

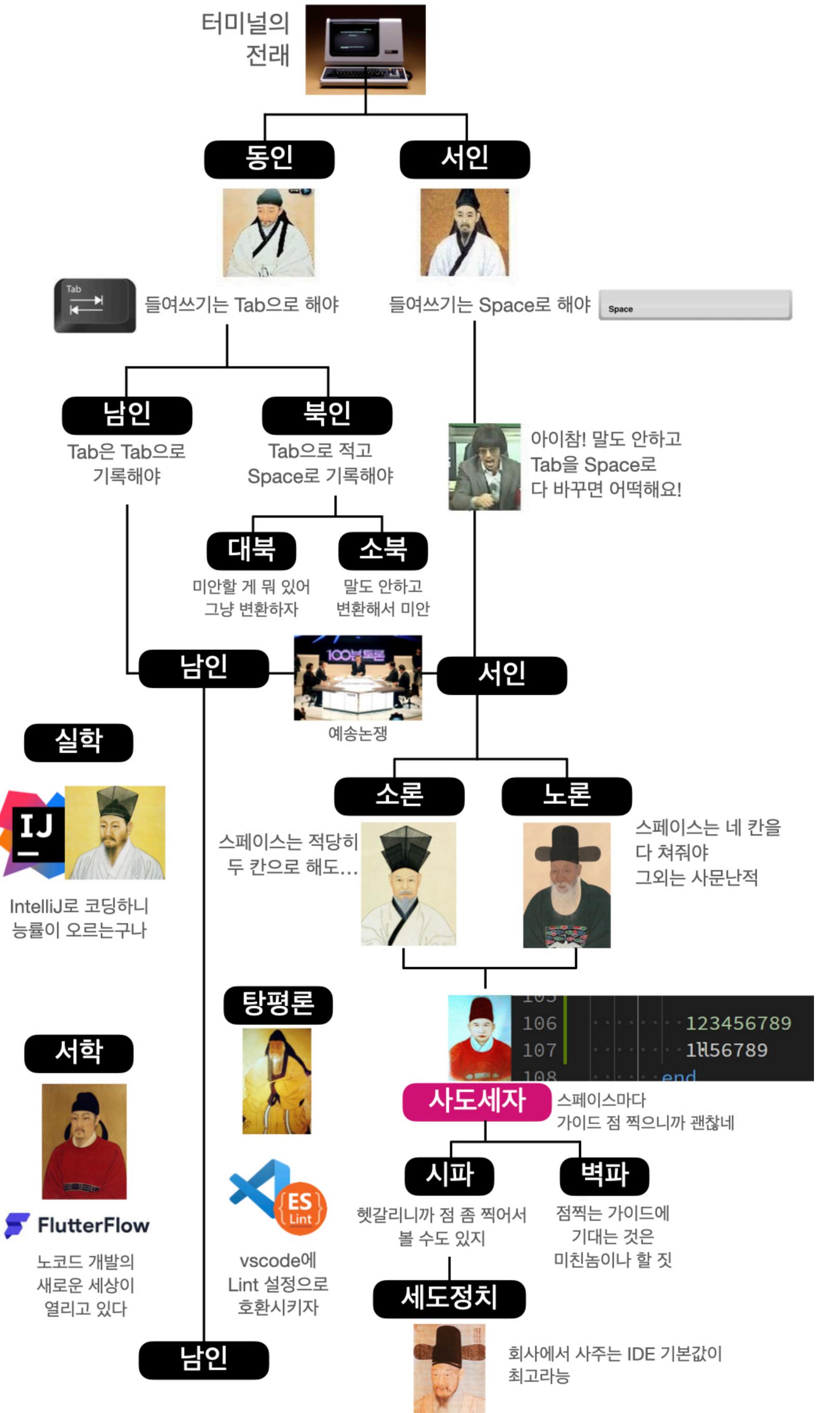


APP SCHOOL : iOS

앱 개발 기초

Swift API Design Guidelines

코딩으로 본 조선붕당의 이해



Apple이 권장하는 Swift 코딩 작성 규칙

Application Programming Interface

기본 사항

Fundamentals

- 사용 시점의 명확성이 가장 중요한 목표입니다.
- 메서드나 프로퍼티와 같은 엔티티는 한 번만 선언되지만 반복적으로 사용됩니다.
- 이러한 사용처를 명확하고 간결하게 만들 수 있도록 API를 설계하세요.
- 설계를 평가할 때는 선언문만 읽는 것만으로는 충분하지 않으며, 항상 사용 사례를 검토하여 문맥상 명확하게 보이는지 확인해야 합니다

기본 사항

Fundamentals

- 간결함보다 더 중요한 것은 명확성입니다.
- Swift 코드는 간결할 수 있지만, 최소한의 문자로 가능한 한 가장 작은 코드를 구현하는 것은 목표가 아닙니다.
- Swift 코드의 간결성은 강력한 타입 시스템과 상용구를 자연스럽게 줄여주는 기능의 부작용입니다.

