

Оглавление

01. Основы	3
Константы и переменные	3
Аннотация типов	3
Название констант и переменных	3
Печать констант и переменных	3
Комментарии	3
Точки с запятой	4
Целые числа	4
Числа с плавающей точкой	4
Строгая типизация и Вывод типов	4
Числовые литералы	4
Преобразования числовых типов	5
Псевдонимы типов	5
Логические типы	5
Кортежи	5
Опциональные типы (опционалы)	6
Инструкция If и Принудительное извлечение	6
Неявно извлеченные опционалы	7
Обработка ошибок	7
Утверждения и предусловия	8
Отладка с помощью утверждений	8
Обеспечение предусловиями	8
02. Базовые операторы	10
Оператор присваивания	10
Арифметические операторы	10
Составные операторы присваивания	10
Операторы сравнения	11
Тернарный условный оператор	11
Оператор объединения по nil	11
Операторы диапазона	11
Логические операторы	12
03. Строки и символы	13
Строковые литералы	13
Многострочные литералы строк	13
Специальные символы в строковых литералах	13
Расширенные разделители строк	14
Работа со строками	14
Работа с символами	14
Интерполяция строк	15
Юникод (Unicode)	15
Расширяемые наборы графем	15
Доступ и изменение строки	16
Подстроки	16
Сравнение строк	17
Юникод представления строк	18
04. Типы коллекций	19
Изменчивость коллекций	19
Массивы	19
Множества	19
Словари	20
05. Циклы For-in	22
Циклы While	22
While	22
Цикл repeat-while	22
Условные инструкции	22
Инструкция if	23
Инструкция switch	23
Операторы передачи управления	24
Оператор Continue	24

Оператор Break	25
Оператор Fallthrough	25
Маркированные инструкции	25
Ранний выход	25
Проверка доступности API	26
06. Функции	27
07. Замыкания	30
08. Перечисления	34
09. Структуры и классы	37
10. Свойства	39
Свойства хранения	39
Ленивое свойство	39
Вычисляемые свойства	40
Наблюдатели свойств	41
Обертки для свойств	41
Свойство типа	43
11. Методы	45
Методы экземпляра	45
Методы типа	46
12. Сабскрипты	47
13. Наследование	48
Доступ к методам, свойствам, индексам суперкласса	48
Переопределения геттеров и сеттеров свойства	48
Переопределение наблюдателей свойства	48
Предотвращение переопределений	48
14. Инициализация	50
Установка начальных значений для свойств хранения	50
Настройка инициализации	50
Дефолтные инициализаторы	51
Делегирование инициализатора для типов значения	52
Наследование и инициализация класса	52
Назначенный и вспомогательный инициализатор	53
Синтаксис назначенных и вспомогательных инициализаторов	53
Делегирование инициализатора для классовых типов	53
Двухфазная инициализация	54
Наследование и переопределение инициализатора	55
Автоматическое наследование инициализатора	55
Назначенные и вспомогательные инициализаторы в действии	56
Проваливающиеся инициализаторы	56
Проваливающиеся инициализаторы для перечислений	57
Проваливающиеся инициализаторы для перечислений с начальными значениями	57
Распространение проваливающегося инициализатора	58
Переопределение проваливающегося инициализатора	59
Проваливающийся инициализатор init!	59
Требуемые инициализаторы	59
Начальное значение свойства в виде функции или замыкания	60
15. Деинициализация	61
16. Опциональная последовательность	62
17. Обработка ошибок	65
Передача ошибки с помощью генерирующей функции	65
Обработка ошибок с использованием do-catch	66
Преобразование ошибок в опциональные значения	67
Запрет на передачу ошибок	67
Установка действий по очистке (Cleanup)	67
18. Согласованность	69
Определение и вызов асинхронных функций	69
Асинхронные последовательности	69
Параллельный вызов асинхронных функций	70

Задачи и группы задач.....	70
Неструктурированный параллелизм.....	71
Отмена задачи.....	71
Акторы.....	71
19. Приведение типов	72
Определение классовой иерархии для приведения типов	72
Проверка типа	72
Понижающее приведение.....	72
Приведение типов для Any и AnyObject	73
20. Вложенные типы	75
21. Расширения	76
Вычисляемые свойства в расширениях	76
Инициализаторы в расширениях	76
Методы в расширениях.....	77
Сабскрипты в расширениях.....	77
Вложенные типы в расширениях	78
22. Непрозрачные типы	79
Различия между типом протокола и непрозрачным типом	80
23. Протоколы	82
Синтаксис протокола	82
Требование свойств	82
Требование методов.....	82
Требования изменяющих методов.....	83
Требование инициализатора.....	83
Протоколы как типы	84
Делегирование	84
Добавление реализации протокола через расширение.....	85
Условное соответствие протоколу.....	86
Принятие протокола через расширение.....	86
Принятие протокола через синтезированную реализацию....	86
Коллекции типов протокола.....	87
Наследование протокола	87
Классовые протоколы	87
Композиция протоколов.....	87
Проверка соответствия протоколу.....	88
Опциональные требования протокола	89
Расширение протоколов	89
Обеспечение реализации по умолчанию (дефолтной реализации).....	90
Добавление ограничений к расширениям протоколов	90
24. Универсальные шаблоны	91
Универсальные функции	91
Параметры типа	91
Именованые параметров типа.....	91
Универсальные типы	91
Расширяем универсальный тип	91
Ограничения типа.....	92
Связанные типы	92
Использование протокола в ограничениях связанного типа и Оговорка where.....	93
Универсальные сабскрипты	94
25. Автоматический подсчет ссылок (ARC)	95
Работа ARC	95

Циклы сильных ссылок между экземплярами классов.....	95
Замена циклов сильных ссылок между экземплярами классов	96
Слабые (weak) ссылки	96
Бесхозные ссылки	98
Бесхозные опциональные ссылки.....	99
Бесхозные ссылки и неявно извлеченные опциональные свойства.....	100
Циклы сильных ссылок в замыканиях	100
Замена циклов сильных ссылок в замыканиях	101
26. Безопасность хранения	103
Конфликт доступа к сквозным параметрам.....	103
Конфликт доступа к self в методах	104
Конфликт доступа к свойствам	104
27. Контроль доступа	106
Модули и исходные файлы	106
Уровни доступа.....	106
Руководящий принцип по выбору уровня доступа	107
Дефолтный уровень доступа.....	107
Уровень доступа для простых однозадачных приложений....	107
Уровень доступа для фреймворка.....	107
Уровни доступа для модуля поэлементного тестирования (unit test target).....	107
Пользовательские типы.....	107
Кортежи типов	108
Типы функций.....	108
Типы перечислений.....	108
Исходные значения и связанные значения.....	108
Вложенные типы	108
Уровень доступа класса и подкласса	109
Константы, переменные, свойства и сабскрипт.....	109
Геттеры и сеттеры.....	109
Инициализаторы	110
Дефолтные инициализаторы	110
Дефолтные почтенные инициализаторы для типов структур.....	110
Протоколы и уровень доступа.....	110
Наследование протокола	110
Соответствие протоколу.....	110
Расширения и уровни доступа	110
Private свойства и методы в расширениях.....	111
Универсальные шаблоны.....	111
Алиасы типов.....	111
28. Продвинутые операторы	112
Побитовые операторы	112
Операторы побитового левого и правого сдвига	113
Поведение побитового сдвига для знаковых целых чисел ...	113
Операторы переполнения	114
Приоритет и ассоциативность.....	115
Операторные функции.....	115
Пользовательские операторы.....	116
Result Builders.....	117

01. Основы

Swift - язык типобезопасный.

Swift включает расширенные типы, которых нет в Objective-C.

Константы и переменные

Константы и переменные связывают со значением определенного типа. Значение константы не может быть изменено после его установки, тогда как переменной может быть установлено другое значение в будущем

Константы объявляются с помощью ключевого слова let, а переменные с помощью var.

Объявление константы	let maximumNumberOfLoginAttempts = 10
Объявление переменной	var currentLoginAttempt = 0
Объявление нескольких констант или нескольких переменных на одной строке	var x = 0.0, y = 0.0, z = 0.0

Аннотация типов

Аннотация типов – это обозначение типа, когда объявляете константу или переменную, чтобы иметь четкое представление о типах значений, которые могут хранить константы или переменные.

Пример аннотации типов	var welcomeMessage: String welcomeMessage = "Hello" var red, green, blue: Double
------------------------	--

Название констант и переменных

Имена констант и переменных не могут содержать пробелы, математические символы, стрелки, приватные (или невалидные) кодовые точки Unicode, а так же символы отрисовки линий или прямоугольников. Так же имена не могут начинаться с цифр, хотя цифры могут быть включены в имя в любом другом месте.

Если вам нужно объявить константу или переменную тем же именем, что и зарезервированное слово Swift, то вы можете воспользоваться обратными кавычками (') написанными вокруг этого слова. Однако старайтесь избегать имен совпадающих с ключевыми словами Swift и используйте такие имена только в тех случаях, когда у вас абсолютно нет выбора.

Пример аннотации типов	var welcomeMessage: String welcomeMessage = "Hello" var red, green, blue: Double
------------------------	--

Печать констант и переменных

Вы можете напечатать текущее значение константы или переменной при помощи функции print(_:separator:terminator:)	print(friendlyWelcome) // Выведет "Bonjour!"
--	---

Swift использует интерполяцию строки для включения имени константы или переменной в качестве плейсхолдера внутри строки, что подсказывает Swift подменить это имя на текущее значение, которое хранится в этой константе или переменной

Пример интерполяции строки	print("Текущее значение friendlyWelcome равно \(friendlyWelcome)") // Выведет "Текущее значение friendlyWelcome равно Bonjour!"
----------------------------	--

Комментарии

Используйте комментарии, чтобы добавить неисполняемый текст в коде, как примечание или напоминание самому себе

Однострочные комментарии	// это комментарий
Многострочные комментарии	/* это тоже комментарий, но написанный на двух строках */
Многострочные комментарии в Swift могут быть вложены в другие многострочные комментарии	/* это начало первого многострочного комментария /* это второго, вложенного многострочного комментария */ это конец первого многострочного комментария */

Точки с запятой

Swift не требует писать точку с запятой (;) после каждого выражения в коде, хотя вы можете делать это, если хотите.

Точки с запятой	<pre>let cat = "?"; print(cat) // Выведет "?"</pre>
-----------------	---

Целые числа

Swift предусматривает знаковые и беззнаковые целые числа в 8, 16, 32 и 64 битном форматах.

В Swift есть дополнительный **тип целого числа** - **Int**, который имеет тот же размер что и разрядность системы:

- На 32-битной платформе, Int того же размера что и Int32;
- На 64-битной платформе, Int того же размера что и Int64;

Swift также предусматривает **беззнаковый тип целого числа** - **UInt**, который имеет тот же размер что и разрядность системы:

- На 32-битной платформе, UInt того же размера что и UInt32
- На 64-битной платформе, UInt того же размера что и UInt64

Доступ к минимальному значению каждого типа целого числа	<pre>let minValue = UInt8.min // minValue равен 0, а его тип UInt8</pre>
Доступ к максимальному значению каждого типа целого числа	<pre>let maxValue = UInt8.max // maxValue равен 255, а его тип UInt8</pre>

Используйте UInt, только когда вам действительно нужен тип беззнакового целого с размером таким же, как разрядность системы. Если это не так, использовать Int предпочтительнее, даже когда известно, что значения будут неотрицательными. Постоянное использование Int для целых чисел способствует совместимости кода, позволяет избежать преобразования между разными типами чисел, и соответствует выводу типа целого числа

Числа с плавающей точкой

Swift предоставляет два знаковых типа с плавающей точкой:

- Double - представляет собой 64-битное число с плавающей точкой (15 десятичных цифр).
- Float - представляет собой 32-битное число с плавающей точкой (6 десятичных цифр).

В случаях, где возможно использование обоих типов, предпочтительным считается **Double**.

Строгая типизация и Вывод типов

Swift - язык со строгой типизацией. В Swift есть автоматический вывод типов.

Автоматический вывод типов	<pre>let meaningOfLife = 42 // meaningOfLife выводится как тип Int let pi = 3.14159 // pi выводится как тип Double let anotherPi = 3 + 0.14159 // anotherPi тоже выводится как тип Double</pre>
----------------------------	---

Числовые литералы

Числовые литералы могут быть написаны как:

- Десятичное число, **без префикса**
- Двоичное число, с префиксом **0b**
- Восьмеричное число, с префиксом **0o**
- Шестнадцатеричное число, с префиксом **0x**

Для десятичных чисел с показателем степени exp, базовое число умножается на 10*(exp):

- 1.25e2 означает 1.25×10^2 , или 125.0.
- 1.25e-2 означает 1.25×10^{-2} , или 0.0125.

Для шестнадцатеричных чисел с показателем степени exp, базовое число умножается на 2*(exp):

- 0xFp2 означает 15×2^2 , или 60.0.
- 0xFp-2 означает 15×2^{-2} , или 3.75.

Целые числа и числа с плавающей точкой могут быть **дополнены нулями** и могут содержать **символы подчеркивания** для увеличения читабельности. Ни один тип форматирования не влияет на базовое значение литерала

Числовые литералы	<pre>let decimalInteger = 17 let binaryInteger = 0b10001 // 17 в двоичной нотации let octalInteger = 0o21 // 17 в восьмеричной нотации let hexadecimalInteger = 0x11 // 17 в шестнадцатеричной нотации</pre>
Все эти числа с плавающей точкой имеют десятичное значение 12.1875	<pre>let decimalDouble = 12.1875 let exponentDouble = 1.21875e1 let hexadecimalDouble = 0xC.3p0</pre>
Форматирование	<pre>let paddedDouble = 000123.456 let oneMillion = 1_000_000 let justOverOneMillion = 1_000_000.000_000_1</pre>

Преобразования числовых типов

Используйте `Int` для всех целочисленных констант и переменных в коде, даже когда они неотрицательны.

Если число не подходит для переменной или константы с определенным размером, выводится ошибка во время компиляции	<pre>let cannotBeNegative: UInt8 = -1 // UInt8 не может хранить отрицательные значения, поэтому эта строка выведет ошибку let tooBig: Int8 = Int8.max + 1 // Int8 не может хранить число больше своего максимального значения, // так что это тоже выведет ошибку</pre>
Функция <code>UInt16(one)</code> для создания нового числа <code>UInt16</code> из значения константы <code>one</code>	<pre>let twoThousand: UInt16 = 2_000 let one: UInt8 = 1 let twoThousandAndOne = twoThousand + UInt16(one)</pre>
Преобразование между целыми числами и числами с плавающей точкой должно происходить явно	<pre>let three = 3 let pointOneFourOneFiveNine = 0.14159 let pi = Double(three) + pointOneFourOneFiveNine // pi равно 3.14159, и для него выведен тип Double</pre>
Тип целого числа может быть инициализирован с помощью <code>Double</code> и <code>Float</code> значений	<pre>let integerPi = Int(pi) // integerPi равен 3, и для него выведен тип Int</pre>
Числа с плавающей точкой всегда урезаются , когда вы используете инициализацию целого числа через этот способ. Это означает, что 4.75 будет 4, а -3.9 будет -3.	

Псевдонимы типов

Псевдонимы типов задают альтернативное имя для существующего типа. Можно задать псевдоним типа с помощью ключевого слова ***typealias***. Псевдонимы типов полезны, когда вы хотите обратиться к существующему типу по имени, которое больше подходит по контексту, например, когда вы работаете с данными определенного размера из внешнего источника

Псевдонимы типов	<pre>typealias AudioSample = UInt16 var maxAmplitudeFound = AudioSample.min // maxAmplitudeFound теперь 0</pre>
------------------	---

Логические типы

В Swift есть простой логический тип ***Bool***. Этот тип называют логическим, потому что он может быть только `true` или `false`. Swift предусматривает две логические константы, `true` и `false` соответственно

Логический тип <code>Bool</code>	<pre>let orangesAreOrange = true let turnipsAreDelicious = false</pre>
----------------------------------	--

Кортежи

Кортежи группируют несколько значений в одно составное значение. Значения внутри кортежа могут быть **любого типа**, то есть, нет необходимости, чтобы они были одного и того же типа.

Кортежи полезны для временной группировки связанных значений. Они не подходят для создания сложных структур данных. Если ваша структура данных, вероятно, будет выходить за рамки временной структуры, то такие вещи лучше проектируйте с помощью классов или структур, вместо кортежей.

Пример кортежа	<pre>let http404Error = (404, "Not Found") // http404Error имеет тип (Int, String), и равен (404, "Not Found")</pre>
Вы можете разложить содержимое кортежа на отдельные константы и переменные	<pre>let (statusCode, statusMessage) = http404Error print("The status code is \(statusCode)") // Выведет "The status code is 404" print("The status message is \(statusMessage)") // Выведет "The status message is Not Found"</pre>

Вы можете игнорировать части кортежа во время разложения с помощью символа подчеркивания (<u>)</u>	<pre>let (justTheStatusCode, _) = http404Error print("The status code is \(justTheStatusCode)") // Выведет "The status code is 404"</pre>
Можно получать доступ к отдельным частям кортежа, используя числовые индексы , начинающиеся с нуля	<pre>print("The status code is \(http404Error.0)") // Выведет "The status code is 404" print("The status message is \(http404Error.1)") // Выведет "The status message is Not Found"</pre>
Вы можете давать имена отдельным элементам кортежа во время объявления	<pre>let http200Status = (statusCode: 200, description: "OK") print("The status code is \(http200Status.statusCode)") // Выведет "The status code is 200" print("The status message is \(http200Status.description)") // Выведет "The status message is OK"</pre>

Опциональные типы (опционалы)

Опциональные типы используются в тех случаях, когда значение может отсутствовать. Опциональный тип подразумевает, что возможны два варианта: или значение есть, и его можно извлечь из опционала, либо его вообще нет.

nil не может быть использован с не опциональными константами и переменными. Если значение константы или переменной при определенных условиях в коде должно когда-нибудь отсутствовать, всегда объявляйте их как опциональное значение соответствующего типа.

nil в Swift не то же самое что nil в Objective-C. В Objective-C nil является указателем на несуществующий объект. В Swift nil не является указателем, а является отсутствием значения определенного типа. Устанавливаться в nil могут опционалы любого типа, а не только типы объектов.

