Руководство по проектированию Swift API

I. Основы

- **Ваша основная цель - обеспечить понятность кода в точке его использования.**
  Методы и свойства объявляются единожды, но используются неоднократно. Проектируйте API так, чтобы методы и свойства были понятными и краткими. При ознакомлении с дизайном API, просто декларирования методов и свойств обычно бывает недостаточно; всегда старайтесь показать методы и свойства в действии; чтобы был понятен контекст их применения.

- **Понятность кода более важна, чем его краткость.**
  Хотя код на языке Swift может быть компактным, наша цель не состоит в том, чтобы создать самый маленький код с несколькими символами. Краткость в Swift , если мы ее наблюдаем, - это побочный эффект системы со строгой типизацией и конструктивными возможностями, естественно уменьшающими некоторые шаблоны.

- **Пишите документирующий комментарий** для каждого декларирования.
  Идеи, усиленные написанием документации, могут дать основательное улучшение вашего дизайна, так что не избегайте этого.

  Если вы не можете описать функциональность API простыми словами, то **возможно вы спроектировали неверный API**

- **Используйте специальный Swift диалект языка разметки Markdown.**

- **Начните с резюме,** которое описывает декларируемую сущность.
  Часто API можно полностью понять из декларирования и его резюме.

```
/// Возвращает `self` "view" , содержащее те же самые элементы, но
/// в обратном порядке.
  func reversed() -> ReverseCollection
```

- **Сфокусируйтесь на резюме -** это самая важная часть.
  Множество отличных документирующих комментариев состоит не более чем из великолепного резюме.

- **Используйте фрагменты одного предложения**, если это возможно, заканчивая точкой. Не используйте всё предложение целиком.

- **Описывайте, ЧТО функция или метод делают и Что возвращают,** пропуская null эффекты и Void возвращаемые значения:

```
/// Вставляет `newHead` в начало `self`.
  mutating func prepend(_ newHead: Int)
/// Возвращает `List`, содержащий `head`, за которым следуют элементы `self`.
  func prepending(_ head: Element) -> List
/// Удаляет и возвращает первый элемент `self`, если он не пустой;
/// в противном случае возвращает `nil`.
  mutating func popFirst() -> Element?
```

Примечание: в редких случаях как в popFirst выше, резюме состоит из нескольких фрагментов предложения, разделенных точкой с запятой.

- **Описывайте, к чему сабскрипт обращается:**

```
/// Доступ к элементу с индексом `index`.
  subscript(index: Int) -> Element { get set }
```

- **Описывайте, что инициализатор создаёт:**

```
/// Создает экземпляр, содержащий `n` повторений `x`.
  init(count n: Int, repeatedElement x: Element)
```

- Для остальных объявлений **описывайте, что представляет собой объявляемая сущность:**

```
/// Коллекция collection, которая поддерживает одинаковую эффективность вставки/удаления
/// элемента в любой позиции.
  struct List {
/// Начальный элемент `self`, или `nil`, если `self` - пустой.
  var first: Element?
...
```

- **В случае необходимости продолжайте** с использованием одного или более параграфов и маркированным списком. Параграфы разделяются пустыми линиями и используют законченные предложения.

```
/// Выводит текстовое представление каждого                    ← Резюме
/// элемента `items` на стандартное устройство вывода.
///                                                            ← Пустая строка
/// Текстовое представление для каждого элемента`x`            ← Дополнительное обсуждение
/// генерируется выражение `String(x)`.
///
///     - Параметр разделителя (separator): этот текст будет печататься   Секция параметров
///       между элементами `items`.
///     - Параметр завершения (terminator): этот текст будет печататься
///       в конце.
///
///     - Замечание: Для печати без перехода в конце                      Symbol commands
///       на новую строчку передайте параметр `terminator: ""`
///     - Также смотри: `CustomDebugStringConvertible`,
///       `CustomStringConvertible`, `debugPrint`.
public func print(
_ items: Any..., separator: String = " ", terminator: String = = "\n")
```

- **Используйте [узнаваемые элементы языка разметки Swift](), ** чтобы добавить информацию за пределами резюме, где это необходимо.