기본 사항

Fundamentals

- 모든 선언에 대해 문서 주석을 작성하세요.
- 문서 작성을 통해 얻은 인사이트는 디자인에 큰 영향을 미칠 수 있으므로 미루지 마세요.

기본 사항

Fundamentals

- API의 기능을 간단한 용어로 설명하는데 어려움이 있다면 잘못된 API를 설계한 것일 수 있습니다.

기본 사항

Fundamentals - 주석 만들기

- Swift의 Markdown 문법을 사용하세요

- https://developer.apple.com/library/archive/documentation/Xcode/Reference/xcode_markup_formatting_ref/

Documentation Archive

Markup Formatting Reference

Markup Essentials

Markup Overview

Markup Functionality

Using Markup

Comment Markers for Markup

Formatting a Line of Text

Formatting Multiple Lines of Text

Formatting a Span of Characters

Inserting Links

Inserting Assets

Inserting Callouts

Escaping Special Characters

Revision History

Markup Overview

Use markup to create playgrounds that show formatted text in rendered documentation mode and to show Quick Help for your Swift code symbols.

Markup for playgrounds includes page level formatting for headings and other elements, formatting spans of characters, showing inline images, and several other features.

For example, Figure 1-1 shows two screenshots of the same playground. The page on the left-hand side of the figure shows the markup in the source editor view for the playground. The right-hand side of the figure shows the playground in rendered documentation mode.

Figure 1-1 Markup in playgrounds

Markup in source editor

Rendered documentation mode

Table of Contents

- Designing Your Tutorial
- Creating the Tutorial Project
- Adding Files

Overview

This is an overview of creating tutorial playgrounds.

Amazing content goes here.

Next Topic

Markup for Swift symbols is used for Quick Help and for the description shown in symbol completion.

For example, the description in the markup in Figure 1-2 is used for the description in symbol completion shown in Figure 1-3.