Пример опционала	<pre>let possibleNumber = "123" let convertedNumber = Int(possibleNumber) // для convertedNumber выведен тип "Int?", или "опциональ ый Int"</pre>
Мы можем установить опциональную переменную в состояние отсутствия значения, путем присвоения ему специального значения nil	<pre>var serverResponseCode: Int? = 404 // serverResponseCode содержит реальное Int значение 404 serverResponseCode = nil // serverResponseCode теперь не содержит значения</pre>
Если объявить опциональную переменную без присвоения значения по умолчанию, то переменная автоматически установится в nil для вас	<pre>var surveyAnswer: String? // surveyAnswer автоматически установится в nil</pre>

Инструкция If и Принудительное извлечение

Вы можете использовать инструкцию if, сравнивая опционал с nil, чтобы проверить, содержит ли опционал значение. Это сравнение можно сделать с помощью оператора «равенства» (==) или оператора «неравенства» (!=).

Если вы уверены, что опционал содержит значение, вы можете получить доступ к его значению, добавив **восклицательный знак (!) в конце имени опционала** (Принудительное извлечение значения опционала).

Попытка использовать ! к несуществующему опциональному значению **вызовет runtime ошибку**. Всегда будьте уверены в том, что опционал содержит не-nil значение, перед тем как использовать ! чтобы принудительно извлечь это значение.

Можно использовать привязку опционалов, чтобы выяснить содержит ли опционал значение, и если да, то сделать это значение доступным в качестве временной константы или переменной. Привязка опционалов может использоваться с инструкциями if и while, для проверки значения внутри опционала, и извлечения этого значения в константу или переменную, в рамках одного действия.

Вы можете включать столько опциональных привязок и логических условий в единственную инструкцию if, **сколько вам требуется, разделяя их запятыми**. Если какое-то значение в опциональной привязке равно nil, или любое логическое условие вычисляется как false, то все условия выражения будет считаться false.

Константы и переменные, **созданные через опциональную привязку в инструкции if**, будут доступны только в теле инструкции if. В противоположность этому, константы и переменные, **созданные через инструкцию guard**, доступны в строках кода, следующих за инструкцией guard

Использования if для сравнения опционалов	<pre>if convertedNumber != nil { print("convertedNumber contains some integer value.") } // Выведет "convertedNumber contains some integer value."</pre>
---	--

Принудительное извлечение значения опционала	<pre>if convertedNumber != nil { print("convertedNumber has an integer value of \(convertedNumber!).") } // Выведет "convertedNumber has an integer value of 123."</pre>
Привязка опционалов	<pre>if let actualNumber = Int(possibleNumber) { print("\(possibleNumber) has an integer value of \(actualNumber)") } else { print("\(possibleNumber) could not be converted to an integer") } // Выведет "123" has an integer value of 123</pre>
Несколько опциональных привязок и логических условий в единственной инструкции if	<pre>if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber < secondNumber && secondNumber < 100 { print("\(firstNumber) < \(secondNumber) < 100") } // Выведет "4 < 42 < 100"</pre>

Неявно извлеченные опционалы

Иногда, сразу понятно из структуры программы, что опционал всегда будет иметь значение, после того как это значение впервые было установлено. В этих случаях, очень полезно избавиться от проверки и извлечения значения опционала каждый раз при обращении к нему, потому что можно с уверенностью утверждать, что он постоянно имеет значение. Эти виды опционалов называются **неявно извлеченные опционалы**.

Вместо того, чтобы ставить восклицательный знак (!) после опционала, когда вы его используете, поместите **восклицательный знак (!) после опционала, когда вы его объявляете**. Неявно извлечённые опционалы в основном используются во время инициализации класса.

Если вы попытаетесь получить доступ к неявно извлеченному опционалу когда он не содержит значения - вы получите **runtime ошибку**.

Не используйте неявно извлечённый опционал, если существует вероятность, что в будущем переменная может стать nil. Всегда используйте нормальный тип опционала, если вам нужно проверять на nil значение в течение срока службы переменной.

Неявно извлеченные опционалы	<pre>let possibleString: String? = "An optional string." let forcedString: String = possibleString! // необходим восклицательный знак let assumedString: String! = "An implicitly unwrapped optional string." let implicitString: String = assumedString // восклицательный знак не нужен let optionalString = assumedString // Тип optionalString является "String?" и assumedString не является принудительно извлеченным значением.</pre>
Вы можете проверить не является ли неявно извлеченный опционал nil точно так же как вы проверяете обычный опционал	<pre>if assumedString != nil { print(assumedString!) } // Выведет "An implicitly unwrapped optional string."</pre>
Вы также можете использовать неявно извлеченный опционал с привязкой опционалов, чтобы проверить и извлечь его значение в одном выражении	<pre>if let definiteString = assumedString { print(definiteString) } // Выведет "An implicitly unwrapped optional string."</pre>

Обработка ошибок

Вы используете обработку ошибок в ответ на появление условий возникновения ошибок во время выполнения программы. **Обработка ошибок позволяет определить причину сбоя**, и, при необходимости, передать ошибку в другую часть вашей программы.

Когда функция обнаруживает условие ошибки, она выдает сообщение об ошибке. Тот, кто вызывает функцию, может затем **поймать ошибку и среагировать соответствующим образом**.

Функция сообщает о возможности генерации ошибки, включив ключевое слово **throws** в объявление. Когда вы вызываете функцию, которая может выбросить ошибку, вы добавляете ключевое слово **try** в выражение. Swift автоматически передает ошибки из их текущей области, пока они не будут обработаны условием **catch**.

Выражение do создает область, содержащую объект, которая позволяет ошибкам быть переданными в **одно или несколько** условий catch.

Пример функции которая может сгенерировать ошибку	<pre>func canThrowAnError() throws { // эта функция может сгенерировать ошибку }</pre>
---	--

Вызов функции которая может создать ошибку и обработка данной ошибки	<pre>do { try canThrowAnError() // ошибка не была сгенерирована } catch { // ошибка сгенерирована }</pre>
Несколько условий обработки ошибки	<pre>func makeASandwich() throws { // ... } do { try makeASandwich() eatASandwich() } catch SandwichError.outOfCleanDishes { washDishes() } catch SandwichError.missingIngredients(let ingredients) { buyGroceries(ingredients) }</pre>

Утверждения и предусловия

Утверждения и **предусловия** являются проверками во время исполнения. Вы используете их для того, чтобы убедиться, что какое-либо условие уже выполнено, прежде чем начнется исполнение последующего кода. Если булево значение в утверждении или в предусловии равно **true**, то выполнение кода просто продолжается далее, но если значение равно **false**, то текущее состояние исполнения программы некорректно и выполнение кода останавливается и ваше приложение завершает работу.

Используйте утверждения и предусловия для выражения допущений, которые вы делаете, пока пишете код, таким образом вы будете использовать их в качестве части вашего кода. **Утверждения помогают** находить вам ошибки и некорректные допущения, а **предусловия помогают** обнаружить проблемы в рабочем приложении.

В отличие от Обработки ошибок, утверждения и **предусловия не используются для ожидаемых ошибок**, получив которые ваше приложение может восстановить свою работу. Так как ошибка, полученная через утверждение или предусловие является индикатором некорректной работы программы, то нет возможности отловить эту ошибку в утверждении, которое ее вызвало.

Различие между утверждениями и предусловиями в том, когда они проверяются: утверждения проверяются только в сборках дебаггера, а предусловия проверяются и в сборках дебаггера и продакшн сборках. В продакшн сборках условие внутри утверждения не вычисляется. Это означает, что вы можете использовать сколько угодно утверждений в процессе разработки без влияния на производительность продакшена.

Отладка с помощью утверждений

Утверждения записываются как функция стандартной библиотеки Swift **assert(_:file:line:)**. Вы передаете в эту функцию выражение, которое оценивается как true или false и сообщение, которое должно отображаться, если результат условия будет false.

Если код уже проверяет условие, то вы используете функцию **assertionFailure(_:file:line:)** для индикации того, что утверждение не выполнилось.

Отладка с помощью утверждений	<pre>let age = -3 assert(age >= 0, "Возраст человека не может быть меньше нуля") // это приведет к вызову утверждения, потому что age >= 0, а указанное значение < 0.</pre>
Сообщение утверждения можно пропускать по желанию	<pre>assert(age >= 0)</pre>
Если код уже проверяет условие	<pre>if age > 10 { print("Ты можешь покататься на американских горках и чертовом колесе.") } else if age > 0 { print("Ты можешь покататься на чертовом колесе.") } else { assertionFailure("Возраст человека не может быть отрицательным.") }</pre>

Обеспечение предусловиями

Используйте предусловие везде, где условие потенциально может получить значение false, но для дальнейшего исполнения кода оно определенно должно равняться true. **Например**, используйте предусловие для проверки того, что значение сабскрипта не вышло за границы диапазона или для проверки того, что в функцию было передано корректное значение.

Для использования предусловий вызовите функцию ***precondition***(_:_:file:line:). Вы передаете этой функции выражение, которое вычисляется как true или false и сообщение, которое должно отобразиться, если условие будет иметь значение как false.

Вы так же можете вызвать функцию ***preconditionFailure***(_:_:file:line:) для индикации, что отказ работы уже произошел, например, если сработал дефолтный кейс инструкции switch, когда известно, что все валидные значения должны быть обработаны любым кейсом, кроме дефолтного.

Если код уже проверяет условие	// В реализации сабскрипта... <i>precondition</i> (index > 0, "Индекс должен быть больше нуля.")
--------------------------------	--

Если вы компилируете в режиме -Ounchecked, то предусловия не проверяются. Компилятор предполагает, что предусловия всегда получают значения true, и он оптимизирует ваш код соответствующим образом. **Однако**, функция ***fatalError***(_:file:line:) всегда останавливает исполнение, несмотря на настройки оптимизации.

Вы можете использовать функцию *fatalError* (_:file:line:) во время прототипирования или ранней разработки для создания заглушек для функциональности, которая еще не реализована, ***напугав fatalError ("Unimplemented")*** в качестве реализации заглушки. Поскольку фатальные ошибки никогда не оптимизируются, в отличие от утверждений или предусловий, вы можете быть уверены, что выполнение кода всегда прекратится, если оно встречает реализацию заглушки.

02. Базовые операторы

Оператор — это специальный символ или выражение для проверки, изменения или сложения величин.

- **Унарные операторы** применяются к одной величине (например, -a).
 - **Унарные префиксные операторы** ставятся непосредственно перед величиной (например, !b)
 - **Унарные постфиксные операторы** ставятся сразу за ней (например, c!).
- **Бинарные операторы** применяются к двум величинам (например, 2 + 3) и являются **инфиксными**, так как ставятся между этими величинами.
- **Тернарные операторы** применяются к трем величинам. Как и в языке C, в Swift есть только один такой оператор, а именно — тернарный условный оператор (a ? b : c).

Операнды - величины, к которым применяются операторы.

Оператор присваивания

Оператор присваивания (=) не возвращает значение, что позволяет избежать путаницы с оператором проверки на равенство (==).

Оператор присваивания (a = b) инициализирует или изменяет значение переменной a на значение b	<pre>let b = 10 var a = 5 a = b // теперь a равно 10</pre>
Если левая часть выражения является кортежем с несколькими значениями, его элементам можно присвоить сразу несколько констант или переменных	<pre>let (x, y) = (1, 2) // x равно 1, а y равно 2</pre>
Оператор присваивания в Swift не может возвращать значение	<pre>if x = y { // это неверно, так как x = y не возвращает ника кого значения }</pre>

Арифметические операторы

Арифметические операторы (+, -, *, /, % и т. д.) могут обнаруживать и предотвращать переполнение типа, чтобы числовой переменной нельзя было присвоить слишком большое или слишком маленькое значение.

Swift позволяет делить с остатком (%) числа с плавающей точкой.

Оператор целочисленного деления (a % b) показывает, какое количество b помещается внутри a, и возвращает остаток деления a на b. Оператор целочисленного деления (%) в некоторых языках называется **оператором деления по модулю**. Однако учитывая его действие над отрицательными числами в Swift, этот оператор, строго говоря, **выполняет деление с остатком, а не по модулю**.

Оператор унарного минуса (-) — осуществляет изменение знака числового значения

Оператор унарного плюса (+) просто возвращает исходное значение без каких-либо изменений. Он придает коду единообразие, позволяя зрительно отличать положительные значения от отрицательных

Сложение (+) Вычитание (-) Умножение (*) Деление (/)	<pre>1 + 2 // равно 3 5 - 3 // равно 2 2 * 3 // равно 6 10.0 / 2.5 // равно 4.0</pre>
Оператор сложения служит также для конкатенации, или же склейки, строковых значений (тип String)	<pre>"hello, " + "world" // равно "hello, world"</pre>
Оператор целочисленного деления (a % b)	<pre>9 % 4 // равно 1 -9 % 4 // равно -1</pre>
Оператор унарного минуса (-)	<pre>let three = 3 let minusThree = -three // minusThree равно -3 let plusThree = -minusThree // plusThree равно 3, т. е. "минус минус три"</pre>
Оператор унарного плюса (+)	<pre>let minusSix = -6 let alsoMinusSix = +minusSix // alsoMinusSix равно -6</pre>

Составные операторы присваивания

В Swift имеются составные операторы присваивания, совмещающие простое присваивание (=) с другой операцией.

Составные операторы присваивания **не возвращают значение**.

Оператор присваивания со сложением (+=)	<pre>var a = 1 a += 2 // теперь a равно 3</pre>
---	---

Операторы сравнения

В языке Swift есть также два оператора проверки на **идентичность/тождественность** (`===` и `!==`), определяющие, ссылаются ли два указателя на один и тот же экземпляр объекта.

Кортежи сравниваются слева направо, по одному значению за раз до тех пор, пока операция сравнения не найдет отличия между значениями. Если все значения кортежей попарно равны, то и кортежи так же считаются равными.

Кортежи могут сравниваться, только в том случае, если оператор сравнения можно применить ко всем членам кортежей соответственно.

Стандартная библиотека Swift включает в себя операторы сравнения кортежей, **которые имеют менее семи значений**. Если вам нужны операторы, которые могут сравнивать кортежи с более, чем шестью элементами, то вам нужно реализовать их самостоятельно.

Равно (a == b)	<code>1 == 1</code> // истина, так как 1 равно 1
Не равно (a != b)	<code>2 != 1</code> // истина, так как 2 не равно 1
Больше (a > b)	<code>2 > 1</code> // истина, так как 2 больше чем 1
Меньше (a < b)	<code>1 < 2</code> // истина, так как 1 меньше 2
Больше или равно (a >= b)	<code>1 >= 1</code> // истина, так как 1 больше либо равно 1
Меньше или равно (a <= b)	<code>2 <= 1</code> // ложь, так как 2 не меньше либо равно 1
Сравнение кортежей	<code>(1, "zebra") < (2, "apple")</code> // true, потому что 1 меньше 2, "zebra" и "apple" не сравниваются <code>(3, "apple") < (3, "bird")</code> // true, потому что 3 равно 3, а "apple" меньше чем "bird" <code>(4, "dog") == (4, "dog")</code> // true, потому что 4 равно 4 и "dog" равен "dog"
Сравнение кортежей	<code>("blue", -1) < ("purple", 1)</code> // OK, расценивается как true <code>("blue", false) < ("purple", true)</code> // Ошибка так как < не может применяться к значениям типа Bool

Тернарный условный оператор

Тернарный условный оператор — это специальный оператор из трех частей, имеющий следующий синтаксис: **выражение ? действие1 : действие2**. Он выполняет одно из двух действий в зависимости от того, является ли выражение true или false. Если выражение равно true, оператор выполняет действие1 и возвращает его результат; в противном случае оператор выполняет действие2 и возвращает его результат.

Тернарный условный оператор — это короткая и удобная конструкция для выбора между двумя выражениями. Однако тернарный условный оператор следует применять с **осторожностью**. Избыток таких коротких конструкций иногда делает код трудным для понимания. **В частности**, лучше не использовать несколько тернарных условных операторов в одном составном операторе присваивания.

Тернарный условный оператор	<code>let contentHeight = 40</code> <code>let hasHeader = true</code> <code>let rowHeight = contentHeight + (hasHeader ? 50 : 20)</code> // rowHeight равно 90
-----------------------------	---

Оператор объединения по nil

Оператор объединения по nil (a ?? b) извлекает опционал a, если он содержит значение, или возвращает значение по умолчанию b, если a равно nil. Выражение a может быть только опционалом. Выражение b должно быть такого же типа, что и значение внутри a.

Оператор объединения по nil **является краткой записью следующего кода**: `a != nil ? a! : b`

Тернарный условный оператор	<code>let defaultColorName = "red"</code> <code>var userDefinedColorName: String? // по умолчанию равно nil</code> <code>var colorNameToUse = userDefinedColorName ?? defaultColorName</code> // userDefinedColorName равен nil, поэтому colorNameToUse получит значение по умолчанию — "red" <code>userDefinedColorName = "green"</code> <code>colorNameToUse = userDefinedColorName ?? defaultColorName</code> // userDefinedColorName не равно nil, поэтому colorNameToUse получит значение "green"
-----------------------------	--

Операторы диапазона

Оператор замкнутого диапазона удобно использовать при последовательном переборе значений из некоторого диапазона.

Операторы полузамкнутого диапазона особенно удобны при работе с массивами и другими последовательностями, пронумерованными с нуля.

Оператор замкнутого диапазона (a...b) задает диапазон от a до b, включая сами a и b.	<pre> for index in 1...3 { print("\(index) умножить на 5 будет \(index * 5)") } // 1 умножить на 5 будет 5 // 2 умножить на 5 будет 10 // 3 умножить на 5 будет 15 </pre>
Оператор полузамкнутого диапазона (a..b) задает диапазон от a до b, исключая значение b	<pre> let names = ["Anna", "Alex", "Brian"] let count = names.count for i in 0..<count +="" 1="" 1)="" 2="" 3="" <="" \(i="" \(names[i])")="" alex="" anna="" brian="" person="" pre="" print("person="" {="" }="" будет=""> </count></pre>
Односторонние диапазоны (...a) - диапазон, который продолжается насколько возможно, но только в одну сторону	<pre> for name in names[2...] { print(name) } // Brian // Jack </pre>
	<pre> for name in names[...2] { print(name) } // Anna // Alex // Brian </pre>
	<pre> for name in names[..<2] { print(name) } // Anna // Alex </pre>
	<pre> let range = ...5 range.contains(7) // false range.contains(4) // true range.contains(-1) // true </pre>

Логические операторы

Логические операторы изменяют или комбинируют логические значения типа Boolean — true и false.

