추가하지 않습니다. 이 API가 좀 더 낫습니다:



```
public mutating func remove(member: Element) -> Element?

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

때때로, 반복 타입 정보는 모호성을 피하기 위해 필요하지만, 일반적으로 타입보다는 매개 변수의 역할을 설명하는 단어 사용이 더 낫습니다. 자세한 내용은 다음 항목을 참조하세요.

- **역할에 따라 이름 변수, 매개 변수, 연관 타입은 타입 제약사항보다 낫습니다.**
- more detail



```
var string = "Hello"
protocol ViewController {
    associatedtype ViewType : View
}
class ProductionLine {
    func restock(from widgetFactory: WidgetFactory)
}
```

이러한 방법으로 타입 이름 용도를 변경하는 것은 명확성과 표현력을 최적화하지 못합니다. 대신, 개체의 역할을 표현하는 이름을 선택하도록 노력합니다.

```
✓
var greeting = "Hello"
protocol ViewController {
    associatedtype ContentView: View
}
class ProductionLine {
    func restock(from supplier: WidgetFactory)
}
```

연관 타입은 프로토콜 이름이 역할인 프로토콜 제약사항으로 매우 밀접하게 결합되었다면, 연관 타입 이름에 **Type** 을 붙여 충돌을 피합니다.

```
protocol Sequence {
    associatedtype IteratorType: Iterator
}
```

- 매개변수의 역할을 명확하도록 **약 타입 정보를 보정합니다.**
- more detail

특히 매개변수 타입이 **NSObject**, **Any**, **AnyObject** 이거나 **Int** 또는 **String** 같은 기본 타입일 때, 사용 시점에 타입 정보와 상황은 의도를 완전히 전달할 수 없을 수 있습니다.

이 예제에서, 선언이 명확할 수 있지만, 사용하는 곳이 막연합니다.

```
✗
func add(observer: NSObject, for keyPath: String)

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

명확성을 복원하려면, 각각의 약 타입 매개변수와 매개변수 역할을 설명하는 명사를 앞으로 보냅니다:

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

## 유창한 사용을 하도록 노력하기(Strive for Fluent Usage)

- 사용하는 곳에 메소드와 함수 이름을 문법상 영어 문구 형태로 선호합니다.



```
x.insert(y, at: z) "x에서 z에다 y를 삽입"  
x.subViews(havingColor: y) "색상 y를 갖는 x의 subviews"  
x.capitalizingNouns() "x에 명사를 대문자화"
```



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

첫 번째 또는 두 번째 인자가 호출 의미에서 중심이 아닐 때 유창하도록 인자 뒤를 떨어뜨리는 것을 허용합니다:

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

- “make”로 팩토리 메소드 이름이 시작합니다. e.g x.makeIterator()
- **이니셜라이저와 팩토리 메소드 호출**은 첫 번째 인자를 포함하지 않는 문구로 구성해야 합니다. e.g x.makeWidget(cogCount: 47)
- more detail

예를 들어, 이러한 호출 때문에 내포된 문구는 첫 번째 인자를 포함하지 않습니다:



```
let foreground = Color(red: 32, green: 64, blue: 128)
let newPart = factory.makeWidget(gears: 42, spindles: 12)
```

다음에서, API 저자는 첫 번째 인자와 문법의 연속성을 만드는데 노력하고 있습니다.



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

실제로, 이 가이드라인은 인자 레이블과 더불어 full-width 타입 변환을 수행하는 호출이 아니면 첫 번째 인자는 레이블을 가짐을 의미합니다.

```
let rgbForeground = RGBColor(cmkForeground)
```

- **사이드 이펙트에 따라 함수와 메소드 이름을 지정합니다.**

- 사이드 이펙트가 없는 것은 명사구로 읽어야 합니다. e.g, x.distance(to: y), i.successor()
- 사이드 이펙트가 있는 것은 반드시 동사 구문으로 읽어야 합니다. e.g, print(x), x.sort(), x.append(y)
- **Mutating/nonmutating 메소드 쌍을 일관되게 이름을 지정합니다.**  
mutating 메소드는 종종 비슷한 의미가 있는 nonmutating variant 있지만, in-place를 갱신하는 것보다 새로운 값을 반환하는 것이 낫습니다.
  - 연산자는 **당연히 동사로 설명될 때**, mutating 메소드는 동사를 반드시 사용하고 nonmutating 메소드 이름을 지정하기 위해 “ed”나 “ing” 접미사를 적용합니다.