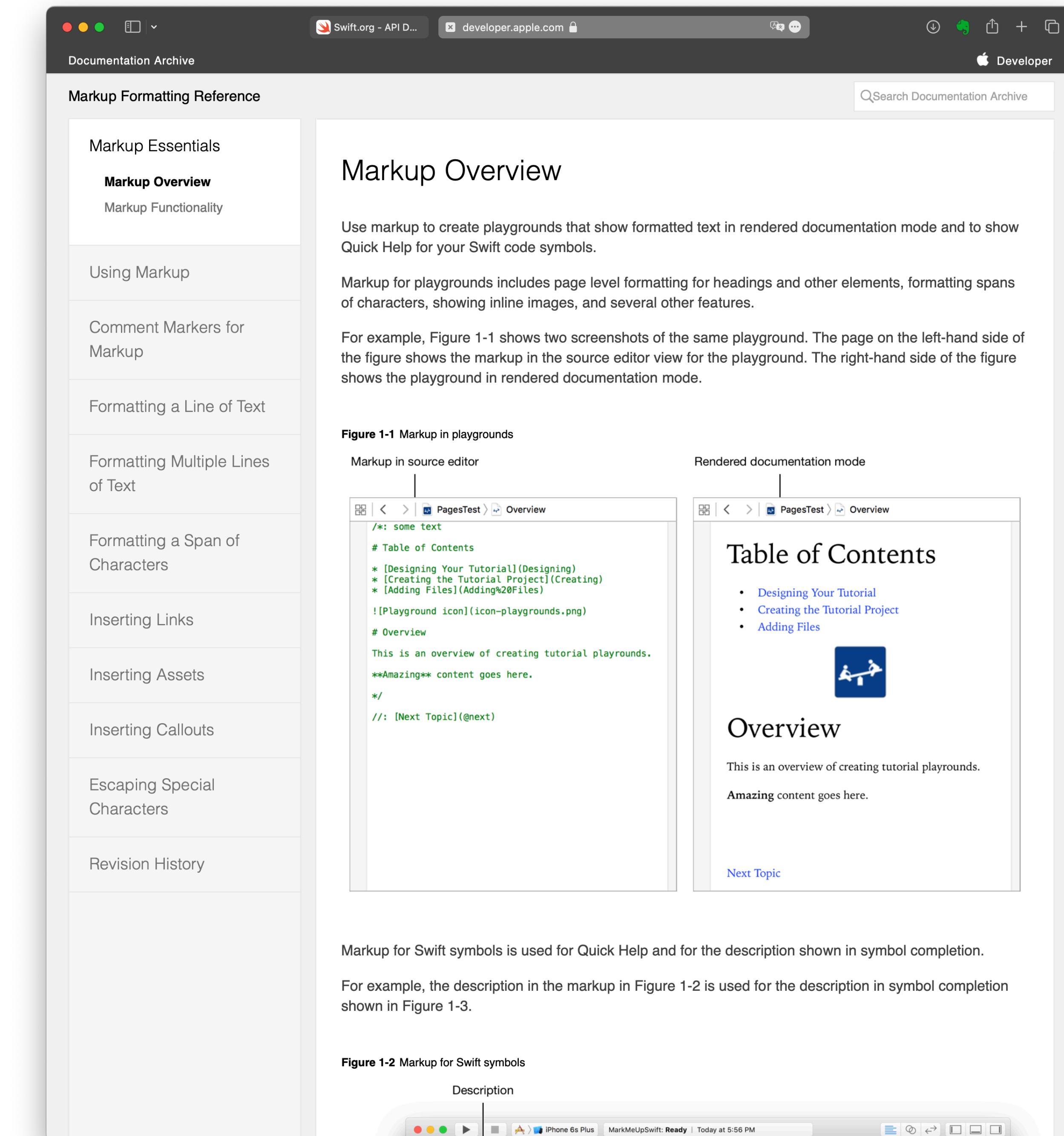
Figure 1-2 Markup for Swift symbols

Description

Swift.org - API D... developer.apple.com

Search Documentation Archive

Apple Developer



기본 사항

Fundamentals - 주석 만들기

- 선언되는 엔티티를 설명하는 요약으로 시작하세요.
- 종종 API는 선언과 요약으로 완전히 이해할 수 있는 경우가 많습니다.

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

기본 사항

Fundamentals - 주석 만들기

- 요약에 집중하세요. 요약이 가장 중요한 부분입니다.
 - 훌륭한 문서 댓글은 대부분 훌륭한 요약으로 구성되어 있습니다.
- 가능하면 마침표로 끝나는 단일 문장 조각을 사용하세요.
 - 완전한 문장을 사용하지 마세요.
- 함수나 메서드가 수행하는 작업과 반환하는 결과를 설명하되, 널 효과와 무효 반환은 생략하세요:

기본 사항

Fundamentals - 주석 만들기

- 함수나 메서드가 수행하는 작업과 반환하는 결과를 설명하되, 널 효과와 무효 반환은 생략하세요:

```
//> /// 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?
```

참고: 위의 팝퍼스트와 같이 드물게 요약이 세미콜론으로 구분된 여러 문장 조각으로 구성되는 경우가 있습니다.

기본 사항

Fundamentals - 주석 만들기

- 아래 첨자가 액세스하는 대상을 설명합니다:

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

기본 사항

Fundamentals - 주석 만들기

- 이니셜라이저가 생성하는 항목을 설명합니다:

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

기본 사항

Fundamentals - 주석 만들기

- 다른 모든 선언의 경우 선언된 엔티티가 무엇인지 설명합니다.

```
/// 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?
    ...
}
```

기본 사항

Fundamentals - 주석 만들기

- 원하는 경우 하나 이상의 단락과 글머리 기호 항목을 계속 작성합니다. 단락은 빈 줄로 구분하고 완전한 문장을 사용합니다.

```
/// Writes the textual representation of each ← Summary
/// element of `items` to the standard output.
///
/// The textual representation for each item `x` ← Additional discussion
/// is generated by the expression `String(x)`.

///
/// - Parameter separator: text to be printed } Parameters section
///   between items.
/// - Parameter terminator: text to be printed } Parameters section
///   at the end.

///
/// - Note: To print without a trailing } Symbol commands
///   newline, pass `terminator: ""`.
///
/// - SeeAlso: `CustomDebugStringConvertible`,
///   `CustomStringConvertible`, `debugPrint`. } Symbol commands
public func print(
    _ items: Any..., separator: String = " ", terminator: String = "\n")
```

기본 사항

Fundamentals - 주석 만들기

- 적절한 경우 인정된 기호 문서 마크업 요소를 사용하여 요약 이외의 정보를 추가할 수 있습니다.
- 기호 명령 구문으로 인식되는 글머리 기호 항목을 파악하고 사용하세요. Xcode와 같이 널리 사용되는 개발 도구는 다음 키워드로 시작하는 글머리 기호 항목을 특별히 처리합니다:

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

이름짓기

Naming

- 명확한 사용 촉진 1
 - 이름을 사용하는 코드를 읽는 사람이 모호함을 느끼지 않도록 필요한 모든 단어를 포함하세요.
 - 예를 들어 컬렉션 내 특정 위치에서 요소를 제거하는 메서드를 생각해 보세요.