Логические операторы Swift && и || являются лево-ассоциированными, что означает, что составные выражения с логическими операторами оценивают в первую очередь выражения слева направо.

Читаемость кода всегда важнее краткости, поэтому желательно ставить круглые скобки везде, где они облегчают понимание

Оператор логического НЕ (!a) инвертирует булево значение — true меняется на false, а false становится true.	<pre> let allowedEntry = false if !allowedEntry { print("ACCESS DENIED") } // Выведет "ACCESS DENIED" </pre>
Оператор логического И (a && b) дает на выходе true тогда и только тогда, когда оба его операнда также равны true.	<pre> let enteredDoorCode = true let passedRetinaScan = false if enteredDoorCode && passedRetinaScan { print("Welcome!") } else { print("ACCESS DENIED") } // Выведет "ACCESS DENIED" </pre>
Оператор логического ИЛИ (a b) будет давать true, если хотя бы один из операндов равен true.	<pre> let hasDoorKey = false let knowsOverridePassword = true if hasDoorKey knowsOverridePassword { print("Welcome!") } else { print("ACCESS DENIED") } // Выведет "Welcome!" </pre>
Комбинирование логических операторов	<pre> if enteredDoorCode && passedRetinaScan hasDoorKey knowsOverridePassword { print("Welcome!") } else { print("ACCESS DENIED") } // Выведет "Welcome!" </pre>
Явное указание круглых скобок	<pre> if (enteredDoorCode && passedRetinaScan) hasDoorKey knowsOverridePassword { print("Welcome!") } else { print("ACCESS DENIED") } // Выведет "Welcome!" </pre>

03. Строки и символы

Строка представляет собой совокупность символов, например "hello, world" или "albatross". Строки в Swift представлены типом **String**.

Тип String в Swift бесшовно шит с классом NSString из Foundation. Если вы работаете с фреймворком Foundation в Cocoa или Cocoa Touch, то весь API NSString доступен для каждого значения типа String создаваемого вами в Swift.

Строковые литералы

Строковый литерал - это фиксированная последовательность текстовых символов, окруженная парой двойных кавычек ("").

Использование строкового литерала как начальное значение для константы или переменной	<pre>let someString = "Some string literal value"</pre>
---	---

Многострочные литералы строк

Литерал многострочной строки - последовательность символов, обернутых в три двойные кавычки (""").

Строка начинается на первой строке после открывающих кавычек ("""), а заканчивается на строке предшествующей закрывающим кавычкам, что означает, что **ни одна из строк ниже ни начинается, ни заканчивается символом переноса строки**.

Если вы хотите использовать символ переноса строки для того, чтобы сделать ваш код более читаемым, но вы не хотите чтобы символ переноса строки отображался в качестве части значения строки, то вам нужно **использовать символ обратного следа (\) в конце этих строк**.

Чтобы создать литерал строки, который **начинается и заканчивается символом возврата каретки (\r)**, напишите пустую строку в самом начале и в конце литерала строки.

Пробел до закрывающей группы двойных кавычек (""") сообщает Swift, сколько пробелов нужно игнорировать в начале каждой строки.

Многострочные литералы строк	<pre>let quotation = """ The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked. "Begin at the beginning," the King said gravely, "and go on till you come to the end; then stop." """</pre>
Использование (\) в конце строк	<pre>let softWrappedQuotation = """ The White Rabbit put on his spectacles. "Where shall I begin, \ please your Majesty?" he asked. "Begin at the beginning," the King said gravely, "and go on \ till you come to the end; then stop." """</pre>
Литерал строки, который начинается и заканчивается символом возврата каретки (\r)	<pre>let lineBreaks = """ This string starts with a line break. It also ends with a line break. """</pre>
Пробел до закрывающей группы двойных кавычек (""")	<pre>let linesWithIndentation = """ Эта строка начинается без пробелов в начале. Эта строка имеет 4 пробела. Эта строка так же начинается без пробелов. """</pre>

Специальные символы в строковых литералах

Строковые литералы могут включать в себя следующие специальные символы:

- экранированные специальные символы:** `\0` (нулевой символ), `\\` (обратный слэш), `\t` (горизонтальная табуляция), `\n` (новая строка), `\r` (возвращение каретки), `\"` (двойные кавычки) и `'` (одиночные кавычки)
- Произвольные скалярные величины Юникода**, записанные в виде `\u{n}`, где n - 1-8 значное шестнадцатеричное число со значением, равным действительной точке кода Юникода.

Чтобы **включить символы """" в многострочную строку**, нужно экранировать хотя бы одну из кавычек.

Примеры специальных символов	<pre>let wiseWords = "\"Imagination is more important than knowledge\" - Einstein" // "Imagination is more important than knowledge" - Einstein let dollarSign = "\u{24}" // \$, Unicode scalar U+0024 let blackHeart = "\u{2665}" // ♥, Unicode scalar U+2665 let sparklingHeart = "\u{1F496}" // , Unicode scalar U+1F496</pre>
Включение символов "" в многострочную строку	<pre>let threeDoubleQuotes = "" Escaping the first quote \" Escaping all three quotes \"\"\" ""</pre>

Расширенные разделители строк

Вы можете поместить строковый литерал внутри расширенного разделителя, чтобы **включить в строку специальные символы, не вызывая эффекта самих символов**. Вы помещаете вашу строку в кавычки (") и оборачиваете ее знаками #.

Например, при печати строкового литерала #\"Line 1\nLine 2\"# выйдет последовательность символов с символом новой строки (\n), а не предложение разбитое на две строки.

Если вам нужен эффект специального символа в строковом литерале, сопоставьте количество знаков (#) в строке после символа экранирования (\). **Например**, если ваша строка #\"Line 1\nLine 2\"#, и вы хотите перенести часть предложения на новую строку, вы можете использовать #\"Line 1\#nLine 2\"# вместо этого. **Аналогично** ####\"Line1\####nLine2\"#### также разывает строку.

расширенные разделители многострочной строке	<pre>let threeMoreDoubleQuotationMarks = #\"\"\" Here are three more double quotes: \"\"\" \"\"\"#</pre>
--	--

Работа со строками

Тип String в Swift является типом значения. Компилятор Swift оптимизирует использование строк, так что фактическое копирование строк происходит только тогда, когда оно действительно необходимо.

Создание пустого String значения	<pre>var emptyString = "" // empty string literal var anotherEmptyString = String() // initializer syntax // обе строки пусты и эквиваленты друг другу</pre>
Проверка пустое ли значение	<pre>if emptyString.isEmpty { print("Nothing to see here") } // Выведет "Nothing to see here"</pre>
Модификация строки	<pre>var variableString = "Horse" variableString += " and carriage" // variableString теперь "Horse and carriage"</pre>

Работа с символами

Тип String в Swift представляет собой **коллекцию значений Character** в указанном порядке

Вы не можете добавить String или Character к уже существующей переменной типа Character, потому что значение типа **Character должно состоять только из одиночного символа**.

Получение доступа к отдельным значениям Character в строке с помощью итерации по этой строке в for-in цикле	<pre>for character in "Dog!🐶" { print(character) } // D // o // g // ! // 🐶</pre>
Создание отдельной Character константы или переменной из односимвольного строкового литерала	<pre>let exclamationMark: Character = "!"</pre>
Создание значения типа String путем передачи массива типа [Character]	<pre>let catCharacters: [Character] = ["C", "a", "t", "!"] let catString = String(catCharacters) print(catString) // Выведет "Cat!"</pre>
Значения типа String могут быть добавлены или конкатенированы с помощью (+)	<pre>let string1 = "hello" let string2 = " there" var welcome = string1 + string2 // welcome равен "hello there"</pre>
Добавление значения типа String к другому, уже существующему значению String, с помощью (+=)	<pre>var instruction = "look over" instruction += string2 // instruction равен "look over there"</pre>
Добавление значения типа Character к переменной типа String, используя метод String append	<pre>let exclamationMark: Character = "!" welcome.append(exclamationMark) // welcome равен "hello there!"</pre>

Конкотинация многострочных литералов	<pre>let badStart = "" one two "" let end = "" three "" print(badStart + end) // Prints two lines: // one // twothree</pre>
--------------------------------------	---

Интерполяция строк

Интерполяция строк - способ создать новое значение типа String из разных констант, переменных, литералов и выражений, включая их значения в строковый литерал.

Выражения, которые вы пишете внутри скобок в интерполируемой строке, **не должны содержать** незранированный обратный слэш, символ перевода строки (\n) или символ возврат каретки (\r). Однако выражения **могут содержать** другие строковые литералы.

Интерполяция строк	<pre>let multiplier = 3 let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)" // message равен "3 times 2.5 is 7.5"</pre>
--------------------	--

Юникод (Unicode)

Юникод является международным стандартом для кодирования, представления и обработки текста в разных системах письма.

String тип в Swift построен из **скалярных значений** (Unicode scalar) Юникода. Скалярная величина Юникода является уникальным **21-разрядным числом** для символа или модификатора, **например**, U+0061 для LATIN SMALL LETTER A ("a").

Скалярная величина Юникода - это любая точка кода в диапазоне **U+0000 до U+D7FF** включительно, или **U+E000 до U+10FFFF** тоже включительно. Скалярные величины Юникода **не включают Юникод суррогатные пары точек кода**, т.е. точки кода в диапазоне **U+D800 до U+DFFF** включительно.

Некоторые скалярные величины содержатся в **резерве** для будущего присваивания

Расширяемые наборы графем

Каждый **экземпляр типа Character** в Swift представляет один расширенный набор графем. **Расширенный набор графем** является последовательностью одного и более скалярных величин Юникода, которые (будучи объединенными) производят один читаемый символ.

Расширенный набор графем может состоять из одного или более скалярных величин Юникода. Это означает, что различные символы, и различное отображение одного и того же символа, **могут потребовать разных объемов памяти для хранения**. Из-за этого, **символы в Swift не занимают одинаковый объем памяти** в строке. В результате этого, количество символов в строке **не может быть подсчитано** без итерации в строке, для определения границ расширенного набора графем. Если вы работаете с особенно длинными значениями строк, имейте ввиду, что свойство **count должно итерировать** все скалярные величины Юникода в строке для того, чтобы определить символы в этой строке. **Количество символов, возвращаемых значением count не всегда совпадает со свойством length у NSString**, которое содержит те же символы. Длина NSString **основывается на числе 16-битовых блоков кода в UTF-16** представлении строки, **а не на количестве** расширенных набора графем внутри строки.

Расширенный набор графем, примеры	<pre>let eAcute: Character = "\u{E9}" // é let combinedEAcute: Character = "\u{65}\u{301}" // e с последующим ´ // eAcute равен é, combinedEAcute равен é let precomposed: Character = "\u{D55C}" let decomposed: Character = "\u{1112}\u{1161}\u{11AB}" //precomposed равен "한", decomposed равен "한"</pre>
Расширенный набор графем позволяет скалярам заключающих символов, заключать другие скаляры Юникода и выглядеть как значение типа Character	<pre>let enclosedEAcute: Character = "\u{E9}\u{20DD}" // enclosedEAcute равен é◌</pre>
Скалярные величины Юникода для региональных символов могут быть объединены в пары для создания одного Character значения	<pre>let regionalIndicatorForUS: Character = "\u{1F1FA}\u{1F1F8}" // regionalIndicatorForUS равен us</pre>

Чтобы получить количество значений Character в строке, используйте count для строки	<pre>let unusualMenagerie = "Коала 🐨, Улитка 🐌, Пингвин 🐧, Верблюд 🐪" print("unusualMenagerie содержит \(unusualMenagerie.count) символов") // Выведет "unusualMenagerie содержит 39 символов"</pre>
Использование расширенных наборов графем для Character значений означает, что конкатенация и модификация не всегда могут повлиять на количество символов в строке.	<pre>var word = "cafe" print("количество символов в слове \(word) равно \(word.count)") // Выведет "количество символов в слове cafe равно 4" word += "\u{301}" // COMBINING ACUTE ACCENT, U+0301 print("количество символов в слове \(word) равно \(word.count)") // Выведет "количество символов в слове café равно 4"</pre>

Доступ и изменение строки

Каждое String значение имеет связанный тип индекса: **String.Index**, что соответствует позиции каждого Character в строке.

Различные символы могут требовать различные объемы памяти для хранения, поэтому для того, чтобы определить, какой Character в определенной позиции, **необходимо итерировать каждую скалярную величину Юникода**, от начала или конца этой строки. По этой причине, **Swift строки не могут быть проиндексированы целочисленными значениями**.

Если String пустая, то startIndex и endIndex равны.

Вы можете использовать свойства startIndex, endIndex и методы index(before:), index(after:) и index(_:offsetBy:) с любым типом, который соответствует **протоколу Collection**. Это включает в себя String, различные типы коллекций, например Array, Dictionary и Set.

Свойство startIndex для доступа позиции первого Character в String.	<pre>let greeting = "Guten Tag!" greeting[greeting.startIndex] // G</pre>
Свойство endIndex — это позиция после последнего символа в String.	<pre>greeting[greeting.index(before: greeting.endIndex)] // !</pre>
Доступ к индексу до и после указанного индекса при помощи методов index(before:) и index(after:) .	<pre>greeting[greeting.index(after: greeting.startIndex)] // u</pre>
Доступ к индексу, расположенного не по соседству с указанным индексом, при помощи метода index(_:offsetBy:)	<pre>let index = greeting.index(greeting.startIndex, offsetBy: 7) greeting[index] // a</pre>
Свойство indices , чтобы создать Range всех индексов, используемых для доступа к отдельным символам строки.	<pre>for index in greeting.indices { print("\(greeting[index]) ", terminator: " ") } // Выведет "G u t e n T a g !"</pre>
Для того, чтобы вставить символ в строку по указанному индексу, используйте insert(_:at:) метод	<pre>var welcome = "hello" welcome.insert("!", at: welcome.endIndex) // welcome теперь равен "hello!"</pre>
Чтобы вставить содержимое другой строки по указанному индексу, используйте метод insert(contentsOf:at:)	<pre>welcome.insert(contentsOf: " there", at: welcome.index(before: : welcome.endIndex)) // welcome теперь равен "hello there!"</pre>
Чтобы удалить символ из строки по указанному индексу используйте remove(at:)	<pre>welcome.remove(at: welcome.index(before: welcome.endIndex)) // welcome теперь равно "hello there"</pre>
Если вы хотите удалить значения по указанному диапазону индексов, используйте метод removeSubrange(_:)	<pre>let range = welcome.index(welcome.endIndex, offsetBy: -6)..< welcome.endIndex welcome.removeSubrange(range)// welcome теперь равно "hello"</pre>

Подстроки

Подстроки в Swift имеют практически те же самые методы, что и строки, что означает, что вы можете работать с подстроками так же как и со строками. Однако, **в отличие от строк**, вы используете подстроки непродолжительное время, пока проводите какие-то манипуляции над строками. Когда вы готовы хранить результат более продолжительное время, **то вы конвертируете подстроку в строку**.

Разница между строками и подстроками в том, что для оптимизации производительности подстрока может использовать часть памяти, используемую для хранения исходной строки или часть памяти, которая используется для хранения другой подстроки. **Исходная строка должна** находиться в памяти до тех пор, пока одна из ее подстрок все еще используется.

И String, и Substring реализуют протокол StringProtocol, что означает, что очень часто бывает удобно для строковых манипуляций принимать значение StringProtocol. Вы можете вызывать такие функции со значением String или Substring.

Вы можете получить подстроку из строки, например, используя сабскрипт или метод типа prefix(_:)	<pre> let greeting = "Hello, world!" let index = greeting.firstIndex(of: ",") ?? greeting.endIndex let beginning = greeting[..<index] // beginning is "Hello" // Конвертируем в строку для хранения более продолжительное время. let newString = String(beginning) </pre>
---	--

Сравнение строк

Swift предусматривает **три способа сравнения текстовых значений**: равенство строк и символов, равенство префиксов, и равенство суффиксов.

Два String значения (или два Character значения) **считаются равными**, если их расширенные наборы графем канонически эквивалентны. **Расширенные наборы графем канонически эквивалентны**, если они имеют один и тот же языковой смысл и внешний вид, даже если они изначально состоят из разных скалярных величин Юникода.

Сравнение строк и символов в Swift **не зависит от локализации**.

Равенство строк и символов проверяется оператором "равенства" (==) и оператором "неравенства" (!=)	<pre> let quotation = "Мы с тобой похожи" let sameQuotation = "Мы с тобой похожи" if quotation == sameQuotation { print("Эти строки считаются равными") } // Выведет "Эти строки считаются равными" </pre>
Равны. Оба этих расширенных набора графем являются допустимыми вариантами представления символа é , и поэтому они считаются канонически эквивалентными	<pre> // "Voulez-vous un café?" используем LATIN SMALL LETTER E WITH ACUTE let eAcuteQuestion = "Voulez-vous un caf\u{E9}?" // "Voulez-vous un café?" используем LATIN SMALL LETTER E и COMBINING ACUTE ACCENT let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?" if eAcuteQuestion == combinedEAcuteQuestion { print("Эти строки считаются равными") } // Выведет "Эти строки считаются равными" </pre>
Не равны. Символы визуально похожи, но имеют разный языковой смысл	<pre> let latinCapitalLetterA: Character = "\u{41}" let cyrillicCapitalLetterA: Character = "\u{0410}" if latinCapitalLetterA != cyrillicCapitalLetterA { print("Эти строки считаются не равными") } // Выведет "Эти строки считаются не равными" </pre>
Чтобы проверить, имеет ли строка определенный строковый префикс или суффикс, вызовите hasPrefix(_:) и hasSuffix(_:) методы, оба из которых принимают единственный аргумент типа String, и возвращают логическое значение	<pre> let romeoAndJuliet = ["Act 1 Scene 1: Verona, A public place", "Act 1 Scene 2: Capulet's mansion", "Act 1 Scene 3: A room in Capulet's mansion", "Act 1 Scene 4: A street outside Capulet's mansion", "Act 1 Scene 5: The Great Hall in Capulet's mansion", "Act 2 Scene 1: Outside Capulet's mansion", "Act 2 Scene 2: Capulet's orchard", "Act 2 Scene 3: Outside Friar Lawrence's cell", "Act 2 Scene 4: A street in Verona", "Act 2 Scene 5: Capulet's mansion", "Act 2 Scene 6: Friar Lawrence's cell"] </pre>
hasPrefix(_:) и hasSuffix(_:) методы используются для символьно-символу канонического эквивалентного сравнения между расширенными наборами графем в каждой строке	<pre> var act1SceneCount = 0 for scene in romeoAndJuliet { if scene.hasPrefix("Act 1 ") { act1SceneCount += 1 } } print("Всего \(act1SceneCount) сцен в Акте 1") // Выведет "Всего 5 сцен в Акте 1" </pre>
	<pre> var mansionCount = 0 var cellCount = 0 for scene in romeoAndJuliet { if scene.hasSuffix("Capulet's mansion") { mansionCount += 1 } else if scene.hasSuffix("Friar Lawrence's cell") { cellCount += 1 } } </pre>


```
print("\(mansionCount) сцен в особняке; \(cellCount) тюремные сцены")
// выводит "6 сцен в особняке; 2 тюремные сцены"
```

Юникод представления строк

Если строка Юникода записывается в текстовый файл или какое-либо другое хранилище, то скалярные величины Юникода в этой строке кодируются в одном из нескольких Юникод-определенных форм кодирования. Сюда включены: **UTF-8 форма кодирования** (которая кодирует строку в 8-битные блоки кода), **UTF-16 форма кодирования** (которая кодирует строку в качестве 16-битных блоков кода), и **UTF-32 форма кодирования** (которая кодирует строку в 32-битные единицы кода).