Mutating	Nonmutating
<code>x.sort()</code>	<code>z = x.sorted()</code>
<code>x.append(y)</code>	<code>z = x.appending(y)</code>

◦ more detail

- 동사의 과거분사(보통은 “ed”를 붙임)를 사용하여 nonmutating variant 이름을 지정  
을 선호합니다:

```
/// `self` in-place를 뒤집음.
mutating func reverse()

/// 뒤집혀진 `self`의 사본을 반환
func reversed() -> Self
...
x.reverse()
let y = x.reversed()
```

- 동사가 목적어를 가져 추가한 “ed”가 문법적이지 않을 때, 동사의 현재 분사를 사  
용하여 nonmutating variant에 “ing”을 붙여 이름을 정합니다.

```
/// `self`에서 모든 줄바꿈을 떼어냄.
mutating func stripNewlines()

/// 모든 줄바꿈을 떼어낸 `self`의 사본을 반환.
func strippingNewlines() -> String
...
s.stripNewlines()
let oneLine = t.strippingNewlines()
```

- 연산자는 당연히 명사로 설명될 때, nonmutating 메소드는 명사를 사용하고  
mutating 메소드는 “form” 접미사를 적용합니다.

Nonmutating	Mutating
<code>x = y.union(z)</code>	<code>y.formUnion(z)</code>
<code>j = c.successor(i)</code>	<code>c.formSuccessor(&amp;i)</code>

- **Boolean 메소드와 속성 사용은 nonmutating일 때 수신자에 관한 주장으로 해석해야 합니다.**  
e.g, `x.isEmpty`, `line1.intersects(line2)`
- **무언가를 설명하는 프로토콜은 명사로 읽어야 합니다.**
- **능력을 설명하는 프로토콜은 able, ible 또는 ing 접미사를 사용하여 이름을 지정해야 합니다.**  
e.g, `Equatable`, `ProgressReporting`
- **다른 타입, 속성, 변수 그리고 상수 의 이름은 명사로 읽어야 합니다.**

## 전문 용어를 잘 사용하기(Use Terminology Well)



Term of Art 명사 - 특정 영역이나 직업 내 정확하고 전문적인 의미가 있는 단어나 구.

- 더욱 일반적인 단어가 의미를 잘 전달할 수 있다면 **애매한 용어는 피합니다.**  
"피부(skin)"가 목적을 전달할 수 있다면 "표피(epidermis)를 언급하지 않습니다."  
Terms of art는 주요한 커뮤니케이션 도구이지만, 손실될 중요한 의미를 포착하는 데 사용되어야 합니다.
- Term of Art를 사용한다면 **기존의 의미에 충실합니다**
- more detail  
더 일반적인 단어보다 기술 용어를 사용하는 유일한 이유는 기술 용어가 정확하게 표현하지만, 반면 모호하거나 불분명합니다. 그러므로 API는 받아들여지는 의미에 따라 엄격하게 용어를 사용해야 합니다.

- **전문가를 놀라게 하지 마세요** : 우리가 기존 용어에 새로운 의미를 고안한 것처럼 보인다면 이미 용어에 익숙한 사람은 놀라고 성을 낼 것입니다.
- **초보자를 혼란스럽게 하지 마세요** : 용어를 배우려고 하는 사람은 웹 검색을 하고 전통적인 의미를 찾을 가능성이 있습니다.
- **약어를 피하세요.** 특히 표준이 아닌 약어는 효과적으로 Term of Art이며, 이해도에 따라 약어를 비 축약 형태로 번역합니다.



사용하는 약어에 의도된 의미는 웹 사이트에서 쉽게 찾을 수 있습니다.

- **선례를 받아들이세요.** 기존 문화에 적합성 비용에 있어 모든 초보자를 위해 용어를 최적화하지 마세요

- more detail

연속 데이터 구조 이름은 List처럼 간단한 용어 사용보다 Array가 낫습니다.

비록 초보자도 쉽게 List의 의미를 파악할 수 있지만 말이죠.

현재 컴퓨팅에서 Array는 기초이고, 모든 프로그래머는 알고 있거나 array가 무엇인지 곧 공부할 것 입니다.

대부분 프로그래머가 잘 알고 있는 용어를 사용하고, 웹 검색과 질문으로 보상받을 것 입니다.