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



이름짓기

Naming

- 명확한 사용 촉진 1
 - 이름을 사용하는 코드를 읽는 사람이 모호함을 느끼지 않도록 필요한 모든 단어를 포함하세요.
 - 메서드 서명에서 `at`이라는 단어를 생략하면 독자는 메서드가 제거할 요소의 위치를 나타내기 위해 `x`를 사용하는 것이 아니라 `x`와 동일한 요소를 검색하여 제거한다는 것을 암시할 수 있습니다.

```
employees.remove(x) // unclear: are we removing x?
```



이름짓기

Naming

- 명확한 사용 촉진 2
 - **불필요한 단어는 생략하세요.** 이름에 포함된 모든 단어는 사용 사이트에서 중요한 정보를 전달해야 합니다.
 - 의도를 명확히 하거나 의미를 모호하게 하기 위해 더 많은 단어가 필요할 수 있지만, 독자가 이미 알고 있는 정보와 중복되는 단어는 생략해야 합니다. 특히 유형 정보를 단순히 반복하는 단어는 생략합니다.

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



```
allViews.removeElement(cancelButton)
```

이름짓기

Naming

- 명확한 사용 촉진 3
 - 변수, 매개변수 및 관련 유형은 유형 제약 조건이 아닌 역할에 따라 이름을 지정합니다.
 - 아래 방식으로 유형 이름을 변경하면 명확성과 표현력을 최적화하지 못합니다.

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



이름짓기

Naming

- 명확한 사용 촉진 3
 - 대신 엔티티의 역할을 표현하는 이름을 선택하도록 노력하세요.

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



이름짓기

Naming

- 명확한 사용 촉진 3
- 연결된 유형이 프로토콜 제약 조건에 너무 엄격하게 바인딩되어 프로토콜 이름이 역할 인 경우 프로토콜 이름에 프로토콜을 추가하여 충돌을 피하세요:

```
protocol Sequence {  
    associatedtype Iterator : IteratorProtocol  
}  
protocol IteratorProtocol { ... }
```

이름짓기

Naming

- 명확한 사용 촉진 4
 - 약한 유형 정보를 보완하여 매개변수의 역할을 명확히 합니다.
 - 특히 매개변수 유형이 `NSObject`, `Any`, `AnyObject`이거나 `Int` 또는 `String`과 같은 기본 유형인 경우 사용 시점의 유형 정보 및 컨텍스트가 의도를 완전히 전달하지 못할 수 있습니다. 이 예에서는 선언은 명확하지만 사용 위치가 모호할 수 있습니다.

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



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

이름짓기

Naming

- 명확한 사용 촉진 4
 - 약한 유형 정보를 보완하여 매개변수의 역할을 명확히 합니다.
 - 명확성을 회복하려면 약하게 입력된 각 매개변수 앞에 그 역할을 설명하는 명사를 붙이세요:

```
func addObserver(_ observer: NS0bject, forKeyPath path: String)  
grid.addObserver(self, forKeyPath: graphics) // clear
```



이름짓기

Naming

- 유창한 사용법을 위한 노력 1
 - 사용 사이트가 문법적인 영어 구문을 형성하는 메서드 및 함수 이름을 선호합니다.

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()

이름짓기

Naming

- 유창한 사용법을 위한 노력 1
 - 첫 번째 인수가 통화 의미의 핵심이 아닌 경우 한두 개의 인수 이후에는 유창성이 저하되는 것은 허용됩니다:

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

이름짓기

Naming

- 유창한 사용법을 위한 노력 2
 - 팩토리 메서드의 이름은 "make"로 시작합니다.
 - 예: x.makelterator()

이름짓기

Naming

- 유창한 사용법을 위한 노력 3
 - 이니셜라이저 및 팩토리 메서드 호출의 첫 번째 인수는 기본 이름으로 시작하는 구를 형성해서는 안 됩니다
 - 예: x.makeWidget(cogCount: 47)

이름짓기

Naming

- 유창한 사용법을 위한 노력 3
 - 예를 들어 이러한 호출의 첫 번째 인수는 기본 이름과 같은 구문의 일부로 읽히지 않습니다:

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



이름짓기

Naming

- 유창한 사용법을 위한 노력 3
 - 아래에서는 API 작성자가 첫 번째 인수를 사용하여 문법적 연속성을 만들려고 시도했습니다.