<

04. Типы коллекций

В Swift есть три основных типа коллекций: Массивы, Множества и Словари.

Массив, Словарь и Множество в Swift реализованы как универсальные коллекции

Изменчивость коллекций

Коллекции могут быть как переменными (var) так и константами (let).

Массивы

Массивы хранят много значений одинакового типа в упорядоченном списке.

Массив в Swift связан с классом Foundation NSArray.

Массивы в Swift всегда начинаются с 0.

Если попытаться получить доступ или изменить значение индекса, который находится за пределами существующих границ массива, то будет ошибка исполнения.

Любые пробелы внутри массива закрываются когда удаляется элемент.

Полная форма массива	<code>var someInts = Array<Int>()</code>
Сокращенная форма массива	<code>var someInts = [Int]()</code>
Создание пустого массива, если контекст уже содержит информацию о типе	<code>someInts = []</code>
Создание массива определенного размера с одинаковыми значениями	<code>var threeDoubles = Array(repeating: 0.0, count: 3)</code> <code>// threeDoubles имеет тип [Double] и равен [0.0, 0.0, 0.0]</code>
Создание массива, путем объединения двух массивов	<code>var sixDoubles = threeDoubles + anotherThreeDoubles</code>
Создание массива через литерал массива	<code>var shoppingList: [String] = ["Eggs", "Milk"]</code>
Возвращает кол-во элементов в массиве	<code>shoppingList.count</code>
Возвращает true или false в зависимости пустой ли массив (true если пустой)	<code>shoppingList.isEmpty</code>
Добавление нового элемента в конец массива	<code>shoppingList.append("Flour")</code>
Добавление нескольких элементов к определенному массиву (+=)	<code>shoppingList += ["Baking Powder"]</code>
Извлечение значения массива с помощью синтаксиса сабскриптов	<code>var firstItem = shoppingList[0]</code>
Изменение значения с помощью синтаксиса сабскриптов	<code>shoppingList[0] = "Six eggs"</code>
Изменение диапазона значений за раз, даже если набор изменяющих значений имеет разную длину, по сравнению с диапазоном который требуется заменить.	<code>shoppingList[4...6] = ["Bananas", "Apples"]</code>
Вставка элемента по заданному индексу внутри массива	<code>shoppingList.insert("Maple Syrup", at: 0)</code>
Удаление элемента массива по заданному индексу, при этом возвращается удаленный элемент (это возвращаемое значение можно игнорировать)	<code>let mapleSyrup = shoppingList.remove(at: 0)</code>
Удаление последнего элемента массива	<code>let apples = shoppingList.removeLast()</code>
Итерация по всему набору значения внутри массива с помощью цикла for-in	<code>for item in shoppingList {</code> <code> print(item)</code> <code>}</code>
Итерация по всему набору значений внутри массива где возвращается кортеж для каждого элемента массива, собрав вместе индекс и значение для этого элемента.	<code>for (index, value) in shoppingList.enumerated() {</code> <code> print("Item \(index + 1): \(value)")</code> <code>}</code>

Множества

Множество хранит различные значения одного типа в виде коллекции в неупорядоченной форме.

Тип Swift Set связан с классом Foundation NSSet.

Тип должен быть хешируемым для того, чтобы мог храниться в множестве. Все базовые типы Swift (Int, String, Double, Bool) являются хешируемыми типами по умолчанию и могут быть использованы в качестве типов значений множества или в качестве типов ключей словаря.

Можно использовать собственный тип в качестве типа значения множества или типа ключа словаря, подписав его под протокол Hashable. Типы, которые подписаны под протокол Hashable должны обеспечивать gettable свойство hashValue. Так как протокол Hashable подписан под протокол Equatable, то подписанные под него типы так же должны предоставлять реализацию оператора равенства ==.

Множество a является надмножеством множества b, если a содержит все элементы b

Множество **b** является подмножеством множества **a**, если все элементы **b** находятся в **a**.

Множества называются разделенными, если у них нет общих элементов.

Полная форма записи множества (у множеств нету сокращенной формы)	<code>var letters = Set<Character>()</code>
Создание пустого множества, если контекст уже содержит информацию о типе	<code>letters = []</code>
Создание множества через литерал массива	<code>var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]</code>
Создание множества через литерал массива, если литерал массива содержит элементы одного типа	<code>var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]</code>
Возвращает кол-во элементов в множества	<code>favoriteGenres.count</code>
Возвращает true или false в зависимости пустое ли множество (true если пустое)	<code>favoriteGenres.isEmpty</code>
Добавление нового элемента в множество	<code>favoriteGenres.insert("Jazz")</code>
Удаление элемента из множества, так же возвращает удаленное значение или nil, если удаляемого элемента нет	<code>if let removedGenre = favoriteGenres.remove("Rock") { print("\(removedGenre)? С меня хватит.") } else { print("Меня это не сильно заботит.") }</code>
Удаление всех объектов в множестве	<code>favoriteGenres.removeAll()</code>
Возвращает true или false в зависимости содержится ли данный элемент в множестве (true если да, содержится)	<code>favoriteGenres.contains("Funk")</code>
Итерация по всему набору значения внутри массива с помощью цикла for-in (итерация по множеству будет идти в беспорядке)	<code>for genre in favoriteGenres { print("\(genre)") }</code>
Итерация по множеству в определенном порядке, где возвращаются элементы коллекции в виде отсортированного массива, по умолчанию используя оператор <.	<code>for genre in favoriteGenres.sorted() { print("\(genre)") }</code>
Создание нового множества состоящего из всех значений обоих множеств.	<code>oddDigits.union(evenDigits)</code>
Создание нового множества из общих значений двух входных множеств.	<code>oddDigits.intersection(evenDigits)</code>
Создание нового множества со значениями не принадлежащих указанному множеству из двух входных.	<code>oddDigits.subtracting(evenDigits)</code>
Создание нового множества из значений, которые не повторяются в двух входных множествах.	<code>oddDigits.symmetricDifference(evenDigits)</code>
Определение все ли значения двух множеств одинаковы (возвращает true или false)	<code>houseAnimals == farmAnimals</code>
Определение все ли значения множества содержатся в указанном множестве (возвращает true или false)	<code>houseAnimals.isSubset(of: farmAnimals)</code>
Определение содержит ли множество все значения указанного множества (возвращает true или false)	<code>farmAnimals.isSuperset(of: houseAnimals)</code>
Определение является ли множество надмножеством, но не равным указанному сету (возвращает true или false)	<code>houseAnimals.isStrictSubset(of: farmAnimals)</code>
Определение является ли множество подмножеством, но не равным указанному сету (возвращает true или false)	<code>farmAnimals.isStrictSuperset(of: houseAnimals)</code>
Определение того, отсутствуют ли общие значения в двух множествах или нет (возвращает true или false)	<code>farmAnimals.isDisjoint(with: cityAnimals)</code>

Словари

Словарь представляет собой контейнер, который хранит несколько значений одного и того же типа. Каждое значение связано с уникальным ключом, который выступает в качестве идентификатора этого значения внутри словаря. Элементы в словаре не имеют определенного порядка.

Тип словаря в Swift имеет связь с классом Foundation NSDictionary

Тип словаря Key должен подчиняться протоколу Hashable, как тип значения множества.

Полная форма словаря	<code>var namesOfIntegers = Dictionary<Int, String>()</code>
Сокращенная форма словаря	<code>var namesOfIntegers = [Int: String]()</code>
Создание пустого словаря, если контекст уже содержит информацию о типе	<code>namesOfIntegers = [:]</code>
Создание словаря через литерал словаря	<code>var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]</code>
Создание словаря через литерал словаря, где выводится тип ключа и тип значения	<code>var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]</code>
Возвращает кол-во элементов в словаре	<code>airports.count</code>
Возвращает true или false в зависимости пустой ли массив (true если пустой)	<code>airports.isEmpty</code>
Добавление нового элемента в словарь через синтаксис индексов или изменение значения уже существующего индекса	<code>airports["LHR"] = "London"</code>
Устанавливает значение для ключа если оно не существует, или обновляет значение, если этот ключ уже существует. Так же возвращает опциональное значение (старое значение для этого ключа, если оно существовало до обновления, либо nil если значение не существовало)	<code>if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") { print("The old value for DUB was \(oldValue).") }</code>
Получение значения из словаря для конкретного ключа, с помощью синтаксиса индексов. Так же возвращает опциональное значение, содержащее существующее значение для этого ключа. В противном случае индекс возвращает nil.	<code>if let airportName = airports["DUB"] { print("The name of the airport is \(airportName).") } else { print("That airport is not in the airports dictionary.") }</code>
Удаление пары ключ-значение из словаря, путем присваивания nil значению для этого ключа	<code>airports["APL"] = nil</code>
Удаление пары ключ-значение если она существует. Так же возвращает значение которое удалили, либо возвращает nil если удаляемое значения не существует.	<code>if let removedValue = airports.removeValue(forKey: "DUB") { print("") } else { print("") }</code>
Итерация по всему набору значения внутри массива с помощью цикла for-in, каждое значение возвращается как кортеж (ключ, значение)	<code>for (airportCode, airportName) in airports { print("\(airportCode): \(airportName)") }</code>
Получение коллекции ключей словаря через обращение к его свойству keys	<code>for airportCode in airports.keys { print("Airport code: \(airportCode)") }</code>
Получение коллекции значений словаря через обращение к его свойствам values:	<code>for airportName in airports.values { print("Airport name: \(airportName)") }</code>
Инициализация нового массива с помощью свойства keys	<code>let airportCodes = [String](airports.keys)</code>
Инициализация нового массива с помощью свойства values	<code>let airportNames = [String](airports.values)</code>
Итерация по ключам словаря в отсортированной последовательности	<code>for airportCode in airports.keys.sorted() { print("Airport code: \(airportCode)") }</code>
Итерация по значениям словаря в отсортированной последовательности	<code>for airportName in airports.values.sorted() { print("Airport name: \(airportName)") }</code>

05. Циклы For-in

Цикл for-in используется для итерации по коллекциям элементов

В примерах ниже index является константой, значение которой автоматически устанавливается в начале каждой итерации цикла. Ее объявление неявно происходит в объявлении цикла, без необходимости использования зарезервированного слова let.

Использование цикла for-in для итерации по элементам массива	<pre>let names = ["Anna", "Alex", "Brian", "Jack"] for name in names { print("Hello, \(name)!") }</pre>
Использование цикла for-in для итерации по парам ключ-значение словаря	<pre>let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4] for (animalName, legCount) in numberOfLegs { print("\(animalName)s have \(legCount) legs") }</pre>
Использование цикла for-in с числовыми диапазонами	<pre>for index in 1...5 { print("\(index) умножить на 5 будет \(index * 5)") }</pre>
Символ _ позволяет опустить переменную принимающую номер текущей итерации цикла	<pre>let power = 10 for _ in 1...power { print("\(power)") }</pre>
Использование цикла for-in с полузамкнутыми диапазонами	<pre>let minutes = 60 for tickMark in 0..<minutes (60="" each="" mark="" minute="" pre="" render="" the="" tick="" times)="" {="" }<=""></minutes></pre>
stride(from: to: by:) дает возможность пропустить некоторые итерации цикла (для замкнутых диапазонов)	<pre>for tickMark in stride(from: 0, to: 60, by: 5) { // render the tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55) }</pre>
stride(from: through: by:) дает возможность пропустить некоторые итерации цикла (для полузамкнутых диапазонов)	<pre>for tickMark in stride(from: 3, through: 12, by: 3) { // render the tick mark every 3 hours (3, 6, 9, 12) }</pre>

Циклы While

Цикл while выполняет набор инструкций до тех пор, пока его условие не станет false. Этот вид циклов лучше всего использовать в тех случаях, когда количество итераций до первого входа в цикл неизвестно.

Swift предлагает два вида циклов while:

- while - вычисляет условие выполнения в начале каждой итерации цикла.
- repeat-while - вычисляет условие выполнения в конце каждой итерации цикла.

While

Цикл while начинается с вычисления условия. Если условие истинно, то инструкции в теле цикла будут выполняться до тех пор, пока оно не станет ложным.

Пример цикла while	<pre>var square = 0 while square < 10 { square += 1 }</pre>
---------------------------	--

Цикл repeat-while

Цикл repeat-while, выполняет одну итерацию до того, как происходит проверка условия. Затем цикл продолжает повторяться до тех пор, пока условие не станет false.

Пример цикла repeat-while	<pre>var square = 0 repeat { square += 1 } while square < 10</pre>
----------------------------------	---

Условные инструкции

Swift предоставляет два варианта добавить условные ответвления кода - это при помощи инструкции if и при помощи инструкции switch.

If используется, если условие простое и предусматривает всего несколько вариантов.

найденному совпадению с условием выбирается соответствующая ветка кода для исполнения.

Инструкция if

switch используется для сложных условий, с многими вариантами выбора, и полезен в ситуациях, где по Инструкции if имеет всего одно условие

Инструкция if без оговорки else	<pre>var temperatureInFahrenheit = 30 if temperatureInFahrenheit <= 32 { print("It's very cold.") }</pre>
Инструкция if с оговоркой else	<pre>temperatureInFahrenheit = 40 if temperatureInFahrenheit <= 32 { print("It's very cold.") } else { print("It's not that cold.") }</pre>
Соединение инструкций if между собой	<pre>temperatureInFahrenheit = 90 if temperatureInFahrenheit <= 32 { print("It's very cold.") } else if temperatureInFahrenheit >= 86 { print("It's really warm.") } else { print("It's not that cold.") }</pre>

Инструкция switch

Инструкция switch подразумевает наличие какого-то значения, которое сравнивается с несколькими возможными шаблонами. После того как значение совпало с каким-либо шаблоном, выполняется код, соответствующий ответвлению этого шаблона, и больше сравнений уже не происходит.

Каждая инструкция switch состоит из нескольких возможных случаев или cases, каждый из которых начинается с ключевого слова case.

Каждая инструкция switch должна быть исчерпывающей.

default определяет случай по умолчанию, который включает в себя все значения, которые не были включены в остальные случаи. default всегда идет после всех остальных случаев.

В Swift в инструкции switch отсутствуют провалы через условия. Инструкция switch прекращает выполнение после нахождения первого соответствия с case и выполнения соответствующего кода в ветке, без необходимости явного вызова break.

Тело каждого случая должно включать в себя хотя бы одно исполняемое выражение, иначе будет ошибка компиляции.

Если возможно совпадение сразу с несколькими шаблонами, то исполняется только первое совпадение из остальных.

Кейс в инструкции switch, который содержит только комментарий, при компиляции выдаст ошибку компиляции. Комментарии - это не утверждения, и они не дают возможности игнорировать кейсы. Если вы хотите игнорировать кейс switch, используйте break.

Пример инструкции switch	<pre>let someCharacter: Character = "z" switch someCharacter { case "a": print("The first letter of the alphabet") case "z": print("The last letter of the alphabet") default: print("Some other character") }</pre>
Использование составного кейса	<pre>let anotherCharacter: Character = "a" switch anotherCharacter { case "a", "A": print("The letter A") default: print("Not the letter A") }</pre>

Пример, где значения в кейсах switch могут быть проверены на их вхождение в диапазон	<pre>let approximateCount = 62 switch approximateCount { case 0: naturalCount = "no" case 1..5: naturalCount = "a few" case 5..12: naturalCount = "several" default: naturalCount = "many" }</pre>
Использование кортежей со switch. Каждый элемент кортежа тестируется в case. Можно использовать <code>_</code> для соответствия любой величине	<pre>let somePoint = (1, 1) switch somePoint { case (0, 0): print("\(somePoint) is at the origin") case (_, 0): print("\(somePoint) is on the x-axis") case (0, _): print("\(somePoint) is on the y-axis") case (-2...2, -2...2): print("\(somePoint) is inside the box") default: print("\(somePoint) is outside of the box") }</pre>
Привязка значений с временными константами или переменным внутри тела кейса в инструкции switch. В примере кейс <code>let (x, y)</code> объявляет кортеж двух констант плейсхолдеров, которые могут соответствовать абсолютно любому значению.	<pre>let anotherPoint = (2, 0) switch anotherPoint { case (let x, 0): print("on the x-axis with an x value of \(x)") case (0, let y): print("on the y-axis with a y value of \(y)") case let (x, y): print("somewhere else at \(x), \(y)") }</pre>
Использование дополнительных условий в switch с помощью ключевого слова where . Кейс switch совпадает с текущим значением point только в том случае, если условие оговорки where возвращает true для этого значения.	<pre>let yetAnotherPoint = (1, -1) switch yetAnotherPoint { case let (x, y) where x == y: print("\(x), \(y) is on the line x == y") case let (x, y) where x == -y: print("\(x), \(y) is on the line x == -y") case let (x, y): print("\(x), \(y) is just some arbitrary point") }</pre>
Составные кейсы так же могут включать в себя привязку значения . Все шаблоны составных кейсов должны включать в себя тот же самый набор значений и каждая связка должна быть одного и того же типа (в примере это distance) из всех шаблонов составного кейса.	<pre>let stillAnotherPoint = (9, 0) switch stillAnotherPoint { case (let distance, 0), (0, let distance): print("On an axis, \(distance) from the origin") default: print("Not on an axis") }</pre>

Операторы передачи управления

Операторы передачи управления меняют последовательность исполнения кода, передавая управление от одного фрагмента кода другому. В Swift есть пять операторов передачи управления: `continue`, `break`, `fallthrough`, `return`, `throw`.