수학처럼 특정 프로그래밍 도메인 내에서

verticalPositionOnUnitCircleAtOriginOfEntOfRadiusWithAngle(x) 처럼 설명 문구보다 sin(x)처럼 넓게 사용되고 있는 선례 용어를 선호합니다.

이 경우, 선례는 약어를 피하기 위한 가이드라인보다 중요합니다: 전체 단어가 sine이지만, “sin(x)”은 수십 년간 프로그래머 사이에서, 수백 년간 수학자 사이에서 흔하게 사용되었습니다.

## 규칙(Conventions)

### 일반적인 규칙(General Conventions)

- **O(1)이 아닌 계산 속성의 복잡성을 문서로 만듭니다.** 사람들은 속성 접근이 중요한 계산을 수반하지 않는다고 종종 가정하는데, 이는 저장 속성은 심성모형(mental model)을 갖기 때 문입니다.

가정이 어긋날 수 있을 때 경고해야 합니다.

- **자유 함수(free functions)보다 메소드와 속성을 선호합니다.** 자유 함수는 특별한 경우에 만 사용됩니다.

- more detail

- 1. 명백한 self가 없을 때:

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

- 2. 함수가 제약되지 않은 제네릭일 때:

```
print(x)unconstrained
```

- 3. 함수 구문이 기존 도메인 표기의 일부일 때:

```
sin(x)
```

- **case convention을 따릅니다.** 타입과 프로토콜의 이름은 **UpperCamelCase**입니다. 그 외 모든 것은 lowerCamelCase 입니다.

- more detail

- 미국식 영어에서는 모두 대문자로 흔하게 표시하는 두문자어(Acronym과 initialism)는 case convention을 따라 대문자이거나 소문자로 일관되어야 합니다.

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

- 다른 약어(acronym)는 일반적인 단어로 다뤄져야 합니다:

```
var raderDetector: RaderScanner
var enjoysScubaDiving = true
```

- 메소드는 같은 기본 의미를 공유할 때 또는 구분된 도메인에서 수행할 때 **기본 이름을 공유** 할 수 있습니다.
- more detail

다음은 권장하는 예제로, 메소드는 본질적으로 같은 것을 수행할 수 있기 때문입니다:

```
✓
extension Shape {
  /// `other`가 `self` 지역 내에 있는 경우 `true`를 반환.
  func contains(other: Point) -> Bool { ... }

  /// `other`가 `self` 지역 내에 완전히 있는 경우 `true`를 반환.
  func contains(other: Shape) -> Bool { ... }

  /// `other`가 `self` 지역 내에 있는 경우 `true`를 반환.
  func contains(other: LineSegment) -> Bool { ... }
}
```

그리고 기하학 타입 또는 컬렉션은 별도의 도메인이기 때문에, 같은 프로그램 내에서 괜찮습니다:


```
✓
extension Collection where Element: Equatable {
  /// 'self'가 'sought'와 같은 요소를 포함하는 경우 `true`를 반환.
  func contains(sought: Element) -> Bool { ... }
}
```

그러나 **index** 메소드가 다른 의미가 있고, 다른 이름으로 되어 있어야 합니다:

```
✗
extension Database {
  /// Database의 검색 index를 rebuild 함
  func index() { ... }

  /// 주어진 table에서 `n`번째 row를 반환.
  func index(n: Int, inTable: TableID) -> TableRow { ... }
}
```

마지막으로, “반환 타입에 오버로딩”을 피하세요. 이는 타입 추론이 있을 때 모호성을 일으키기 때문입니다.




```
extension Box {  
    /// `self`에 저장된 `Int`를 반환하고, 없으면 nil을 반환.  
    func value() -> Int? { ... }  
  
    /// `self`에 저장된 `String`을 반환하고, 없으면 nil을 반환.  
    func value() -> String? { ... }  
}
```

## 매개 변수(Parameters)

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


- **문서를 제공하는 매개 변수 이름을 선택합니다.** 매개 변수 이름이 함수나 메소드의 사용 시점에 드러나지 않더라도, 중요한 설명하는 역할을 수행합니다.
- more detail