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

이름짓기

Naming

- 유창한 사용법을 위한 노력 3
 - 실제로 이 지침은 인자 레이블에 대한 지침과 함께 호출이 값 보존 유형 변환을 수행하지 않는 한 첫 번째 인자에 레이블을 갖는다는 것을 의미합니다.

```
let rgbForeground = RGBColor(cmykForeground)
```

이름짓기

Naming

- 유창한 사용법을 위한 노력 4
 - 함수와 메서드의 부작용(side-effect)를 고려해 이름을 지정합니다.
 - 부작용이 없는 함수는 명사 구문으로 읽어야 합니다.
 - 예: `x.distance(to: y)`, `i.successor()`
 - 부작용이 있는 것은 명령형 동사 구문으로 읽어야 합니다.
 - 예: `print(x)`, `x.sort()`, `x.append(y)`

이름짓기

Naming

- 유창한 사용법을 위한 노력 4
 - **변형(Mutating) / 비변형(nonmutating)** 메서드 쌍의 이름을 일관되게 지정합니다.
 - Mutating 메서드에는 종종 비슷한 의미를 가진 nonmutating 변형이 있지만 인스턴스를 제자리에서 업데이트하지 않고 새 값을 반환합니다.

이름짓기

Naming

- 유창한 사용법을 위한 노력 4
 - 연산이 동사로 자연스럽게 설명되는 경우, 동사의 명령형을 mutating 메서드에 사용하고 "ed" 또는 "ing" 접미사를 적용하여 변경이 일어나지 않는 상대방의 이름을 지정합니다.

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

이름짓기

Naming

- 유창한 사용법을 위한 노력 4
 - 동사의 과거 분사를 사용하여 비변형 변형의 이름을 지정하는 것을 선호합니다(일반적으로 "ed"를 추가):

```
/// Reverses `self` in-place.  
mutating func reverse()  
  
/// Returns a reversed copy of `self`.  
func reversed() -> Self  
...  
x.reverse()  
let y = x.reversed()
```

이름짓기

Naming

- 유창한 사용법을 위한 노력 4
 - 동사에 직접 목적어가 있어 "ed"를 추가하는 것이 문법적으로 맞지 않는 경우 동사의 현재 분사를 사용하여 "ing"를 추가하여 nonmutating형 이름을 지정합니다.

```
/// Strips all the newlines from `self`
mutating func stripNewlines()

/// Returns a copy of `self` with all the newlines stripped.
func strippingNewlines() -> String
...
s.stripNewlines()
let oneLine = t.strippingNewlines()
```

이름짓기

Naming

- 유창한 사용법을 위한 노력 4
 - 연산이 명사로 자연스럽게 설명되는 경우, nonmutating이 메서드에 명사를 사용하고 "form" 접두사를 적용하여 변이 메서드에 해당하는 이름을 지정합니다.

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

이름짓기

Naming

- 유창한 사용법을 위한 노력 5
 - Boolean 메서드 및 프로퍼티의 사용은 변이가 없는 경우 수신자에 대한 어설션으로 읽어야 합니다.
 - x.isEmpty
 - line1.intersects(line2)

이름짓기

Naming

- 유창한 사용법을 위한 노력 6
 - 어떤 것이 무엇인지 설명하는 프로토콜은 명사로 읽어야 합니다.
 - 예: Collection

이름짓기

Naming

- 유창한 사용법을 위한 노력 7
 - 기능을 설명하는 프로토콜은 접미사 able, ible 또는 ing를 사용하여 이름을 지정해야 합니다.
 - 예: Equatable, ProgressReporting

이름짓기

Naming

- 유창한 사용법을 위한 노력 8
 - 다른 유형, 속성, 변수 및 상수의 이름은 명사로 읽어야 합니다.

올바른 용어 사용

Use Terminology Well

- Term of Art
 - noun
 - a word or phrase that has a precise, specialized meaning within a particular field or profession.
 - 명사
 - 특정 분야 또는 직업 내에서 정확하고 전문적인 의미를 갖는 단어 또는 구문입니다.

올바른 용어 사용

Use Terminology Well

- 더 일반적인 단어로도 충분히 의미를 전달할 수 있다면 모호한 용어는 피하세요.
- ‘피부(skin)’가 목적에 부합한다면 ‘표피(epidermis)’라고 말하지 마세요.
- 전문 용어는 필수적인 커뮤니케이션 도구이지만, 그렇지 않으면 놓칠 수 있는 중요한 의미를 포착할 때만 사용해야 합니다.