Оператор Continue

Оператор `continue` говорит циклу прекратить текущую итерацию и начать новую.

Пример использования оператора continue	<pre>for index in 1..5 { if index == 4 { continue } else { print("\(index)") } }</pre>
--	--

Оператор Break

Оператор break останавливает выполнение всей инструкции управления потоком.

Оператор break может быть использован внутри инструкции switch или внутри цикла.

Пример использования оператора break	<pre>let numberSymbol: Character = "三" var possibleIntegerValue: Int? switch numberSymbol { case "1", "一", "一", "一": possibleIntegerValue = 1 case "2", "二", "二", "二": possibleIntegerValue = 2 case "3", "三", "三", "三": possibleIntegerValue = 3 case "4", "四", "四", "四": possibleIntegerValue = 4 default: break } if let integerValue = possibleIntegerValue { print("") } else { print("") }</pre>
---	--

Оператор Fallthrough

Ключевое слово fallthrough не проверяет условие кейса, оно производит проваливание из конкретного кейса в следующий или в default

Пример использования оператора fallthrough	<pre>let integerToDescribe = 5 var description = "The number \integerToDescribe is" switch integerToDescribe { case 2, 3, 5, 7, 11, 13, 17, 19: description += " a prime number, and also" fallthrough default: description += " an integer." }</pre>
---	---

Маркированные инструкции

Можно маркировать цикл или инструкцию switch маркером инструкций и использовать его вместе с оператором break или оператором continue для предотвращения или продолжения исполнения маркированной инструкции.

Пример использования маркера инструкции	<pre>gameLoop: while square != finalSquare { diceRoll += 1 if diceRoll == 7 { diceRoll = 1 } switch square + diceRoll { case finalSquare: break gameLoop case let newSquare where newSquare > finalSquare: continue gameLoop default: square += diceRoll square += board[square] } }</pre>
--	---

Ранний выход

Инструкция **guard**, как и инструкция if, выполняет выражения в зависимости от логического значения условия.

guard используется, чтобы указать на то, что условие обязательно должно быть true, чтобы код после самой инструкции guard выполнялся.

В отличие от инструкции if, guard всегда имеет код внутри else, который выполняется, когда условие оценивается как false.

Ветка else должна перебросить исполнение кода на выход из этого блока кода, в котором был определен guard (при помощи инструкций return, break, continue, throw или вызвать метод, который ничего не возвращает, например fatalError(_file:line:)).

Проверка доступности API

В Swift есть встроенная поддержка для проверки доступности API, благодаря которой можно быть уверенным, что используются API-интерфейсы, недоступные для данной deployment target.

Компилятор использует информацию о доступности в SDK, чтобы убедиться, что все API-интерфейсы, используемые в коде, доступны для deployment target, указанного в проекте. Swift выдает сообщение об ошибке во время компиляции, если вы пытаетесь использовать недоступный API.

Использование условия доступности (#available) в if или guard инструкциях	<pre>if #available(iOS 10, macOS 10.12, *) { // Используйте API iOS 10 для iOS и используйте API macOS 10.12 на macOS } else { // Используйте более старые API для iOS и macOS }</pre>
--	--

06. Функции

Функции – это самостоятельные фрагменты кода, решающие определенную задачу. Каждой функции присваивается уникальное имя, по которому ее можно идентифицировать и "вызвать" в нужный момент.

Каждая функция в Swift имеет тип (функциональный тип), описывающий тип параметров функции и тип возвращаемого значения.

Параметры функции – это одно или несколько именованных типизированных значений, которые являются входными данными функции.

Параметры функции по умолчанию являются константами. Попытка изменить значение параметра функции из тела этой функции приводит к ошибке компиляции.

Сквозные параметры – это параметры, у которого функция может изменить значение параметра и сохранить его после того как закончится вызов функции. (ключевое слово inout в теле функции, амперсанд (&) перед именем переменной)

Сквозные параметры – это не то же самое, что возвращаемые функцией значения. Сквозные параметры – это альтернативный способ передачи изменений, сделанных внутри функции, за пределы тела этой функции.

Возвращаемый тип – это тип который возвращает функция в качестве результата.

Аргументы функции – это входные значения функции.

Ярлык аргумента - используется при вызове функции.

Имя параметра - используется в теле функции.

Глобальные функции – это функции определенные в глобальном контексте.

Вложенные функции – это функции определенные внутри других функций. Вложенные функции по умолчанию недоступны извне, а вызываются и используются только заключающей функцией. Заключающая функция может также возвращать одну из вложенных, чтобы вложенную функцию можно было использовать за ее пределами.

Вариативный параметр – это параметр, который может иметь сразу несколько значений или не иметь ни одного. Чтобы объявить параметр как вариативный, нужно поставить три точки (...) после его типа. Значения, переданные через вариативный параметр, доступны внутри функции в виде массива соответствующего типа.

Return – возвращаемое значение в теле функции, после возврата значения код в теле функции дальше не исполняется.

Функции, для которых не задан возвращаемый тип, получают специальный тип Void. По сути, это просто пустой кортеж, т. е. кортеж с нулем элементов, который записывается как ().

Пример объявления функции	<pre>func greet(person: String) -> String { let greeting = "Привет, " + person + "!" return greeting }</pre>
Пример вызова функции	<pre>print(greet(person: "Anna"))</pre>
Пример функции без параметров	<pre>func sayHelloWorld() -> String { return "hello, world" } print(sayHelloWorld()) // Выведет "hello, world"</pre>
Пример функции с несколькими входными параметрами	<pre>func greet(person: String, alreadyGreeted: Bool) -> String { if alreadyGreeted { return greetAgain(person: person) } else { return greet(person: person) } } print(greet(person: "Tim", alreadyGreeted: true)) // Выведет "Hello again, Tim!"</pre>
Пример функции, не возвращающие значения	<pre>func greet(person: String) { print("Привет, \(person)!") } greet(person: "Dave") // Выведет "Привет, Dave!"</pre> <p>Если у функции нет выходного значения, то в ее объявлении отсутствует результирующая стрелка (->) и возвращаемый тип</p>

<p>Пример функции, возвращающие несколько значений</p>	<pre>func minMax(array: [Int]) -> (min: Int, max: Int) { var currentMin = array[0] var currentMax = array[0] for value in array[1..<array.count] { if value < currentMin { currentMin = value } else if value > currentMax { currentMax = value } } return (currentMin, currentMax) }</pre>
<p>Опциональный кортеж как возвращаемый тип</p> <p>Кортеж-опционал вида (Int, Int)? - это не то же самое, что кортеж, содержащий опционалы: (Int?, Int?). Кортеж-опционал сам является опционалом, но не обязан состоять из опциональных значений.</p>	<pre>func minMax(array: [Int]) -> (min: Int, max: Int)? { if array.isEmpty { return nil } var currentMin = array[0] var currentMax = array[0] for value in array[1..<array.count] { if value < currentMin { currentMin = value } else if value > currentMax { currentMax = value } } return (currentMin, currentMax) }</pre>
<p>Функции с неявным возвращаемым значением</p> <p>Если тело функции состоит из единственного выражения, то функция неявно возвращает это выражение (можно не писать ключевое слово return).</p>	<pre>func greeting(for person: String) -> String { "Привет, " + person + "!" } print(greeting(for: "Дейв")) // Выведет "Привет, Дейв!"</pre>
<p>Ярлык аргумента (argumentLabel) и Имя параметра (parameterName)</p>	<pre>func someFunction(argumentLabel parameterName: Int) { // В теле функции parameterName относится к значению аргумента }</pre>
<p>Ярлык аргумента можно опустить с помощью символа подчеркивания (_)</p>	<pre>func someFunction(_ firstParameterName: Int) { // В теле функции firstParameterName } someFunction(1, secondParameterName: 2)</pre>
<p>Значения по умолчанию для параметров</p> <p>Параметры, у которых нет дефолтных значений располагаются в начале списка параметров функции до параметров с дефолтными значениями</p>	<pre>func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) { // Если вы пропускаете второй аргумент при вызове функции, то значение parameterWithDefault будет равняться 12 внутри тела функции. } someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) // parameterWithDefault равен 6 someFunction(parameterWithoutDefault: 4) // parameterWithDefault равен 12</pre>
<p>Функция с вариативными параметрами</p> <p>Ярлык аргумента позволяет однозначно определить, какие аргументы передаются вариативному, а какие - параметрам, которые идут после вариативного параметра.</p>	<pre>func arithmeticMean(_ numbers: Double...) -> Double { var total: Double = 0 for number in numbers { total += number } return total / Double(numbers.count) } arithmeticMean(1, 2, 3, 4, 5) arithmeticMean(3, 8.25, 18.75)</pre>
<p>Функции со сквозными параметрами</p>	<pre>func swapTwoInts(_ a: inout Int, _ b: inout Int) { let temporaryA = a a = b b = temporaryA } var someInt = 3 var anotherInt = 107 swapTwoInts(&someInt, &anotherInt)</pre>

Функциональные типы. В примере у обоих функций тип: (Int, Int) -> Int	<pre>func addTwoInts(a: Int, _ b: Int) -> Int { return a + b } func multiplyTwoInts(a: Int, _ b: Int) -> Int { return a * b }</pre>
Пример использования функционального типа	<pre>var mathFunction: (Int, Int) -> Int = addTwoInts</pre>
Функциональные типы как типы параметров	<pre>func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) { print("Result: \\$(mathFunction(a, b))") } printMathResult(addTwoInts, 3, 5) // Выведет "Result: 8"</pre>
Функциональные типы как возвращаемые типы	<pre>func stepForward(_ input: Int) -> Int { return input + 1 } func stepBackward(_ input: Int) -> Int { return input - 1 } func chooseStepFunction(backward: Bool) -> (Int) -> Int { return backward ? stepBackward : stepForward } var currentValue = 3 let moveNearerToZero = chooseStepFunction(backward: currentValue > 0) // moveNearerToZero ссылается на функцию stepBackward()</pre>
Пример вложенной функции	<pre>func chooseStepFunction(backward: Bool) -> (Int) -> Int { func stepForward(input: Int) -> Int { return input + 1 } func stepBackward(input: Int) -> Int { return input - 1 } return backward ? stepBackward : stepForward } let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)</pre>

07. Замыкания

Замыкания - это самодостаточные блоки с определенным функционалом, которые могут быть переданы и использованы в вашем коде.

Замыкания могут **захватывать константы и переменные** из окружающего контекста, в котором оно объявлено. После захвата замыкание может ссылаться или модифицировать значения этих констант и переменных внутри своего тела, даже если область, в которой были объявлены эти константы и переменные уже больше не существует.

Функции и замыкания являются **ссылочными типами**.

Три формы замыкания:

- **Глобальные функции** являются замыканиями, у которых есть имя и которые не захватывают никакие значения.
- **Вложенные функции** являются замыканиями, у которых есть имя и которые могут захватывать значения из включающей их функции.
- **Выражения замыкания** являются безымянными замыканиями, написанные в облегченном синтаксисе, которые могут захватывать значения из их окружающего контекста.

Оптимизированный синтаксис замыканий включает в себя:

- Вывод типа параметра и возврат типа значения из контекста
- Неявные возвращаемые значения однострочных замыканий
- Сокращенные имена параметров
- Синтаксис последующих замыканий

Swift автоматически предоставляет **сокращённые имена** для однострочных замыканий, которые могут быть использованы для обращения к значениям параметров замыкания через имена \$0, \$1, \$2, и так далее.

Функция **sorted(by:)** производит сортировку массива, by: принимает замыкание (например замыкание типа: (String, String) -> Bool), которое определяет, как сортировать массив.

Последующее замыкание - замыкание, которое записано в виде замыкающего выражения вне (и после) круглых скобок вызова функции, даже несмотря на то, что оно все еще является аргументом функции. Последующие замыкания полезны в случаях, когда само замыкание достаточно длинное, и его невозможно записать в одну строку.

Сбегающее замыкание - это замыкание которое было передано в функцию в качестве аргумента и вызывается уже после того, как функция вернула значение. Когда объявляется функция, которая имеет замыкание в качестве одного из параметров, то вы пишете **@escaping** до типа параметра, для того чтобы указать, что замыкание может сбежать. Например если замыкание хранится в переменной, которая была объявлена вне функции, а затем эта переменная была передана в качестве аргумента в функцию, то получается, что замыкание, которое посредством переменной передается в функцию, сбегающее.

Замыкания могут захватывать константы и переменные из окружающего контекста, в котором оно объявлено. После захвата замыкание может ссылаться или модифицировать значения этих констант и переменных внутри своего тела, даже если область, в которой были объявлены эти константы и переменные уже больше не существует.

В качестве оптимизации Swift может захватить и хранить копию значения, если это значение не изменяется самим замыканием, а так же не изменяется после того, как замыкание было создано. Swift также берет на себя управление памятью по утилизации переменных, когда они более не нужны.

Если вы присваиваете замыкание свойству экземпляра класса, и замыкание захватывает этот экземпляр по ссылке на него или его члены, вы создаете **сильные обратные связи между экземпляром и замыканием**.

Обычно замыкание захватывает переменные неявно, просто используя их внутри тела, **но в случае с self вам нужно делать это явно**. Если вы хотите захватить self, напишете self явно, когда используете его, или включите self в лист захвата замыкания. Когда вы пишете self явно, вы явно указываете свое намерение, а так же помогаете сами себе тем, что напоминаете проверить наличие цикла сильных ссылок.

Если self является экземпляром структуры или перечисления, то вы можете всегда ссылаться на self неявно. **Однако**, сбегающие замыкания не могут захватить изменяемую ссылку на self, когда self является экземпляром структуры или перечисления.

Автозамыкания - замыкания, которые автоматически создаются для заключения выражения, которое было передано в качестве аргумента функции. Такие замыкания не принимают никаких аргументов при вызове и возвращают значение выражения, которое заключено внутри нее. Синтаксически вы можете опустить круглые скобки функции вокруг параметров функции, просто записав обычное выражение вместо явного замыкания.

Автозамыкания позволяют вам откладывать вычисления, потому как код внутри них не исполняется, пока вы сами его не запустите. Это полезно для кода, который может иметь сторонние эффекты или просто является дорогим в вычислительном отношении, потому что вы можете контролировать время исполнения этого кода.

Например, функция `assert(condition: message: file: line:)` принимает автозамыкания на место `condition` и `message` параметров. Ее параметр `condition` вычисляется только в сборке дебаггера, а параметр `message` вычисляется, если только `condition` равен `false`.

Слишком частое использование автозамыканий может сделать ваш код сложным для чтения. Контекст и имя функции должны обеспечивать ясность отложенности исполнения кода.

Синтаксис замыкающего выражения { (параметры) -> тип результата in выражения }	<pre>let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"] var reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 })</pre>
Определение типа из контекста тип (String, String) -> Bool выведен из контекста	<pre>reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 })</pre>
Неявные возвращаемые значения из замыканий с одним выражением Так как замыкание состоит из одно выражения, можно опустить <code>return</code>	<pre>reversedNames = names.sorted(by: { s1, s2 in s1 > s2 })</pre>
Сокращенные имена параметров	<pre>reversedNames = names.sorted(by: { \$0 > \$1 })</pre>
Операторные функции Тип <code>String</code> в <code>Swift</code> определяет свою специфичную для строк реализацию оператора больше (<code>></code>) как функции, имеющей два строковых параметра и возвращающей значение типа <code>Bool</code> .	<pre>reversedNames = names.sorted(by: >)</pre>
Пример последующего замыкания	<pre>reversedNames = names.sorted() { \$0 > \$1 }</pre>
Если выражение замыкания является единственным аргументом функции, и вы пишете его используя синтаксис последующего замыкания, то вы можете опустить написание круглых скобок вызова самой функции после ее имени.	<pre>reversedNames = names.sorted { \$0 > \$1 }</pre>
Если функция принимает несколько последующих замыканий , вы можете пропустить ярлык параметра для первого из них, а для остальных уже указать нужно.	<pre>func loadPicture(from server: Server, completion: (Picture) -> Void, onFailure: () -> Void) { //тело функции } loadPicture(from: someServer) { picture in //тело замыкания completion } onFailure: { //тело замыкания onFailure }</pre>
Пример захвата значений Функция <code>incrementer()</code> не имеет ни одного параметра и она ссылается на <code>runningTotal</code> и <code>amount</code> внутри тела функции. Она делает это, захватывая существующие значения от <code>runningTotal</code> и <code>amount</code> из окружающей функции и используя их внутри.	<pre>func makeIncrementer(forIncrement amount: Int) -> () -> Int { var runningTotal = 0 func incrementer() -> Int { runningTotal += amount return runningTotal } return incrementer } let incrementByTen = makeIncrementer(forIncrement: 10) incrementByTen() // возвращает 10 incrementByTen() // возвращает 20 incrementByTen() // возвращает 30 let incrementBySeven = makeIncrementer(forIncrement: 7) incrementBySeven() // возвращает значение 7 incrementByTen() // возвращает 40</pre>