문서를 쉽게 읽을 수 있도록 매개 변수 이름을 선택합니다. 예를 들어, 매개 변수 이름은 문서를 자연스럽게 읽을 수 있도록 만듭니다:



```
/// `predicate`를 만족하는 `self`의 요소를 포함하는 `Array`를 반환.  
func filter(_ predicate: (Element) -> Bool) -> [Generator.Element]  
  
/// 주어진 요소의 `subRange`를 `newElements`로 치환.  
mutating func replaceRange(_ subRange: Range, with newElements: [E])
```

그러나 매개 변수는 어색하고 문법에 맞지 않는 문서를 만듭니다:



```
/// `includedInResult`를 만족하는 `self`의 요소를 포함하는 `Array`를 반환.  
func filter(_ includedInResult: (Element) -> Bool) -> [Generator.Element]  
  
/// `r`로 표시된 요소의 range를 `with`의 콘텐츠로 치환.  
mutating func replaceRange(_ r: Range, with: [E])
```

- **기본 매개 변수**는 일반적인 사용을 단순화할 때 **이용합니다**. 하나만 흔히 사용하는 값을 가진 매개 변수는 기본값 후보입니다.
- more detail

기본 인자는 관련 없는 정보를 숨김으로써 가독성을 향상합니다 예:



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

더 단순하게 될 수 있습니다:



```
let order = lastName.compare(royalFamilyName)
```

기본 인자는 일반적으로 메소드 집합(family) 사용보다 더 나운데, API를 이해하려고 하는 사람에게 낮은 인지 부담을 지우기 때문입니다.



```
extension String {
    /// ..description...
    public func compare(
        other: String, options: CompareOptions = [],
        range: Range? = nil, locale: Locale? = nil
    ) -> Ordering
}
```

위는 간단하지 않은 수 있지만 아래보다 더 간단합니다.



```
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
}

```

메소드 집합의 모든 구성원은 유저마다 별도의 설명과 이해가 필요합니다.

이들 사이에서 결정하려면, 메소드 집합 전부를 이해할 필요가 있습니다. 그리고 가끔 놀라운 관계-예제로, **foo(bar: nil)와 foo()**은 항상 동의어가 아니다-는 대부분 같은 문서에서 작은 타이틀 찾는 지루한 과정을 만듭니다.

기본으로 단일 메소드 사용은 훨씬 뛰어난 프로그래머 경험을 제공합니다.

- 매개 변수 목록의 **마지막 방향으로 매개 변수와 기본값의 위치를 선호합니다**. 기본값이 없는 매개 변수는 보통 메소드의 의미에 더 중요하고, 메소드가 호출된 곳에서 안정적인 사용의 초기 패턴을 제공합니다.

## 인자 레이블(Argument Labels)

```

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

```

- 인자가 유용하게 구별될 수 없을 때 모든 레이블은 생략합니다. e.g min(number1, number2), zip(sequence1, sequence2)
- 이니셜라이저에서 full-width 타입 변환을 수행하며, 첫 번째 인자 레이블은 생략합니다. e.g. Int64(someUInt32)
- more detail

첫 번째 인자는 항상 변환의 출처가 될 것 입니다.

```

extension String {
  // radix에서 `x`가 텍스트 표시로 변경
  init(_ x: BigInt, radix: Int = 10) ← 처음 밑줄을 주의
}

text = "The value is: "
text += String(veryLargeNumber)
text += " and in hexadecimal, it's"
text += String(veryLargeNumber, radix: 16)

```


“narrowing” 타입 변환에서 narrowing을 설명하는 label을 추천합니다.



```
extension UInt32 {
  /// 저장된 `value`를 가지는 인스턴스를 생성.
  init(_ value: Int16) ← Widening이고 label이 아님
  /// `source`의 가장 낮은 32bits를 가지는 인스턴스를 생성.
  init(truncating source: UInt64)
  /// `valueToApproximate`에 가장 가깝게 표시할 수 있는 추정
  /// 가지는 인스턴스를 생성
  init(saturating valueToApproximate: UInt64)
}
```


- 첫 번째 인자가 전치사구의 부분 형태일 때, 인자 레이블이 주어집니다. 인자 레이블은 보통 전치사로 시작해야 합니다.
- more detail

처음 두 개의 인자는 하나의 추상화 일부를 나타낼 때 예외가 나타납니다.



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

이런 경우, 전치사 뒤에 인자 레이블이 시작하고 명확한 추상화를 유지합니다.