올바른 용어 사용

Use Terminology Well

- 예술 용어를 사용할 경우 정해진 의미에 충실하세요.
- 일반적인 단어 대신 전문 용어를 사용하는 유일한 이유는 모호하거나 불분명할 수 있는 내용을 정확하게 표현하기 위해서입니다. 따라서 API는 해당 용어가 허용되는 의미에 따라 엄격하게 사용해야 합니다.
- **전문가를 놀라게 하지 마세요:** 이미 이 용어에 익숙한 사람은 우리가 새로운 의미를 만들어낸 것처럼 보이면 놀랄 것이고 아마도 화를 낼 것입니다.
- **초보자를 혼동하지 않기:** 이 용어를 배우려는 사람은 웹 검색을 통해 기존 의미를 찾을 가능성이 높습니다.

올바른 용어 사용

Use Terminology Well

- 약어는 피하세요.
- 약어, 특히 비표준 약어는 줄임말이 아닌 형태로 정확하게 번역해야만 이해할 수 있으므로 사실상 전문 용어에 해당합니다.
- 사용하는 약어의 의도된 의미는 웹 검색을 통해 쉽게 찾을 수 있어야 합니다.

올바른 용어 사용

Use Terminology Well

- 선례를 수용하세요.
- 기존 문화에 순응하는 대신 초보자를 위해 용어를 최적화하지 마세요.
- 초보자가 List의 의미를 더 쉽게 이해할 수 있더라도 List와 같은 단순화된 용어를 사용하는 것보다 연속적인 데이터 구조의 이름을 Array로 지정하는 것이 더 낫습니다.
- 배열은 최신 컴퓨팅의 기본이므로 모든 프로그래머는 배열이 무엇인지 알고 있거나 곧 알게 될 것입니다.
- 대부분의 프로그래머가 익숙한 용어를 사용하면 웹 검색과 질문에 대한 보상을 받을 수 있습니다.

올바른 용어 사용

Use Terminology Well

- 선례를 수용하세요.
- 기존 문화에 순응하는 대신 초보자를 위해 용어를 최적화하지 마세요.
- 수학과 같은 특정 프로그래밍 영역에서는 각도(x)와 같은 설명 문구보다는 $\sin(x)$ 와 같이 널리 사용되는 용어를 사용하는 것이 좋습니다.
- 이 경우 축약어를 피하라는 지침보다 선례가 더 중요하다는 점에 유의하세요.
- 완전한 단어는 사인이지만 " $\sin(x)$ "는 수십 년 동안 프로그래머들 사이에서, 그리고 수세기 동안 수학자들 사이에서 일반적으로 사용되어 왔습니다.

규칙

Conventions

- 일반 규칙 1
 - $O(1)$ 이 아닌 모든 계산된 프로퍼티의 복잡도를 문서화합니다.
 - 사람들은 프로퍼티를 멘탈 모델로 저장해 두었기 때문에 프로퍼티 액세스에 큰 계산이 필요하지 않다고 생각하는 경우가 많습니다.
 - 이러한 가정이 깨질 수 있는 경우 반드시 경고하세요.

규칙

Conventions

- 일반 규칙 2
 - 자유로운 함수보다 메서드와 속성을 선호합니다. 자유 함수는 특별한 경우에만 사용됩니다.
 - 명백한 `self`가 없는 경우:
 - `min(x, y, z)`
 - 함수가 제약되지 않은 제네릭인 경우:
 - `print(x)`
 - 함수 구문이 정해진 도메인 표기법의 일부인 경우:
 - `sin(x)`

규칙

Conventions

- 일반 규칙 3
 - 대소문자 규칙을 따릅니다. 유형과 프로토콜의 이름은 대문자 대소문자를 사용합니다. 그 외 모든 것은 소문자로 표기합니다.
 - 미국 영어에서 일반적으로 모두 대문자로 표시되는 약어와 이니셜은 대소문자 규칙에 따라 균일하게 대문자 또는 소문자로 표시해야 합니다:

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

- 다른 약어는 일반 단어로 취급해야 합니다:

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

규칙

Conventions

- 일반 규칙 4
 - 메서드는 동일한 기본 의미를 공유하거나 서로 다른 도메인에서 작동하는 경우 기본 이름을 공유할 수 있습니다.
 - 예를 들어, 메서드는 본질적으로 동일한 작업을 수행하므로 다음과 같이 사용하는 것이 좋습니다:

```
extension Shape {  
    /// Returns `true` if `other` is within the area of `self`;  
    /// otherwise, `false`.  
    func contains(_ other: Point) -> Bool { ... }  
  
    /// Returns `true` if `other` is entirely within the area of `self`;  
    /// otherwise, `false`.  
    func contains(_ other: Shape) -> Bool { ... }  
  
    /// Returns `true` if `other` is within the area of `self`;  
    /// otherwise, `false`.  
    func contains(_ other: LineSegment) -> Bool { ... }  
}
```

규칙

Conventions

- 일반 규칙 4
 - 그리고 기하학적 유형과 컬렉션은 별도의 도메인이므로 동일한 프로그램에서 이 또한 괜찮습니다:

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

- 그러나 이러한 인덱스 메서드는 서로 다른 의미를 가지므로 이름을 다르게 지정해야 합니다:

```
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 { ... }  
}
```

규칙

Conventions

- 일반 규칙 4
 - 마지막으로, 유형 추론이 있는 경우 모호성을 유발하므로 '반환 유형에 대한 과부하'를 피하세요.

```
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? { ... }  
}
```

규칙

Conventions

- 매개변수

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

규칙

Conventions

- 매개변수 1
 - 문서를 제공할 매개변수 이름을 선택합니다.
 - 매개변수 이름은 함수나 메서드의 사용 지점에 나타나지 않더라도 중요한 설명 역할을 합니다.
- 이러한 이름을 선택하면 문서를 읽기 쉽게 만들 수 있습니다. 예를 들어, 이러한 이름을 사용하면 문서를 자연스럽게 읽을 수 있습니다:

```
// 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])
```

규칙

Conventions

- 매개변수 1
 - 문서를 제공할 매개변수 이름을 선택합니다.
 - 매개변수 이름은 함수나 메서드의 사용 지점에 나타나지 않더라도 중요한 설명 역할을 합니다.
 - 하지만 이렇게 하면 문서가 어색하고 문법에 맞지 않게 됩니다:

```
/// 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])
```

규칙

Conventions

- 매개변수 2
 - 기본값 매개변수는 일반적인 사용을 단순화할 때 활용하세요.
 - 일반적으로 사용되는 값이 하나뿐인 매개변수는 모두 기본값 후보가 될 수 있습니다.

규칙

Conventions

- 매개변수 2
 - 기본 인수는 관련 없는 정보를 숨겨 가독성을 높여줍니다. 예를 들어

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



- 위 코드는 아래와 같이 훨씬 더 간단해질 수 있습니다:

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



규칙

Conventions

- 매개변수 2
 - 기본 인수는 일반적으로 메서드 패밀리를 사용하는 것보다 선호되는데, 이는 API를 이해하려는 모든 사람에게 인지적 부담을 덜 주기 때문입니다.

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



규칙

Conventions

- 매개변수 2
 - 앞선 방법은 간단하지 않을 수 있지만, 다음의 경우보다 훨씬 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  
}
```

규칙

Conventions

- 매개변수 2
 - 메서드 패밀리의 모든 멤버는 개별적으로 문서화되어 사용자가 이해할 수 있어야 합니다.
 - 사용자는 모든 메서드를 이해해야 하며, 때로는 예상치 못한 관계(예: foo(bar: nil)와 foo()가 항상 동의어가 아닌 경우)로 인해 대부분 동일한 문서에서 사소한 차이점을 찾아내는 지루한 과정을 거쳐야 합니다.
 - 기본값이 있는 단일 메서드를 사용하면 훨씬 더 뛰어난 프로그래머 환경을 제공합니다.

규칙

Conventions

- 매개변수 3
 - 기본값이 있는 매개변수는 매개변수 목록의 맨 끝에 위치하는 것이 좋습니다.
 - 기본값이 없는 매개변수는 일반적으로 메서드의 의미론에 더 필수적이며, 메서드가 호출되는 초기 사용 패턴을 안정적으로 제공합니다.

규칙

Conventions

- 매개변수 4
 - API가 프로덕션 환경에서 실행될 경우 다른 것보다 #fileID를 선호하세요.
 - #fileID는 공간을 절약하고 개발자의 개인정보를 보호합니다.
 - 전체 경로가 개발 워크플로우를 간소화하거나 파일 I/O에 사용되는 경우 최종 사용자가 실행하지 않는 API(예: 테스트 헬퍼 및 스크립트)에는 #filePath를 사용하세요.
 - Swift 5.2 이하 버전과의 소스 호환성을 유지하려면 #file을 사용하세요.

규칙

Conventions

- 인수 레이블 1

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

- 인수를 유용하게 구분할 수 없는 경우 모든 레이블을 생략합니다
 - 예: min(number1, number2), zip(sequence1, sequence2)).

규칙

Conventions

- 인수 레이블 2
 - 값 보존 유형 변환을 수행하는 이니셜라이저에서는 첫 번째 인자 레이블을 생략합니다.
 - 예: Int64 (someUInt32)

규칙

Conventions

- 인수 레이블 2
 - 첫 번째 인수는 항상 변환의 소스여야 합니다.