- **Познакомьтесь и используйте узнаваемые маркеры с [синтаксисом командных обозначений]().**
  Такие популярные инструменты разработки, как Xcode, предоставляют специальную обработку для маркеров которые начинаются со следующих ключевых слов:

Attention	Author	Authors	Bug

Complexity	Copyright	Date	Experiment
Important	Invariant	Note	Parameter
Parameters	Postcondition	Precondition	Remark
Requires	Returns	SeeAlso	Since
Throws	Todo	Version	Warning

II. Именование.

1. Способствуйте понятному использованию

- **Включайте все слова необходимые чтобы избежать неоднозначности** для человека, читающего код с использованием этого имени.
  Например, рассмотрим метод, который удаляет элемент коллекции в определённой позиции:

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

Если мы опустим предлог "at" из описания метода, читателю может показаться, что метод ищет и удаляет элемент равный x, вместо того, чтобы использовать x для указания позиции элемента для удаления

```
employees.remove(x) // непонятно: мы удаляем x?
```

- **Избегайте ненужных слов.** Каждое слово в имени должно передавать существенную информацию с точки зрения использования.

  Большее количество слов может понадобиться чтобы уточнить намерение или устранить неоднозначность, но те лишние слова с информацией, которой читатель уже обладает должны быть устранены. В частности, опускайте слова, которые просто повторяют информацию о типах.

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

allViews.removeElement(cancelButton)
```

  В этом случае слово Element ничего не добавляет для читающего код. API может быть лучше:

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

allViews.remove(cancelButton)
```

  Случается, что повтор информации о типе необходим, чтобы избежать неоднозначности, но, в основном, лучше использовать слова, которые описывают роль параметра, а не его тип. Смотрите следующий пункт для более детальной информации.

- **Называйте переменные, параметры и ассоциативные типы исходя из их ролей,** а не из ограничений их типов.

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

  Называя переменные именем Типа (**string**), вы лишаете программиста, читающего ваш код, ясность понимания и выразительности.
  Вместо этого постарайтесь выбрать имя, которое выражает роль переменной:

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

```
}
```

Если **associatedtype** прочно связан с ограничениями протокола и имя протокола является его ролью, избегайте противоречий, которые могут возникнут при добавлении слова **Type** к имени **associatedtype**:

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

- **Жертвуйте информацией о типах в угоду ясности роли параметра.**
  Особенно когда параметр имеет тип NSObject, Any, AnyObject или фундаментальный тип Int или String, тип информации или контекст в точке использования может не полно передавать намерение. В этом примере, декларация понятная, но использование осталось неясным:

```
func add (_ observer: NSObject, for keyPath: String)
grid.add(self, for: graphics) // н е п о н я т н о
```

Чтобы восстановить ясность, **добавьте каждому слабо типизированному параметру существительное с описанием роли**:

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

2. Стремитесь к использованию "чистого" английского языка

- **Предпочитайте имена методов и функций, которые можно истолковать как грамматические фразы на английском языке**

```
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()
```

Это правило ослабляет свое действие после одного-двух аргументов, когда оставшиеся аргументы не играют решающего значении при вызове метода:

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

- **Начинайте имена “фабричных” методов со слова "make",** например, `x.makeIterator()`
- **Вызовы инициализаторов и [фабричных методов](#)** должны создавать фразу, не включающую первый аргумент, например: `x.makeWidget(cogCount: 47)`

  Например, фразы, воспроизводимые этими вызовами, не включают первый аргумент:

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

  В следующем API автор пробует создать грамматическую целостность с первым аргументом:

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

На практике эта рекомендация вместе с рекомендациями **[для названий аргументов](#)** означает, что первый аргумент будет иметь название, если вызов метода не связан с **[преобразованием типов](#)**