Пример что замыкание это ссылочный тип	<pre>let alsoIncrementByTen = incrementByTen alsoIncrementByTen() //возвращает 50 incrementByTen() //возвращает 60</pre>
Пример сбегающего замыкания Функции, которые выполняют асинхронные операции в завершающем обработчике, который является замыканием. То есть получается, что функция завершает свою работу, после чего вызывается завершающий обработчик.	<pre>var completionHandlers: [() -> Void] = [] func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) { completionHandlers.append(completionHandler) }</pre>
Например, в коде справа замыкание переданное в метод someFunctionWithEscapingClosure(_) ссылается на self явно . А вот замыкание, переданное в метод someFunctionWithNonescapingClosure(_) является несбегающим, что значит, что оно может сослаться на self неявно	<pre>func someFunctionWithNonescapingClosure(closure: () -> Void) { closure() } class SomeClass { var x = 10 func doSomething() { someFunctionWithEscapingClosure { self.x = 100 } someFunctionWithNonescapingClosure { x = 200 } } } let instance = SomeClass() instance.doSomething() print(instance.x) // Выведет "200" completionHandlers.first?() print(instance.x) // Выведет "100"</pre>
Версия doSomething(), которая захватывает self, включая его в лист захвата замыкания , а затем неявно ссылается на него	<pre>class SomeOtherClass { var x = 10 func doSomething() { someFunctionWithEscapingClosure { [self] in x = 100 } someFunctionWithNonescapingClosure { x = 200 } } }</pre>
Вызов функции someFunctionWithEscapitngClosure в примере вызовет ошибку , так как находится замыкание внутри mutable метода, таким образом self так же получается изменяемым (mutable). Ошибка получается из-за того , что мы нарушаем правило, которое гласит, что в структурах сбегающие замыкания не могут захватывать изменяемую ссылку на self	<pre>struct SomeStruct { var x = 10 mutating func doSomething() { someFunctionWithNonescapingClosure { x = 200 } // Ok someFunctionWithEscapingClosure { x = 100 } // Error } }</pre>
Автозамыкания Даже если первый элемент массива customersInLine удаляется кодом внутри замыкания, элемент массива фактически не удаляется до тех пор, пока само замыкание не будет вызвано	<pre>var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"] print(customersInLine.count) // Выведет "5" let customerProvider = { customersInLine.remove(at: 0) } print(customersInLine.count) // Выведет "5" print("Now serving \(customerProvider())!") // Выведет "Now serving Chris!" print(customersInLine.count) // Выведет "4"</pre>
customerProvider является не String, а () -> String, то есть функция не принимает аргументов, но возвращает строку. Вы получите то же самое поведение , когда сделаете это внутри функции	<pre>// customersInLine павен ["Alex", "Ewa", "Barry", "Daniella"] func serve(customer customerProvider: () -> String) { print("Now serving \(customerProvider())!") } serve(customer: { customersInLine.remove(at: 0) }) // Выведет "Now serving Alex!"</pre>
Версия функции serve(customer:) ниже выполняет ту же самую операцию, но	<pre>// customersInLine павен ["Ewa", "Barry", "Daniella"]</pre>

<p>Вместо использования явного замыкания, она использует автозамыкание, поставив маркировку при помощи атрибута @autoclosure. Теперь вы можете вызывать функцию, как будто бы она принимает аргумент String вместо замыкания. Аргумент автоматически преобразуется в замыкание, потому что тип параметра customerProvider имеет атрибут @autoclosure.</p>	<pre>func serve(customer customerProvider: @autoclosure () -> String) { print("Now serving \(customerProvider()!)") } serve(customer: customersInLine.remove(at: 0)) // Выведет "Now serving Ewa!"</pre>
<p>Если вы хотите чтобы автозамыкание могло сбежать, то вам нужно использовать оба атрибута и @autoclosure, и @escaping.</p> <p>Вместо того, чтобы вызывать переданное замыкание в качестве аргумента customer, функция collectCustomerProviders(_) добавляет замыкание к массиву customerProviders. Массив объявлен за пределами функции, что означает, что замыкание в массиве может быть исполнено после того, как функция вернет значение. В результате значение аргумента customerProvider должен иметь "разрешение" на "побег" из зоны видимости функции.</p>	<pre>// customersInLine павен ["Barry", "Daniella"] var customerProviders: [() -> String] = [] func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () -> String) { customerProviders.append(customerProvider) } collectCustomerProviders(customersInLine.remove(at: 0)) collectCustomerProviders(customersInLine.remove(at: 0)) print("Collected \(customerProviders.count) closures.") // Выведет "Collected 2 closures." for customerProvider in customerProviders { print("Now serving \(customerProvider()!)") } // Выведет "Now serving Barry!" // Выведет "Now serving Daniella!"</pre>

08. Перечисления

Перечисления определяют общий тип для группы ассоциативных значений и позволяют работать с этими значениями в типобезопасном режиме.

Перечисления **обладают особенностями**, которые обычно поддерживаются классами, например, вычисляемые свойства, методы экземпляра. Перечисления так же могут объявлять инициализаторы, могут быть расширены и могут соответствовать протоколам.

Перечисления начинаются с ключевого слова **enum**, после которого идет имя перечисления и полное его определение в фигурных скобках:

Значения, объявленные в перечислении, называются **кейсами перечисления**. Используйте ключевое слово **case** для включения нового кейса перечисления.

В Swift кейсам перечисления **не присваиваются** целочисленные значения по умолчанию при их создании. В примере CompassPoint, значения членов north, south, east и west неявно не равны 0, 1, 2, 3. Вместо этого различные члены перечисления по праву полностью самостоятельны, с явно объявленным типом CompassPoint.

Каждое объявление перечисления **объявляет и новый тип**.

Для некоторых перечислений можно получить **коллекцию всех кейсов** перечисления. Нужно лишь написать: **Caseterable** после имени перечисления. Swift предоставляет коллекцию всех кейсов, как свойство **allCases** типа перечисления.

Ассоциативные значения — это значения других типов хранимые вместе со значениями кейсов перечисления. Они позволяют хранить дополнительную пользовательскую информацию вместе со значением кейса и разрешает изменять эту информацию каждый раз как вы используете этот кейс перечисления в вашем коде. Можно в перечислении хранить ассоциативные значения любого необходимого типа, и **типы значений могут отличаться** для каждого члена перечисления. (Перечисления такого типа известны как размеченные объединения, маркированные объединения или варианты в других языках программирования)

Кейсы перечисления могут иметь **начальные значения** (исходные значения), которые всегда одного типа.

Исходные значения - это **не то же самое**, что ассоциативные значения. Исходные значения устанавливаются в качестве дефолтных значений, когда вы в первый раз определяете перечисление в вашем коде. **Исходное значение** для конкретного кейса перечисления **всегда одно и то же**. **Ассоциативные значения** устанавливаются при создании новой константы или переменной, основываясь на одном из кейсов перечисления, и **могут быть разными каждый раз**, когда вы делаете это.

Swift автоматически присваивает перечислениям, которые хранят целочисленные или строковые **неявные исходные значения**. Когда **целые числа** используются в качестве исходных значений, неявное значение для каждого кейса будет на единицу больше, чем в предыдущем кейсе. Если первый кейс не имеет заданного значения, его значение равно 0. Когда **строки** используются в качестве исходных значений, неявное значение для каждого кейса является текстом имени этого кейса.

Если перечисление объявлено вместе с типом исходного значения, то перечисление автоматически получает **инициализатор**, который берет значение типа исходного значения (как параметр **rawValue**) и возвращает либо **член перечисления** либо **nil**. Можно использовать этот **инициализатор**, чтобы попытаться создать новый экземпляр перечисления. Инициализатор исходного значения - проваливающийся инициализатор, потому как не каждое исходное значение будет возвращать кейс перечисления.

Рекурсивные перечисления - это такие перечисления, экземпляры которого являются ассоциативным значением одного или более кейсов перечисления. Вы обозначаете такие кейсы перечисления при помощи ключевого слова **indirect** перед кейсом, что сообщает компилятору о том, что нужен дополнительный слой индирекции.

Рекурсивные функции - самый простой путь работать с данными, которые имеют рекурсивную структуру.

Синтаксис перечислений

```
enum CompassPoint {  
    case north  
    case south  
    case east  
    case west  
}  
  
var directionToHead = CompassPoint.west  
directionToHead = .east
```

Множественные значения члена перечисления могут записываться в одну строку	enum Planet { case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune }
Использование перечислений инструкцией switch	c directionToHead = .south switch directionToHead { case .north: print("Lots of planets have a north") case .south: print("Watch out for penguins") default: print("Some another") }
Итерация по кейсам перечисления	enum Beverage: CaseIterable { case coffee, tea, juice } let numberOfChoices = Beverage.allCases.count for beverage in Beverage.allCases { print(beverage) }
Ассоциативные значения	enum Barcode { case upc(Int, Int, Int, Int) case qrCode(String) } var productBarcode = Barcode.upc(8, 85909, 51226, 3) productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
Исходные значения	enum ASCIIControlCharacter: Character { case tab = "\t" case lineFeed = "\n" case carriageReturn = "\r" }
Неявно установленные исходные значения	enum Planet: Int { case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune } enum CompassPoint: String { case north, south, east, west } let earthsOrder = Planet.earth.rawValue // значение earthsOrder равно 3 let sunsetDirection = CompassPoint.west.rawValue // значение sunsetDirection равно "west"
Инициализация через исходное значение	let possiblePlanet = Planet(rawValue: 7) // possiblePlanet имеет тип Planet? и его значение равно Planet.uranus if let somePlanet = Planet(rawValue: 11) { print("\(somePlanet)") } else { print("No planet") } // Выведет "No planet"
Рекурсивные перечисления Можно написать indirect прямо перед самым перечислением, что обозначает, что все члены перечисления поддерживают индиректность Пример: Перечисление может хранить: простое число, сложение двух выражений, умножение двух выражений. Члены addition и multiplication имеют два ассоциативных значения, которые так же являются арифметическими выражениями. Эти ассоциативные значения делают возможным вложение выражений.	enum Expression { case number(Int) indirect case addition(Expression, Expression) indirect case multiplication(Expression, Expression) } indirect enum Expression { case number(Int) case addition(Expression, Expression) case multiplication(Expression, Expression) } // пример выражения (5 + 4) * 2 let five = Expression.number(5) let four = Expression.number(4) let sum = Expression.addition(five, four) let product = Expression.multiplication(sum, Expression.number(2)) // product равно 18

Пример рекурсивной функции

```
let product = Expression.multiplication(sum, Expression.number(
2))
func evaluate(_ expression: Expression) -> Int {
    switch expression {
    case let .number(value):
        return value
    case let .addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}
print(evaluate(product))
// Выведет "18"
```

09. Структуры и классы

Swift не требует создавать **отдельные файлы** для интерфейсов и реализаций пользовательских классов и структур. В Swift, вы объявляете структуру или класс в одном файле, и внешний интерфейс автоматически становится доступным для использования в другом коде.

Экземпляр класса традиционно называют **объектом**. Тем не менее, классы и структуры в Swift гораздо ближе по функциональности

Общее у классов и структур в Swift:

- Объявлять **свойства** для хранения значений
- Объявлять **методы**, чтобы обеспечить функциональность
- Объявлять **индексы**, чтобы обеспечить доступ к их значениям, через синтаксис индексов
- Объявлять **инициализаторы**, чтобы установить их первоначальное состояние
- Они оба могут быть **расширены**, чтобы расширить их функционал за пределами стандартной реализации
- Они оба могут соответствовать **протоколам**, для обеспечения стандартной функциональности определенного типа

Классы имеют дополнительные возможности, которых нет у структур:

- **Наследование** позволяет одному классу наследовать характеристики другого
- **Приведение типов** позволяет проверить и интерпретировать тип экземпляра класса в процессе выполнения
- **Деинициализаторы** позволяют экземпляру класса освободить любые ресурсы, которые он использовал
- **Подсчет ссылок** допускает более чем одну ссылку на экземпляр класса. Для получения дополнительной информации смотрите Наследование, Приведение типов, Деинициализаторы и Автоматический подсчет ссылок.

Дополнительные возможности поддержки классов связаны с увеличением сложности. **Лучше использовать структуры и перечисления**, потому что их легче понимать. Также не забывайте про классы. На практике - большинство пользовательских типов данных, с которыми вы работаете - это структуры и перечисления.

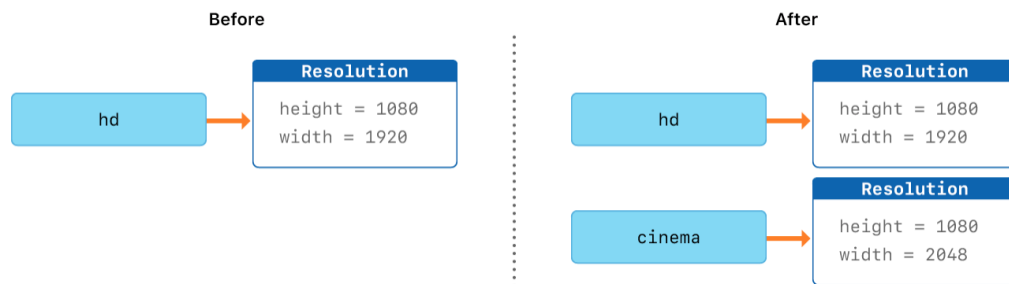
Для **объявления** классов, используйте ключевое слово **class**, а для структур - ключевое слово **struct**.

Доступ к свойствам экземпляра, осуществляется через **точечный синтаксис**. В точечном синтаксисе имя свойства пишется сразу после имени экземпляра, а между ними вписывается точка (.) без пробелов.

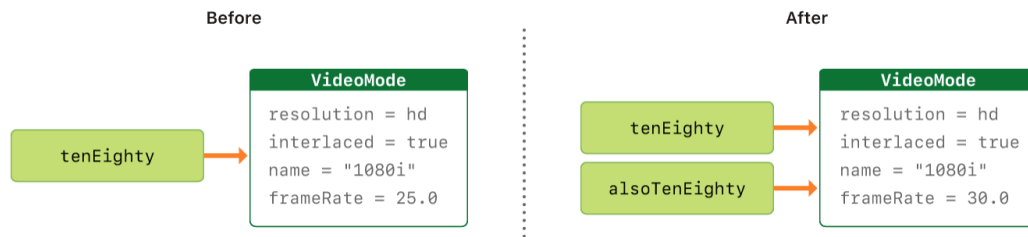
Все **структуры имеют** автоматически генерированный "**поэлементный инициализатор**". Начальные значения для свойств нового экземпляра могут быть переданы поэлементному инициализатору по имени. **Классы не получили** поэлементного инициализатора исходных значений.

Структуры и перечисления - типы значения. **Тип значения** - это тип, значение которого копируется, когда оно присваивается константе или переменной, или когда передается функции.

Коллекции, определенные стандартной библиотекой, такие как массивы, словари и строки, используют **оптимизацию для снижения затрат на копирование**. Вместо того, чтобы немедленно сделать копию, эти коллекции совместно используют память, в которой элементы хранятся между исходным экземпляром и любыми копиями. Если одна из копий коллекции модифицирована, элементы копируются непосредственно перед изменением.



Классы - ссылочный тип. **Ссылочный тип** не копируется, когда его присваивают переменной или константе, или когда его передают функции. Вместо копирования используется ссылка на существующий экземпляр.



Операторы тождественности определяют ссылаются ли две константы или переменные на один и тот же экземпляр класса. В Swift есть два оператора тождественности: **Идентичен (===), Не идентичен (!==)**.

«идентичность» (в виде ===) не имеет в виду «равенство» (в виде ==). **Идентичность или тождественность значит**, что две константы или переменные ссылаются на один и тот же экземпляр класса. **Равенство значит**, что экземпляры равны или эквивалентны в значении в самом обычном понимании «равны».

В Swift константы и переменные, которые ссылаются на экземпляр какого-либо ссылочного типа, **аналогичны указателям C**, но это не прямые указатели на адрес памяти, и они не требуют от вас написания звездочки(*) для индикации того, что вы создаете ссылку. Вместо этого такие ссылки объявляются как другие константы или переменные в Swift

Синтаксис объявления структуры	<pre>struct Resolution { var width = 0 var height = 0 }</pre>
Синтаксис объявления класса	<pre>class VideoMode { var resolution = Resolution() var interlaced = false var frameRate = 0.0 var name: String? }</pre>
Экземпляры класса и структуры	<pre>let someResolution = Resolution() let someVideoMode = VideoMode()</pre>
Доступ к свойствам	<pre>print("\(someResolution.width)") // Выведет "0" someVideoMode.resolution.width = 1280 print("\(someVideoMode.resolution.width)") // Выведет "1280"</pre>
Поэлементные инициализаторы структурных типов	<pre>let vga = Resolution(width: 640, height: 480)</pre>
Структуры и перечисления - типы значения	<pre>let hd = Resolution(width: 1920, height: 1080) var cinema = hd cinema.width = 2048 print("cinema is now \(cinema.width) pixels wide") // Выведет "cinema is now 2048 pixels wide" print("hd is still \(hd.width) pixels wide") // Выведет "hd is still 1920 pixels wide"</pre>
Классы - ссылочный тип	<p>tenEighty и alsoTenEighty объявлены как константы, а не переменные. Однако вы все равно можете менять tenEighty.frameRate и alsoTenEighty.frameRate, потому что значения tenEighty и alsoTenEighty сами по себе не меняются, так как они не «содержат» значение экземпляра VideoMode, они лишь ссылаются на него. Это свойство frameRate лежащего в основе VideoMode, которое меняется, а не значения константы ссылающейся на VideoMode.</p>
Операторы тождественности	<pre>let tenEighty = VideoMode() tenEighty.resolution = hd tenEighty.interlaced = true tenEighty.name = "1080i" tenEighty.frameRate = 25.0 let alsoTenEighty = tenEighty alsoTenEighty.frameRate = 30.0 print("\(tenEighty.frameRate)") // Выведет "30.0" if tenEighty === alsoTenEighty { print("Identity") } // Выведет "Identity"</pre>

10. Свойства

Свойства это вещь которая связывает значения с определённым классом, структурой или перечислением.

Если вы создаете экземпляр структуры и присваиваете его **константе**, то вы **не можете изменять** его свойства, даже если они объявлены как переменные. Такое поведение объясняется тем, что структура является типом значений. Когда экземпляр типа значений отмечен как константа, то все его свойства так же считаются константами. Такое поведение не применимо к классам, так как они являются ссылочным типом.

Глобальные переменные - переменные, которые объявляются вне любой функции, метода, замыкания или контекста типа. **Локальные переменные** - переменные, которые объявляются внутри функции, метода или внутри контекста замыкания.

Так же **можно объявить** вычисляемые переменные и объявить обозреватели для переменных хранения в глобальной или в локальной области своего действия

Глобальные константы и переменные всегда являются **вычисляемыми отложено**, аналогично свойствам ленивого хранения. В отличии от свойств ленивого хранения глобальные константы и переменные не нуждаются в маркере lazy. **Локальные константы и переменные** никогда **не вычисляются отложено**.

Свойства хранения

Свойства хранения содержат значения константы или переменной как часть экземпляра. Свойства хранения **обеспечиваются** только классами или структурами.

Свойства хранения могут быть или **переменными свойствами хранения** (var), или **постоянными свойствами хранения** (let). Можно присвоить **значение по умолчанию** для свойства хранения как часть его определения. Вы так же можете присвоить **начальное значение** для свойства хранения во время его инициализации (это в том числе возможно для постоянных свойств).

Свойства хранения	<pre>struct FixedLengthRange { var firstValue: Int let length: Int } var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3) rangeOfThreeItems.firstValue = 6</pre>
Свойства хранения постоянных экземпляров структуры	<pre>let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4) rangeOfFourItems.firstValue = 6 // вызовет ошибку, даже учитывая что firstValue переменная</pre>

Ленивое свойство

Ленивое свойство хранения (lazy) - свойство, начальное значение которого не вычисляется до первого использования. Всегда объявляйте свойства ленивого хранения как **переменные** (var), потому что ее значение может быть не получено до окончания инициализации. Свойства-константы всегда должны иметь значение до того, как закончится инициализация, следовательно они не могут быть объявлены как свойства ленивого хранения.