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

- 반면, 첫 번째 인자는 문법적인 구문 일부 형태일 때, 레이블을 생략하고, 기본 이름에 선행 단어를 추가합니다.  
e.g. x.addSubview(y)

- more detail

이 가이드라인은 첫 번째 인자가 문법적인 구문 일부를 형성하지 않으면 레이블을 가져야 함을 의미합니다.



```
view.dismiss(animated: false)
let text = words.split(maxSplits: 12)
let studentsByName = students.sorted(isOrderedBefore: Student.namePrecedes)
```

구문은 올바른 의미를 전달하는 것이 중요함을 주의합니다. 다음은 문법적이 될 것이지만 잘못 표현하는 것입니다.



```
view.dismiss(false)    dismiss 안하나요? Bool을 dismiss하나요?  
words.split(12)        숫자 12를 분리하나요?
```

기본값이 있는 인자 또는 생략할 수 있음을 유의하고, 이경우에 문법적인 구문의 일부를 형성하지 않습니다.

그래서 항상 레이블을 가지고 있어야 합니다.

- 다른 모든 인자는 레이블을 지정합니다.

## 특별 지침(Special Instructions)

- 클로저 매개 변수와 튜플 멤버가 API에서 나타난 곳에 레이블을 지정합니다.
- more detail

These names have이 이름은 설득력이 있고, 문서 주석에서 참조될 수 있으며, 튜플 멤버에 풍부한 접근을 제공합니다.

```
/// 적어도 `requestedCapacity` 요소에 대해 uniquely-referenced storage를  
/// 유지하고 있음을 보증합니다.  
///  
/// 더 많은 storage가 필요하다면, 할당하기 위해 최대로 정렬된 바이트 수와  
/// 같은 `byteCount`를 사용하여 `allocate`를 호출합니다.  
///  
/// - 반환:  
///   - reallocated: 메모리의 새로운 block이 할당되었다면  
///     `true`를 반환.  
///   - capacityChanged: `capability`가 업데이트 되었다면 `true`를 반환.  
mutating func ensureUniqueStorage(  
    minimumCapacity requestedCapacity: Int,  
    allocate: (byteCount: Int) -> UnsafePointer<Void>  
) -> (reallocated: Bool, capacityChanged: Bool)
```


클로저에서 사용될 때 기술적으로 인자 레이블이지만, 레이블을 선택해야 하고 매개 변수 이름이었던 것처럼 문서에서 사용해야 합니다.

함수 본체에서 클로저 호출은 첫 번째 인자를 포함하지 않는 기본 이름에서 구문을 시작하는 함수를 일관되게 읽을 수 있습니다:

```
allocate(byteCount: newCount * elementSize)
```

- 오버로드(overload) 세트에서 모호성을 피하도록 **제약되지 않은 다형성에 주의를 더 기울여야 합니다**(e.g Any, AnyObject 그리고 제약되지 않은 제네릭 매개 변수)
- more detail


예를 들어, 오버로드 세트를 고려합니다:

```

struct Array {
  /// `self.endIndex`에 `newElement`를 삽입.
  public mutating func append(newElement: Element)

  /// 순서대로 `self.endIndex`에 `newElements` 콘텐츠를 삽입.
  public mutating func append<
    S : SequenceType where S.Iterator.Element == Element
  >(newElements: S)
}
```

이들 메소드는 의미론 집합을 구성하고, 인자 타입은 확연히 구별되도록 처음에 나타냅니다.

그러나 Element가 Any일 때, 하나의 요소의 시퀀스로서 같은 타입을 가질 수 있습니다.

```

var values: [Any] = [1, "a"]
values.append([2, 3, 4]) // [1, "a", [2, 3, 4]] or [1, "a", 2, 3, 4]?
```

모호성을 제거하려면, 더 명시적으로 두 번째 오버로드 이름을 지정합니다.

```
struct Array {
  /// `self.endIndex`에 `newElement`를 삽입.
  public mutating func append(newElement: Element)

  /// 순서대로 `self.endIndex`에 `newElements` 콘텐츠를 삽입
  public mutating func append<
    S : SequenceType where S.Iterator.Element == Element
  >(newElements: S)
```

```
>(contentsOf newElements: S)  
}
```

새로운 이름이 문서 주석과 더 일치하는 방법을 알 수 있습니다. 이 경우에는, 문서 주석을 작성하는 행위는 실제로 API 저자의 주의를 이슈로 가져옵니다.