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

- **Называйте функции и методы исходя из их побочных эффектов**
  - названия функций и методов без побочных эффектов должны читаться как существительные, например: `x.distance(to: y)`, `i.successor()`.
  - названия функций и методов с побочными эффектами должны читаться как глаголы в повелительном наклонении, например: `print(x)`, `x.sort()`, `x.append(y)`.
  - **Именование изменяющих (mutating) / неизменяющих (nonmutating) методов должны попарно соответствовать друг другу.** Изменяющий метод зачастую будет иметь схожую семантику с неизменяющим методом, но будет возвращать новое значение, а не изменять существующий экземпляр "по месту".
  -
    - Когда операция естественно описывается глаголом, используйте глагол в повелительном наклонении для изменяющих методов и добавляйте суффикс “ed” or “ing” для соответствующих не изменяющих методов

Изменяющий	Не изменяющий
`x.sort()`	`z = x.sorted()`
`x.append(y)`	`z = x.appending(y)`

- Стремитесь называть неизменяющие методы с использованием глагольного причастия в прошедшем времени (обычно с окончанием "ed"):

```
/// Переворачивает (Reverses) `self` "по месту".
  mutating func reverse()
///Возвращает перевернутую (reversed) копию `self`.
  func reversed() -> Self
  ...
  x.reverse()
  let y = x.reversed()
```

- Когда добавление "ed" грамматически неверно, потому что глагол имеет прямое дополнение, называйте неизменяющий вариант с использованием глагольного причастия в настоящем времени (обычно с окончанием "ing"):

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

/// Возвращает копию `self` with all the newlines stripped.
  func strippingNewlines() -> String
...
  s.stripNewlines()
  let oneLine = t.strippingNewlines(
```

- Когда операция естественно описывается существительным, используйте существительное для неизменяющих методов и добавляйте префикс “form”, чтобы назвать соответствующий изменяющий вариант:

Неизменяющий	Изменяющий
`x = y.union(z)`	`y.formUnion(z)`

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

- **Использование булевых методов и свойств должно читаться как утверждение о результате,** когда используется неизменяемая форма, например: `x.isEmpty`, `line1.intersects(line2)`

- **Протоколы, которые описывают то, что чем-то является, должны читаться как существительные,** (например: `Collection`).

- **Протоколы, которые описывают возможность, должны именоваться с суффиксами** able**,** ible**, or** ing (например: `Equatable`, `ProgressReporting`)**.**

- **Имена остальных типов, свойств, переменных и констант должны читаться как существительные.**

3. Правильно используйте терминологию

Специальный термин (сущ.) - слово или фраза, которая имеет точное специализированное значение в конкретной области или профессии.

- **Избегайте неясных терминов,** если более общее слово хорошо отражает смысл.
Не говорите "эпидермис", если "кожа" подойдет для использования в наших целях.
Специальные термины - ценный инструмент коммуникации, но следует их использовать чтобы ухватить ключевой смысл, который в другом случае пропадет.

- **Придерживайтесь установленных обозначений, если используете специальный термин.**

  Единственная причина использовать технический термин вместо более общего слова - это точное выражение чего-то, что в другом случае будет противоречиво или неясно. Поэтому API должно использовать термин прямо в соответствии с принятым значением.

  - **Не удивляйте эксперта:** любой, кто уже знаком с термином будет удивлён и возможно раздосадован, если мы будем изобретать новое значение этого термина

  - **Не путайте новичка:** любой, кто начинает изучать термин скорее всего будет делать интернет поиск и найдёт традиционное значение.

- **Избегайте аббревиатур.** Аббревиатуры, особенно нестандартные, - это фактически специальные термины, потому что их понимание зависит от правильного перевода аббревиатур в полную форму.

> Расшифровка любой аббревиатуры, которую вы намереваетесь использовать, должна легко находиться при поиске в интернете.

- **Включайте прецеденты.**

  Не оптимизируйте значения терминов для абсолютных новичков ценой отказа от уже существующей культуре терминов в этой области.

Лучше назвать последовательную структуру данных `Array`, чем использовать упрощённый термин `List`, хотя новичок может усвоить значение `List` более легко. Массивы (`Arrays`) являются основополагающими структурами данных в современных вычислениях, поэтому каждый программист рано или поздно узнает, что такое массивы `Arrays`. Используйте термины, с которым большинство программистов знакомы, и поиски и вопросы новичков будут вознаграждены.

В определённом программистском кругу, таком как математики, широко распространены математические термины типа `sin(x)` , которые более предпочтительны объяснительным фразам типа `verticalPositionOnUnitCircleAtOriginOfEndOfRadiusWithAngle`. Учтите, что в этом случае прецедент перевешивает рекомендации избегать аббревиатур: хотя полное слово - sine, "$\sin(x)$" - было в общем употреблении среди программистов на протяжении десятилетий, а среди математиков - на протяжении веков.

III. Соглашения.

1. Основные соглашения

- **Документируйте сложность любого вычисляемого свойства (computed property) которая не равно O(1).** Люди часто по своему опыту предполагают, что обращение к свойствам не вовлекает значительных вычислений, потому что они хранят значения. Будьте уверены, что предупредили их, когда это предположение может быть нарушено.

- **Стремитесь использовать методы и свойства, а не свободные функции.** Свободные функции используются только в специальных случаях:

1. Когда нет явного `self`:

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

2. Когда функция неограниченно обобщенная (generic):