Если к свойству обозначенному через **модификатор lazy** обращаются сразу с нескольких потоков одновременно, и если оно еще не было инициализировано, то нет никакой гарантии того, что оно будет инициализировано всего один раз.

Ленивые свойства хранения	<pre>class DataImporter { // Считаем, что классу DataImporter требуется большое количество времени для инициализации var fileName = "data.txt" } class DataManager { lazy var importer = DataImporter() var data = [String]() } let manager = DataManager() manager.data.append("Some data") manager.data.append("Some more data") // экземпляр класса DataImporter для свойства importer еще не создано print(manager.importer.fileName) // экземпляр DataImporter для свойства importer только что был создан и выведется "data.txt"</pre>
Пример: экземпляр DataImporter для свойства importer создается только тогда, когда впервые к нему обращаются	

Вычисляемые свойства

Вычисляемые свойства **обеспечиваются** классами, структурами или перечислениями. Вычисляемые свойства всегда **объявляются** как переменные свойства.

Вычисляемые свойства вычисляют значения, а не хранят их (вместо этого они предоставляют **getter** и **опциональный setter** для получения и установки других свойств косвенно). **Блок get** вызывается при вызове значения свойства, **блок set** вызывается при изменении значения свойства.

Если **setter** вычисляемого свойства не определяет имени для нового значения, то используется имя по умолчанию **newValue**

Если все тело **getter** представляет собой одно выражение, геттер **неявно возвращает** это выражение. Пропуск ключевого слова **return** в геттере работает аналогично пропуску ключевого слова **return** в функциях.

Вычисляемое свойство имеющее геттер, но не имеющее сеттера известно так же как **вычисляемое свойство только для чтения**. Такое вычисляемое свойство только для чтения возвращает значение, но не может изменить свое текущее значение.

Вы должны **объявлять вычисляемые свойства**, включая вычисляемые свойства для чтения, как переменные свойства с ключевым словом **var**, потому что их значение не фиксировано. Ключевое слово **let** используется только для константных свойств, значение которых не может меняться, после того как было установлено как часть инициализации экземпляра.

Вычисляемые свойства	<pre>struct Rect { var x = 0.0 var width = 0.0 var center: Double { get { let centerX = x + (width / 2) return centerX } set(newCenter) { x = newCenter - (width / 2) } } } var square = Rect(x: 0.0, width: 10.0) let initialSquareCenter = square.center // срабатывает get // initialSquareCenter равно 5.0 square.center = 12.0 // срабатывает set, x равно 7.0 print("\(square.x)") // срабатывает get, print выведет "(12.0)"</pre>
Сокращенный вариант объявления сеттера (вычисляемые свойства)	<pre>struct Rect { var x = 0.0 var width = 0.0 var center: Double { get { let centerX = x + (width / 2) return centerX } set { x = newValue - (width / 2) } } }</pre>
Сокращенный вариант объявления геттера (вычисляемые свойства)	<pre>struct Rect { var x = 0.0 var width = 0.0 var center: Double { get { x + (width / 2) } set { x = newValue - (width / 2) } } }</pre>

Вычисляемые свойства только для чтения

Можно упростить объявление вычисляемых свойств только для чтения, удаляя ключевое слово `get` и его скобки

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    } // это get
}

let a = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(a.volume)")
// Выведет "the volume of fourByFiveByTwo is 40.0"
```

Наблюдатели свойств

Наблюдатели свойств это вещь которая предназначена для отслеживания изменений по значению свойства, и которые могут вызывать пользовательское действие, по случаю изменения или присвоения значения свойству.

Наблюдатели свойств **вызываются** каждый раз, как устанавливается значение свойству, даже если устанавливаемое значение не отличается от старого.

Можно **добавить наблюдателей** в следующие места:

- Свойства хранения, которые вы определяете
- Свойства хранения, которые вы наследуете
- Вычисляемые свойства, которые вы наследуете

Для наследуемых свойств вы добавляете наблюдателей свойства, **переопределяя** свойство в подклассе. Для определяемого вычисляемого свойства, используйте **сеттер** для наблюдения и реакции на изменения значения свойства, вместо того, чтобы пытаться создавать наблюдатель.

Два наблюдателя свойства:

- **willSet** вызывается прямо перед сохранением значения. В наблюдатель `willSet` передается новое значение свойства как константный параметр (по умолчанию **newValue**)
- **didSet** вызывается сразу после сохранения значения. В `didSet` будет передан параметр-константа, содержащий старое значение свойства (по умолчанию **oldValue**)

Наблюдатели willSet и didSet суперкласса вызываются, когда свойство устанавливается в инициализаторе подкласса. Они не вызываются в то время, пока класс устанавливает свои собственные свойства, до того, пока не будет вызван инициализатор суперкласса.

Если вы **передаете свойство**, имеющее наблюдателей, в функцию в качестве сквозного параметра, то наблюдатели `willSet` и `didSet` всегда вызываются. Это происходит из-за модели памяти копирования сору-ин сору-аут для сквозных параметров: Значение всегда записывается обратно в свойство в конце функции.

Пример наблюдателя свойства

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("Вот-вот значение будет равно \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue {
                print("Добавлено \(totalSteps - oldValue) шагов")
            }
        }
    }
}

let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// Вот-вот значение будет равно 200
// Добавлено 200 шагов
stepCounter.totalSteps = 360
// Вот-вот значение будет равно 360
// Добавлено 160 шагов
```

Обертки для свойств

Обертка свойства добавляет слой разделения между кодом, который определяет как свойство хранится и кодом, который определяет само свойство. Например, если у вас есть свойства, которые предоставляют потокобезопасную проверку или просто хранят данные в базе данных, то вы должны писать сервисный код для каждого свойства.

Для того, чтобы **определить** обертку, вы создаете структуру, перечисление или класс, который определяет свойство **wrappedValue**.

Вы применяете обертку для свойства написав имя обертки перед свойством в виде атрибута.

Для установки начального значения или другой настройки обертка свойств должна добавить **инициализатор**.

Когда вы пишете **аргументы в скобках после настраиваемого атрибута**, Swift использует инициализатор, который принимает эти аргументы, для настройки обертки.

В дополнение к обернутому значению обертка свойства может предоставлять дополнительные функциональные возможности, определяя **проецируемое значение**. Имя проецируемого значения такое же, как и значение в обертке, за исключением того, что оно начинается со знака доллара (\$). Поскольку ваш код не может определять свойства, начинающиеся с символа \$, проецируемое значение никогда не влияет на свойства, которые вы определяете.

Обертка, которая должна предоставлять больше информации, **может вернуть экземпляр** какого-либо другого типа данных или **может вернуть self**, чтобы предоставить экземпляр обертки в качестве его проецируемого значения.

Когда вы получаете доступ к проецируемому значению из кода, который является частью типа, например, для метода получения свойства или метода экземпляра, **вы можете опустить self**. перед именем свойства, как при доступе к другим свойствам.

Пример определения структуры, которая определяет как свойство хранится	<pre>@propertyWrapper struct TwelveOrLess { private var number = 0 var wrappedValue: Int { get { return number } set { number = min(newValue, 12) } } }</pre>
Применение обертки для свойства	<pre>struct SmallRectangle { @TwelveOrLess var height: Int @TwelveOrLess var width: Int } var rectangle = SmallRectangle() print(rectangle.height) // Выведет "0" rectangle.height = 10 print(rectangle.height) // Выведет "10" rectangle.height = 24 print(rectangle.height) // Выведет "12"</pre>
Установка исходных значений для оберток свойств через инициализаторы Пример: Определения структуры, которая определяет как свойство хранится и определяет 3 вида инициализаторов	<pre>@propertyWrapper struct SmallNumber { private var maximum: Int private var number: Int var wrappedValue: Int { get { return number } set { number = min(newValue, maximum) } } init() { maximum = 12 number = 0 } init(wrappedValue: Int) { maximum = 12 number = min(wrappedValue, maximum) } init(wrappedValue: Int, maximum: Int) { self.maximum = maximum number = min(wrappedValue, maximum) } }</pre>
Пример: Когда вы применяете обертку к свойству и не указываете начальное значение, Swift использует инициализатор init() для настройки обертки	<pre>struct ZeroRectangle { @SmallNumber var height: Int @SmallNumber var width: Int } var zeroRectangle = ZeroRectangle() print(zeroRectangle.height, zeroRectangle.width) // Выведет "0 0"</pre>

<p>Пример: Когда вы указываете начальное значение для свойства, Swift использует инициализатор <code>init(wrappedValue :)</code> для настройки обертки</p>	<pre>struct UnitRectangle { @SmallNumber var height: Int = 1 @SmallNumber var width: Int = 1 } var unitRectangle = UnitRectangle() print(unitRectangle.height, unitRectangle.width) // Выведет "1 1"</pre>
<p>Пример: Когда вы пишете аргументы в скобках после настраиваемого атрибута, Swift использует инициализатор, который принимает эти аргументы, для настройки обертки. Swift использует инициализатор <code>init(wrappedValue: maximum :)</code></p>	<pre>struct NarrowRectangle { @SmallNumber(wrappedValue: 2, maximum: 5) var height: Int @SmallNumber(wrappedValue: 3, maximum: 4) var width: Int } var narrowRectangle = NarrowRectangle() print(narrowRectangle.height, narrowRectangle.width) // Выведет "2 3" narrowRectangle.height = 100 narrowRectangle.width = 100 print(narrowRectangle.height, narrowRectangle.width) // Выведет "5 4"</pre>
<p>Пример определения структуры (обертки свойства) определяющая проецированное значение</p>	<pre>@propertyWrapper struct SmallNumber { private var number = 0 var projectedValue = false var wrappedValue: Int { get { return number } set { if newValue > 12 { number = 12 projectedValue = true } else { number = newValue projectedValue = false } } } }</pre>
<p>Применение обертки для свойства с вызовом проецированного значения</p>	<pre>struct SomeStructure { @SmallNumber var someNumber: Int } var someStructure = SomeStructure() someStructure.someNumber = 4 print(someStructure.\$someNumber) // Выведет "false" someStructure.someNumber = 55 print(someStructure.\$someNumber) // Выведет "true"</pre>
<p>Пример: Код в <code>resize(to :)</code> обращается к высоте и ширине, используя их обертку свойств. Если вы вызываете <code>resize(to: .large)</code>, регистр переключателя для <code>.large</code> устанавливает высоту и ширину прямоугольника равными 100. Обертка предотвращает, чтобы значение этих свойств было больше 12, и устанавливает для проецируемого значения значение <code>true</code>, чтобы зафиксировать тот факт, что он скорректировал их значения. В конце <code>resize(to :)</code> оператор <code>return</code> проверяет <code>\$height</code> и <code>\$width</code>, чтобы определить, изменила ли обертка свойств высоту или ширину.</p>	<pre>enum Size { case small, large } struct SizedRectangle { @SmallNumber var height: Int @SmallNumber var width: Int mutating func resize(to size: Size) -> Bool { switch size { case .small: height = 10 width = 20 case .large: height = 100 width = 100 } return \$height \$width } }</pre>

Свойство типа

Свойства типа – это свойства, которые принадлежат самому типу, а не экземплярам этого типа. Будет всего одна копия этих свойств, и не важно сколько экземпляров вы создадите. Свойства типа полезны при объявлении значений, которые являются универсальными для всех экземпляров конкретного типа

Свойства экземпляров - свойства, которые принадлежат экземпляру конкретного типа. Каждый раз, когда вы создаете экземпляр этого типа, он имеет свои собственные свойства экземпляра, отдельные от другого экземпляра.

Свойства хранения типа могут быть переменными или постоянными. **Вычисляемые свойства типа** всегда объявляются как переменные свойства, таким же способом, как и вычисляемые свойства экземпляра.

В отличие от свойств хранения экземпляра, вы должны **всегда давать** свойствам типов **значение по умолчанию**. Это потому, что тип сам по себе не имеет инициализатора, который мог бы присвоить значение хранимому свойству типа.

Хранимые свойства типа **отложено инициализируются** при первом обращении к ним. Они гарантированно инициализируются только один раз, даже если они доступны сразу для нескольких потоков. Эти свойства не нуждаются в маркировке lazy.

В Swift, свойства типа **записаны как** часть определения типа, внутри его фигурных скобок, и каждое свойство ограничено областью типа, который оно поддерживает.

Чтобы объявить **свойства типа**, используйте ключевое слово **static**. Для **вычисляемых свойств самого класса**, вы должны использовать ключевое слово **class**, чтобы разрешать подклассам переопределение инструкций суперкласса.

Обращение к свойству типа и присваивание ему значения происходит с использованием **точечного синтаксиса**. Однако запрос и присваивание значения происходит в свойстве типа, а не в экземпляре того типа.

Синтаксис (структура)	свойства	типа	<pre>struct SomeStructure { static var storedTypeProperty = "Some value." static var computedTypeProperty: Int { return 1 } }</pre>
Синтаксис (перечисление)	свойства	типа	<pre>enum SomeEnumeration { static var storedTypeProperty = "Some value." static var computedTypeProperty: Int { return 6 } }</pre>
Синтаксис свойства типа (класс)			<pre>class SomeClass { static var storedTypeProperty = "Some value." static var computedTypeProperty: Int { return 27 } class var overridableComputedTypeProperty: Int { return 107 } // ключевое слово class дает разрешение подклассам переопределять overridableComputedTypeProperty }</pre>
Запросы и установка свойств типа			<pre>print(SomeStructure.storedTypeProperty) // Выведет "Some value." SomeStructure.storedTypeProperty = "Another value." print(SomeStructure.storedTypeProperty) // Выведет "Another value." print(SomeEnumeration.computedTypeProperty) // Выведет "6" print(SomeClass.computedTypeProperty) // Выведет "27"</pre>

11. Методы

Методы - это функции, которые связаны с определенным типом. **Классы, структуры и перечисления** - все они могут определять методы экземпляра, которые включают в себя определенные задачи и функциональность для работы с экземпляром данного типа.

Классы, структуры и перечисления так же могут определить **методы типа**, которые связаны с самим типом.

Методы экземпляра

Методы экземпляра являются функциями, которые принадлежат экземплярам конкретного класса, структуры или перечисления. Методы экземпляра имеют абсолютно одинаковый синтаксис как и функции.

Метод экземпляра имеет **неявный доступ** ко всем остальным методам экземпляра и свойствам этого типа. Метод экземпляра может быть **вызван** только для конкретного экземпляра типа, которому он принадлежит. Его **нельзя вызвать** в изоляции, без существующего экземпляра.

Параметры методов могут иметь и **имя аргумента** (для использования внутри функций), и **ярлык аргумента** (для использования при вызове функций)

Каждый экземпляр типа имеет неявное свойство **self**, которое является абсолютным эквивалентом самому экземпляру. Вы используете свойство self для ссылки на текущий экземпляр, внутри методов этого экземпляра. Если вы не пишете self, то Swift полагает, что вы ссылаетесь на свойство или метод текущего экземпляра каждый раз, когда вы используете известное имя свойства или метода внутри метода.

Ключевое слово **mutating** перед словом func используется для того что бы метод мог изменить у структуры и перечисления свои свойства изнутри данного метода. При этом все изменения будут сохранены в исходную структуру, перечисление, когда выполнение метода закончится. Метод так же может присвоить совершенно новый экземпляр для свойства self, и этот новый экземпляр заменит существующий, после того как выполнение метода закончится. По умолчанию, свойства типов значений (структуры и перечисления) не могут быть изменены изнутри методов экземпляра.

Невозможно вызвать изменяющий (**mutating**) метод для константных типов структуры, потому как ее свойства не могут быть изменены, даже если свойства являются переменными

Изменяющие методы могут присваивать полностью новый экземпляр неявному свойству self.

Пример объявления методов класса и вызова методов экземпляра класса	<pre>class Counter { var count = 0 func increment() { count += 1 } func increment(by amount: Int) { count += amount } func reset() { count = 0 } } let counter = Counter() // начальное значение counter равно 0 counter.increment() // теперь значение counter равно 1 counter.increment(by: 5) // теперь значение counter равно 6 counter.reset() // теперь значение counter равно 0</pre>
Пример использования свойства self	<pre>struct Point { var x = 0.0, y = 0.0 func isToTheRightOf(x: Double) -> Bool { return self.x > x // тут self ссылается на свойство Point.x } } let somePoint = Point(x: 4.0, y: 5.0) if somePoint.isToTheRightOf(x: 1.0) { print("Эта точка находится справа от линии, где x == 1.0") } // Выведет "Эта точка находится справа от линии, где x == 1.0"</pre>

Изменение типов значений методами экземпляра	<pre> struct Point { var x = 0.0, y = 0.0 mutating func moveBy(x deltaX: Double, y deltaY: Double) { x += deltaX y += deltaY } } var somePoint = Point(x: 1.0, y: 1.0) somePoint.moveBy(x: 2.0, y: 3.0) print("Сейчас эта точка на \(somePoint.x), \(somePoint.y)") // Выведет "Сейчас эта точка на (3.0, 4.0)" </pre>
Присваивание значения для self внутри изменяющего метода	<pre> struct Point { var x = 0.0, y = 0.0 mutating func moveBy(x deltaX: Double, y deltaY: Double) { self = Point(x: x + deltaX, y: y + deltaY) } } </pre>
Изменяющие методы для перечислений могут установить отдельный член перечисления как неявный параметр self	<pre> enum TriStateSwitch { case off, low, high mutating func next() { switch self { case .off: self = .low case .low: self = .high case .high: self = .off } } } var ovenLight = TriStateSwitch.low ovenLight.next() // ovenLight равен .high ovenLight.next() // ovenLight равен .off </pre>

Методы типа

Метод типа это метод которые вызывается самим типом. Индикатор такого метода - ключевое слово **static**, которое ставится до ключевого слова метода func. Классы так же могут использовать ключевое слово **class**, чтобы разрешать подклассам переопределение инструкций суперкласса этого метода.

Такие методы так же используют **точечный синтаксис**, как и методы экземпляра. Однако эти методы вызываются самим типом, а не экземпляром этого типа.

Внутри тела метода типа **неявное свойство self** ссылается на сам тип, а не на экземпляр этого типа.