```
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)
```

규칙

Conventions

- 인수 레이블 2
 - 하지만 ‘축약’ 유형 전환에서는 축약을 설명하는 레이블을 사용하는 것이 좋습니다.

```
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)  
}
```

규칙

Conventions

- 인수 레이블 3
 - 첫 번째 인수가 전치사 구의 일부를 구성하는 경우 인자 레이블을 지정합니다.
 - 인자 레이블은 일반적으로 전치사에서 시작해야 합니다.
 - 예: `x.removeBoxes(havingLength: 12)`

규칙

Conventions

- 인수 레이블 3
 - 처음 두 인수가 단일 추상화의 일부를 나타내는 경우에는 예외가 발생합니다.

```
a.move(toX: b, y: c)
```



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

- 이러한 경우 추상화를 명확하게 하기 위해 전치사 뒤에 인자 레이블을 시작하세요.

```
a.moveTo(x: b, y: c)
```



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

규칙

Conventions

- 인수 레이블 4
 - 그렇지 않으면 첫 번째 인수가 문법 구의 일부를 구성하는 경우 레이블을 생략하고 기본 이름에 앞의 단어를 추가합니다.
 - 예: x.addSubview(y)
 - 이 지침은 첫 번째 인수가 문법 구의 일부를 구성하지 않는 경우 레이블을 지정해야 함을 의미합니다.

규칙

Conventions

- 인수 레이블 4

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

- 문구가 정확한 의미를 전달하는 것이 중요하다는 점에 유의하세요.
- 다음은 문법적으로는 맞지만 잘못된 것을 표현합니다.

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

- 기본값이 있는 인수는 생략할 수 있으며, 이 경우 문법 구문의 일부를 구성하지 않으므로 항상 레이블을 사용해야 합니다.

규칙

Conventions

- 인수 레이블 5
 - 다른 모든 인수에 레이블을 지정합니다.

특별 지침

Special Instructions

- 튜플 멤버에 레이블을 지정하고 API에 표시되는 클로저 매개변수에 이름을 지정합니다.
- 이러한 이름은 설명력이 있고, 문서 주석에서 참조할 수 있으며, 튜플 멤버에 대한 표현식 액세스를 제공합니다.

특별 지침

Special Instructions

```
/// 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` if a new block of memory
///     was allocated; otherwise, `false`.
///   - capacityChanged: `true` if `capacity` was updated;
///     otherwise, `false`.
mutating func ensureUniqueStorage(
    minimumCapacity requestedCapacity: Int,
    allocate: (_ byteCount: Int) -> UnsafePointer<Void>
) -> (reallocated: Bool, capacityChanged: Bool)
```

- 클로저 매개변수에 사용되는 이름은 최상위 함수의 매개변수 이름처럼 선택해야 합니다.
- 호출 사이트에 표시되는 클로저 인수의 레이블은 지원되지 않습니다.

특별 지침

Special Instructions

- 제약되지 않은 다형성(예: 임의, 임의 객체 및 제약되지 않은 일반 매개변수)을 사용할 때는 과부하(overload) 세트의 모호성을 피하기 위해 각별히 주의하세요.

특별 지침

Special Instructions

- 예를 들어 이 과부하 세트를 살펴보겠습니다:

```
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  
}
```



특별 지침

Special Instructions

- 이러한 메서드는 의미적 계열을 형성하며 인자 유형은 처음에는 뚜렷하게 구분되는 것처럼 보입니다.
- 그러나 Element가 Any인 경우 단일 엘리먼트는 엘리먼트 시퀀스와 동일한 유형을 가질 수 있습니다.

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



특별 지침

Special Instructions

- 모호함을 없애려면 두 번째 과부하의 이름을 더 명확하게 지정하세요.

```
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  
}
```



특별 지침

Special Instructions

- 새 이름이 문서 주석과 더 잘 어울리는지 확인하세요.
- 이 경우 문서 주석을 작성하는 행위가 실제로 API 작성자의 주의를 환기시켰습니다.



감사합니다