```
print(x)
```

3. Когда синтаксис функции является частью нотации определенного сообщества:

```
sin(x)
```

- **Следуйте соглашениям использования определенного регистра.** Имена типов и протоколов - "верблюжий" стиль с прописными (заглавными) первыми буквами UpperCamelCase, всё остальное - "верблюжий" стиль со строчными первыми буквами lowerCamelCase.

Акронимы и сокращения по первым буквам, которые часто представляются полностью прописными буквами в американском английском, должны одинаково использовать верхний/ нижний регистр в соответствии с соглашениями:

```
var utf8Bytes: [UTF8.CodeUnit]

var isRepresentableAsASCII = true

var userSMTPServer: SecureSMTPServer
```

Другие акронимы должны считаться обычными словами:

```
var radarDetector: RadarScanner

var enjoysScubaDiving = true
```

- **Методы могут разделять общее имя,** когда они разделяют общее значение или когда они функционируют в четко обозначенных доменах.

  Например, рекомендуются следующие методы, так как они делают, в основном, одно и то же:

```
extension Shape {
/// Возвращает`true`, если `other` находится внутри области `self`.
 func contains(_ other: Point) -> Bool { ... }

///Возвращает`true`, если `other` находится полностью внутри области `self`.
 func contains(_ other: Shape) -> Bool { ... }

/// Возвращает`true`, если `other` находится внутри области `self`.
 func contains(_ other: LineSegment) -> Bool { ... }
}
```

Так как геометрические типы и коллекции - это разные области, то допустимо иметь в одной программе:

```
extension Collection where Element : Equatable {

/// Возвращает`true`, если содержит элемент, равный `sought`.
 func contains(_ sought: Element) -> Bool { ... }

}
```

Однако следующие `index` методы имеют разный смысл и должны быть названы по разному:

```
extension Database {
/// Перестраивает поисковый индекс в базе данных
func index() { ... }

/// Возвращает `n`-ую строку в заданной таблице.
func index(_ n: Int, inTable: TableID) -> TableRow { ... }
}
```

В заключении, избегайте "перегрузку возвращаемого значения", потому что это вызывает противоречие при "выводе типа из контекста":

```
extension Box {
/// Возвращает `Int`, сохраненное в `self`, и
/// `nil`, если сохранить не удалось.
func value() -> Int? { ... }

/// Возвращает `String`, сохраненное в `self`, и
/// `nil`, если сохранить не удалось.
func value() -> String? { ... }
}
```

2. Параметры

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

- **Выбирайте имена параметров (внутреннее имя аргумента), которые будут одновременно служить и целям документации.** Хотя имена параметров не появляются при использовании функции или метода, они играют важную объяснительную роль.

  Выбирайте эти имена так, чтобы сделать документацию легко читаемой. Например, следующие имена делают документацию легко читаемой:

```
/// Возвращает `Array`, содержащий элементы `self`,
/// которые удовлетворяют предикату `predicate`.
 func filter(_ predicate: (Element) -> Bool) -> [Generator.Element]

/// Заменяет заданный поддиапазон `subRange` элементов новыми`newElements`.
mutating func replaceRange(_ subRange: Range, with newElements: [E])
```

  Нижеприведенные методы, однако, делают документацию странной и грамматически не верной:

```
/// Возвращает`Array`, содержащий элементы `self`,
/// которые удовлетворяют условию `includedInResult`.
func filter(_ includedInResult: (Element) -> Bool) -> [Generator.Element]

/// Заменяет диапазон элементов, обозначенный `r`, с
/// содержимым с `with`.
mutating func replaceRange(_ r: Range, with: [E])
```

- **Используйте преимущества параметров по умолчанию, если это упрощает использование метода или функции в наиболее общих случаях.** Любой параметр с единственным часто используемым значением является кандидатом параметра со значением по умолчанию.

  Значения аргументов по-умолчанию улучшают читаемость, скрывая излишнюю информацию. Например:

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

  Может существенно упростить вызов функции:

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

Предпочтительное использовать значения аргументов по-умолчанию, чем целое семейство методов, потому что они налагают меньшую когнитивную нагрузку для того, кто пробует понять API.

```
extension String {
/// ...description...

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

Пример сверху может быть не таким простым, но в любом случае он проще, чем это:

```
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
}
```

Каждый член семейства методов нуждается в отдельном документировании и отдельном понимании/восприятии пользователем. Чтобы выбрать между ними, пользователь должен понять их **все,** и учитывать некоторые случайные сюрпризы, например: что `foo(bar: nil)` и `foo()` не всегда синонимы. Это заставляет пользователя утомительно выискивать мельчайшие различия в по большей части идентичной документации.
Один метод с параметрами по умолчанию снабжает вас более продвинутым инструментом программирования.

- **Стремитесь разместить параметры со значениями по умолчанию ближе к концу списка параметров.** Параметры без значений по умолчанию обычно более существенны для семантики (смысла) метода, и обеспечивают устойчивый начальный шаблон применения метода

3. Метки аргументов (внешние имена аргументов)

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

- **Избегайте меток, когда аргументы нельзя различить, например,** `min(number1, number2)`, `zip(sequence1, sequence2).`

- **В инициализаторах, которые выполняют изменение типа с сохранением значений, избегайте метки для первого аргумента, например,** `Int64(someUInt32)`

  Первый аргумент всегда должен быть источником преобразования.

```
 extension String {
// Преобразует `x` в его текстовое представление в заданном radix
    init(_ x: BigInt, radix: Int = 10) ← Заметьте начальный символ "подчеркивания"
}

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

В "сужающих" преобразованиях типов рекомендуется метка, описывающее "сужение".

```
extension UInt32 {
       /// Создает экземпляр, имеющий заданное значение `value`.
       init(_ value: Int16) ← Расширение, так что метки нет
       /// Создает экземпляр, имеющий младшие 32 бита источника `source`.
       init(truncating source: UInt64)
       /// Создает экземпляр, имеющий ближайшее представительное
       /// приближение `valueToApproximate`.
       init(saturating valueToApproximate: UInt64)
}
```

Сохранение значения при преобразовании типов - это [мономорфизм](), т.е. каждое различное значение источника приводит к различному значению результата. Например, преобразование из `Int8` в `Int64` - это преобразование с сохранением значения, потому что каждое различное значение `Int8` преобразуется в различное значение `Int64`. Преобразование в обратную сторону уже не является преобразованием с сохранением значения: `Int64` имеет больше возможных значений, чем можно представить с помощью `Int8`.
Замечание: возможность восстанавливать первоначальное значение не зависит от того, является ли преобразование преобразованием с сохранением значений.

- **Когда первый аргумент образует часть предшествующей фразы с предлогом, то дайте этому аргументу метку.** Метку аргумента стоит начинать с предлога, например, `x.removeBoxes(havingLength: 12)`.

  Исключение составляет случай, когда первые два аргумента представляют собой части одной и той же абстракции.

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

  В таких случаях, давайте аргументам метку после предлога, чтобы сохранить абстракцию понятной:

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

- **В других случаях, если первый аргумент образует часть грамматической фразы, пропустите его метку,** добавляя любые предшествующие слова к базовому имени, например, `x.addSubview(y)`

  Эта рекомендация подразумевает, что если первый аргумент не составляет часть грамматической фразы, он должен иметь метку.

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

  Заметьте, что важно, чтобы фраза передавала правильный смысл. Следующий код будет грамматически правильным, но будет выражать смысловую ошибку:

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

  Также заметьте, что аргументы со значениями по умолчанию могут быть опущены, и если в этом случае они не составляют часть грамматической фразы, они всегда должны иметь метку.

- **Давайте метку всем остальным аргументам.**

IV. Особые указания.

- **Давайте названия параметрам замыкания closure и членам кортежей tuple**, когда они появляются в вашем API

  Эти имена несут объяснительную функцию и на них можно ссылаться в документируемых комментариях, а также обеспечить осмысленный доступ к элементам кортежа.

```
/// Гарантирует, что память с уникальной ссылкой содержит по крайней мере
/// `requestedCapacity` элементов.
///
/// Если требуется больше памяти, вызывается `allocate` с
/// `byteCount`, равным числу максимально выровненных байтов
///
/// - Возвращает:
///       - reallocated: `true`, если размещен новый блок памяти
///       - capacityChanged: `true`, если  `capacity`была модифицирована.
   mutating func ensureUniqueStorage(
     minimumCapacity requestedCapacity: Int,
     allocate: (byteCount: Int) -> UnsafePointer<Void>
 ) -> (reallocated: Bool, capacityChanged: Bool)
```

Хотя в замыканиях (closures) технически используются метки аргументов, вам следует использовать эти метки в документации, как если бы они были именами параметров. Вызов замыкания (closure) в теле функции будет читаться последовательно с именем функцией, чья фраза начинается с базового имени и не включает первый аргумент:

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

- **С особой осторожностью обходитесь с естественным полиморфизмом** (`Any`, `AnyObject` и другими обобщенными (generic) параметрами, чтобы избежать неоднозначности в множестве перегружаемых функций.

  Например, рассмотрим следующее множество перегружаемых функций:

```
struct Array {
/// Вставляет новый элемент `newElement` в конец `self.endIndex`.
  public mutating func append(_ newElement: Element)

/// Вставляет содержимое массива `newElements` по порядку
/// в конец `self.endIndex`.
  public mutating func append(_ newElements: S)
    where S.Generator.Element == Element
```

```
  }
```

Эти методы принадлежат одному семантическому семейству, и типы аргументов по-началу кажутся абсолютно различными. Однако, если `Element` является `Any`, то один элемент может иметь такой же тип, как и последовательность этих элементов.

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

Чтобы избежать неоднозначности, назовите второй перегружаемый метод более отчетливо:

```
struct Array {
/// Вставляет новый элемент `newElement` в конец `self.endIndex`.
  public mutating func append(_ newElement: Element)

/// Вставляет содержимое (contents of) `newElements` по порядку в
/// конец `self.endIndex`.
  public mutating func append(contentsOf newElements: S)
    where S.Generator.Element == Element
}
```

Заметьте как хорошо новое имя соответствует документирующему комментарию. В этом случае комментарий перенесен автором на API.