Если обобщить, то любое имя метода и свойства, которое вы используете в теле метода типа, будет **ссылаться** на другие методы и свойства на уровне типа. Метод типа может вызвать другой метод типа с иным именем метода, без использования какого-либо префикса имени типа.

Синтаксис метода типа в классе	<pre> class SomeClass { class func someTypeMethod(){ //здесь идет реализация метода } } SomeClass.someTypeMethod() </pre>
Синтаксис метода типа в структуре	<pre> struct LevelTracker { static var highestUnlockedLevel = 1 var currentLevel = 1 static func unlock(_ level: Int) { if level > highestUnlockedLevel { highestUnlockedLevel = level } } static func isUnlocked(_ level: Int) -> Bool { return level <= highestUnlockedLevel } } </pre>

12. Сабскрипты

Классы, структуры и перечисления могут определять **сабскрипты**, которые являются сокращенным вариантом доступа к члену коллекции, списка или последовательности.

Сабскрипты позволяют вам **запрашивать** экземпляры определенного типа, написав одно или несколько значений в квадратных скобках после имени экземпляра.

Синтаксис сабскрипта аналогичный синтаксису метода экземпляра и вычисляемому свойству. Вы пишете определения сабскрипта с помощью ключевого слова **subscript** и указываете один или более входных параметров и возвращаемый тип, точно так же как и в методах экземпляра. В отличии от методов экземпляра, сабскрипты могут быть **read-write** или **read-only** (такое поведение сообщается **геттером** и **сеттером** в точности так же как и в вычисляемых свойствах).

Сабскрипты могут **принимать** любое количество входных параметров, и эти параметры могут быть любого типа. Сабскрипты так же могут **возвращать** любой тип.

Класс или структура могут обеспечить столько сабскриптных **реализаций**, сколько нужно, и подходящий сабскрипт, который будет использоваться, будет выведен, основываясь на типе значения или значений, которые содержатся внутри скобок сабскрипта, в том месте, где этот сабскрипт используется. Определение множественных сабскриптов так же известно как **перегрузка сабскрипта**.

Сабскрипты сущностей, как было сказано выше, являются сабскриптами экземпляров конкретного типа.

Сабскрипты типа - сабскрипты, которые вызываются у самого типа.

Вы указываете **сабскрипт типа** при помощи ключевого слова **static** перед ключевым словом subscript. Классы могут использовать ключевое слово **class** вместо static, чтобы позволить подклассам переопределять реализацию родительского класса этого сабскрипта.

Синтаксис (read-write)	сабскрипта	<pre>subscript(index: Int) -> Int { get { //возвращает надлежащее значение скрипта } set(newValue) { //проводит надлежащие установки } }</pre>
Синтаксис (read-only)	сабскрипта	<pre>subscript(index: Int) -> Int { //возвращает надлежащее значение скрипта }</pre>
Пример сабскрипта		<pre>struct TimesTable { let multiplier: Int subscript(index: Int) -> Int { return multiplier * index } } let threeTimesTable = TimesTable(multiplier: 3) // threeTimesTable = 18</pre>
Опции сабскрипта		<pre>struct SomeStruct { var c: Int subscript(a: Int, b: Int) -> Int { get { return a * b } set { c = newValue * a * b } } } var d = SomeStruct(a: 2, b: 2) // d = 4 matrix[2, 1] = 3 // c = 6</pre>
Пример сабскрипта типа		<pre>enum Planet: Int { case mercury = 1, venus, earth, mars, jupiter, saturn static subscript(n: Int) -> Planet { return Planet(rawValue: n)! } } let mars = Planet[4] print(mars)</pre>

13. Наследование

Класс может наследовать методы, свойства и другие характеристики другого класса. Когда один класс наследует у другого класса, то наследующий класс называется **подклассом**, класс у которого наследуют - **суперклассом**. Наследование - фундаментальное поведение, которое отделяет классы от других типов Swift.

Любой класс, который ничего не наследует из другого класса, называется **базовым классом**.

Наследование является актом создания нового класса на базе существующего класса (базового класса). Подкласс наследует характеристики от существующего класса, который затем может быть усовершенствован. Вы так же можете добавить новые характеристики подклассу.

Для **индикации** того, что подкласс имеет суперкласс, просто напишите имя подкласса, затем имя суперкласса и разделите их двоеточием.

Переопределение это возможность подкласса иметь свои собственные реализации методов экземпляра, методов класса, свойств экземпляра, свойств класса или индекса, которые наследуются от суперкласса. Для переопределения характеристик, которые все равно будут унаследованы, вы приписываете к переписываемому определению ключевое слово **override**.

Доступ к методам, свойствам, индексам суперкласса

Можно в подклассе получить доступ к методу, свойству, индексу версии суперкласса, используя префикс **super**, примеры:

- **Переопределенный метод** `someMethod` может вызвать версию суперкласса метода `someMethod`, написав **`super.someMethod()`** внутри переопределения реализации метода.
- **Переопределённое свойство** `someProperty` может получить доступ к свойству версии суперкласса `someProperty` как **`super.someProperty`** внутри переопределения реализации геттера или сеттера.
- **Переопределенный индекс** для `someIndex` может получить доступ к версии суперкласса того же индекса как **`super[someIndex]`** изнутри переопределения реализации индекса.

Переопределения геттеров и сеттеров свойства

Вы можете предусмотреть **пользовательский геттер** (и **сеттер**, если есть в этом необходимость) для переопределения любого унаследованного свойства, несмотря на то, как свойство было определено в самом источнике, как свойство хранения или как вычисляемое. Подкласс **не знает** каким является унаследованное свойство хранимым или вычисляемым, все что он знает, так это имя свойства и его тип. Вы всегда должны констатировать и имя, и тип свойства, которое вы переопределяете, для того чтобы компилятор мог проверить соответствие и наличие переопределяемого свойства у суперкласса.

Вы можете представить унаследованное свойство **только для чтения**, как свойство, которое можно читать и редактировать, прописывая и геттер и сеттер в вашем переопределяемом свойстве подкласса. Однако вы **не можете** сделать наоборот, то есть сделать свойство редактируемое и читаемое только свойством для чтения.

Переопределение наблюдателей свойства

Можно использовать переопределение свойства для добавления **наблюдателей** к унаследованному свойству. Это позволяет вам получать уведомления об изменении значения унаследованного свойства, несмотря на то, как изначально это свойство было реализовано.

Нельзя добавить **наблюдателей свойства** на унаследованное константное свойство или на унаследованные вычисляемые свойства только для чтения. Значение этих свойств не может меняться, так что нет никакого смысла вписывать `willSet`, `didSet` как часть их реализации.

Также заметим, что вы **не можете обеспечить одно и то же свойство** и переопределяемым наблюдателем, и переопределяемым сеттером. Если вы хотите наблюдать за изменениями значения свойства, и вы готовы предоставить пользовательский сеттер для этого свойства, то вы можете просто наблюдать за изменением какого-либо значения из сеттера.

Предотвращение переопределений

Модно предотвратить переопределение **метода, свойства или индекса**, обозначив его как **конечный**. Сделать это можно написав ключевое слово **`final`** перед ключевым словом метода, свойства или индекса (`final var`, `final func`, `final class func`, и `final subscript`). Любая попытка переписать конечный метод, свойство или индекс в подклассе приведет к ошибке компиляции.

Можно отметить **целый класс** как конечный или финальный, написав слово **`final`** перед ключевым словом `class` (**`final class`**). Любая попытка унаследовать класс также приведет к ошибке компиляции.

Пример базового класса	<pre> class Vehicle { var currentSpeed = 0.0 var description: String { return "движется на скорости \\$(currentSpeed) миль в час" } func makeNoise() { //ничего не делаем, так как не каждый транспорт шумит } } let someVehicle = Vehicle() print("Транспорт: \\$(someVehicle.description)") //Транспорт: движется на скорости 0.0 миль в час </pre>
Синтаксис наследования	<pre> class SomeSubclass: SomeSuperclass { // определение подкласса проводится тут } </pre>
Пример наследования подклассом базового класса	<pre> class Bicycle: Vehicle { var hasBasket = false //добавили новое свойство } let bicycle = Bicycle() bicycle.hasBasket = true bicycle.currentSpeed = 15.0 print("Велосипед: \\$(bicycle.description)") //Велосипед: движется на скорости 15.0 миль в час class Tandem: Bicycle { var currentNumberOfPassengers = 0 } let tandem = Tandem() tandem.hasBasket = true tandem.currentNumberOfPassengers = 2 tandem.currentSpeed = 22.0 print("Тандем: \\$(tandem.description)") // Тандем: движется на скорости 22.0 миль в час </pre>
Переопределение методов	<pre> class Train: Vehicle { override func makeNoise() { print("Чу-чу") } } let train = Train() train.makeNoise() // Выведет "Чу-чу" </pre>
Пример доступа к свойству суперкласса	<pre> class Car: Vehicle { var gear = 1 override var description: String { return super.description + " на передаче \\$(gear)" } } let car = Car() car.currentSpeed = 25.0 car.gear = 3 print("Машина: \\$(car.description)") // Выведет "Машина: движется на скорости 25.0 миль в час на передаче 3" </pre>
Пример переопределения наблюдателей свойства	<pre> class AutomaticCar: Car { override var currentSpeed: Double { didSet { gear = Int(currentSpeed / 10.0) + 1 } } } let automatic = AutomaticCar() automatic.currentSpeed = 35.0 print("Машина с автоматом: \\$(automatic.description)") //Выведет "Машина с автоматом: движется на скорости 35.0 миль в час на пе редаче 4" </pre>

14. Инициализация

Инициализация - подготовительный процесс экземпляра класса, структуры или перечисления для дальнейшего использования. Этот процесс включает в себя установку начальных значений для каждого свойства хранения этого экземпляра и проведение любых настроек или инициализации, которые нужны до того, как экземпляр будет использоваться.

Инициализаторы в Swift не возвращают значения. Основная роль инициализаторов - убедиться в том, что новый экземпляр типа правильно инициализирован до того, как будет использован в первый раз.

Деинициализаторы - проводят любую чистку прямо перед тем, как экземпляр класса будет освобожден.

Установка начальных значений для свойств хранения

Классы и структуры **должны** устанавливать начальные значения у всех свойств хранения во время создания класса или структуры. Свойства хранения **не могут** быть оставлены в неопределённом состоянии.

Когда вы присваиваете значение по умолчанию свойству хранения или устанавливаете исходное значение в инициализаторе, то **значение устанавливается напрямую, без вызова наблюдателей.**

Инициализаторы вызываются для создания нового экземпляра конкретного типа. В самой простой своей форме инициализатор работает как метод экземпляра без параметров, написанный с помощью ключевого слова **init**.

Можно установить значение свойства по умолчанию, **как часть определения свойства**. Если свойство каждый раз берет одно и то же исходное значение, то лучше указать это значение, в качестве значения по умолчанию, чем каждый раз устанавливать его в инициализаторе.

Синтаксис инициализатора	<pre>init() { // инициализация проводится тут }</pre>
Пример инициализатора без параметров	<pre>struct Fahrenheit { var temperature: Double init() { temperature = 32.0 } } var f = Fahrenheit() print("Температура \((f.temperature)° по Фаренгейту") //Выведет "Температура 32.0° по Фаренгейту"</pre>
Установка значение свойства по умолчанию, как часть определения свойства	<pre>struct Fahrenheit { var temperature = 32.0 }</pre>

Настройка инициализации

Можно показать **параметры инициализации** как часть определения инициализатора, для определения типов и имен значений, которые настраивают процесс инициализации. Параметры инициализации имеют те же возможности и синтаксис как и параметры функции или метода.

Параметры инициализации могут иметь **локальные имена** для использования внутри тела инициализатора и **внешние имена** для использования при вызове инициализатора.

Инициализаторы не имеют своего имени до круглых скобок, как это имеют методы или функции. Поэтому имена и типы параметров инициализатора играют важную роль в определении того, какой инициализатор и где может быть использован.

Swift предоставляет **автоматические внешние имена** для каждого параметра, если вы, конечно, не укажете свое внешнее имя. **Невозможно вызвать инициализатор** без использования внешних имен.

Можно написать **подчеркивание** () вместо явного указания внешнего имени для параметра инициализатора, что бы опустить внешнее имя и чтобы переопределить поведение по умолчанию по автоматическому присваиванию внешнего имени.

Свойства **опционального типа** автоматически инициализируются со значением nil, указывая на то, что значение стремится иметь значение "пока что отсутствие значение" на этапе инициализации.

В экземплярах класса **постоянное свойство** может быть изменено только во время инициализации класса, в котором оно представлено. Оно не может быть изменено подклассом.

Пример структуры с инициализатором с параметрами инициализации	<pre> struct Celsius { var temperatureInCelsius: Double init(fromFahrenheit fahrenheit: Double) { temperatureInCelsius = (fahrenheit - 32.0) / 1.8 } init(fromKelvin kelvin: Double) { temperatureInCelsius = kelvin - 273.15 } } let boilingPointOfWater = Celsius(fromFahrenheit: 212.0) // boilingPointOfWater.temperatureInCelsius is 100.0 let freezingPointOfWater = Celsius(fromKelvin: 273.15) // freezingPointOfWater.temperatureInCelsius is 0.0 </pre>
Пример структуры с несколькими инициализаторами	<pre> struct Color { let red, green, blue: Double init(red: Double, green: Double, blue: Double) { self.red = red self.green = green self.blue = blue } init(white: Double) { red = white green = white blue = white } } let magenta = Color(red: 1.0, green: 0.0, blue: 1.0) let halfGray = Color(white: 0.5) </pre>
Пример структуры с _ во внешнем именем параметра инициализатора	<pre> struct Celsius { var temperatureInCelsius: Double init(_ celsius: Double) { temperatureInCelsius = celsius } } let bodyTemperature = Celsius(37.0) // bodyTemperature.temperatureInCelsius is 37.0 </pre>
<p>Опциональные типы свойств</p> <p>response автоматически присваивается значение nil при инициализации SurveyQuestion, значащее, что “значения пока нет”.</p> <p>Присваивание значений постоянным свойствам во время инициализации</p> <p>В примере: свойство text является постоянным, но оно может быть установлено в инициализаторе</p>	<pre> class SurveyQuestion { let text: String var response: String? init(text: String) { self.text = text } func ask() { print(text) } } let cheeseQuestion = SurveyQuestion(text: "Нравится ли вам сыр?") cheeseQuestion.ask() // Выведет "Нравится ли вам сыр?" cheeseQuestion.response = "Да, я люблю сыр" </pre>

Дефолтные инициализаторы

Swift предоставляет **дефолтный инициализатор** для любой структуры или базового класса, который имеет значение по умолчанию для всех его свойств и не имеет ни одного инициализатора. Дефолтный инициализатор просто создает новый экземпляр со всеми его свойствами с уже присвоенными значениями по умолчанию.

Структурные типы автоматически получают **почленный инициализатор**, если они не определяют своего пользовательского инициализатора. Это верно даже при условии, что хранимые свойства не имеют значений по умолчанию.

Почленный инициализатор - сокращенный способ инициализировать свойства члена нового экземпляра структуры. Начальные значения для свойств нового экземпляра могут быть переданы в почленный инициализатор по имени.

Пример инициализатора дефолтного (Свойство name - свойство опционального типа String, значит значение по умолчанию равно nil)	<pre>class ShoppingListItem { var name: String? var quantity = 1 var purchased = false } var item = ShoppingListItem()</pre>
Пример инициализатора почленного	<pre>struct Size { var width = 0.0, height = 0.0 } let twoByTwo = Size(width: 2.0, height: 2.0)</pre>

Делегирование инициализатора для типов значения

Делегирование инициализатора это процесс когда инициализатор вызывает другой(ие) инициализатор(ы) для инициализации части экземпляра. Он позволяет избегать дублирования кода в разных инициализаторах.

Для типов значений вы используете self.init для ссылки на остальные инициализаторы одного и того же типа значения, когда вы пишете свои инициализаторы. Вы можете вызывать self.init из инициализатора.

Если вы определите пользовательский инициализатор для типов значений, то вы больше не будете иметь доступа к дефолтному инициализатору (или почленному инициализатору, если это структура) для этого типа.

Если вы хотите, **чтобы ваш пользовательский тип значения имел возможность** быть инициализированным дефолтным инициализатором или почленным инициализатором, или вашим пользовательским инициализатором, то вам нужно написать свой пользовательский инициализатор в расширении вашего типа, чем как часть реализации типа значения.

Инициализатор Rect - init(center:size:) начинается с вычисления соответствующей исходной точки, основываясь на точке center и значении size. Только потом он вызывает (или делегирует) init(origin:size:) инициализатор, который хранит новую исходную точку и значения размеров соответствующих свойств	<pre>struct Size { var width = 0.0, height = 0.0 } struct Point { var x = 0.0, y = 0.0 } struct Rect { var origin = Point() var size = Size() init() {} init(origin: Point, size: Size) { self.origin = origin self.size = size } init(center: Point, size: Size) { let originX = center.x - (size.width / 2) let originY = center.y - (size.height / 2) self.init(origin: Point(x: originX, y: originY), size: size) } } let basicRect = Rect() //исходная точка Rect (0.0, 0.0) и его размер (0.0, 0.0) let originRect = Rect(origin: Point(x: 2.0, y: 2.0), size: Size(width: 5.0, height: 5.0)) //исходная точка Rect (2.0, 2.0) и его размер (5.0, 5.0) let centerRect = Rect(center: Point(x: 4.0, y: 4.0), size: Size(width: 3.0, height: 3.0)) //исходная точка centerRect'а равна (2.5, 2.5) и его размер (3.0, 3.0)</pre>
---	---

Наследование и инициализация класса

Всем свойствам класса, включая и те, что унаследованы у суперкласса должны быть присвоены начальные значения, во время их инициализации

Swift определяет два вида инициализаторов классовых типов для проверки того, что все свойства получили какие-либо значения: **назначенные инициализаторы (конструкторы)** или **вспомогательные инициализаторы**.

Назначенный и вспомогательный инициализатор

Назначенные инициализаторы в основном инициализаторы класса. Они предназначены для того, чтобы полностью инициализировать все свойства представленные классом и чтобы вызвать соответствующий инициализатор суперкласса для продолжения процесса инициализации цепочки наследований суперклассов.

Назначенные инициализаторы объединяют в себе все точки, через которые проходит процесс инициализации и через которые процесс инициализации идет по цепочке в суперкласс.

Каждый класс должен иметь хотя бы один назначенный инициализатор. В некоторых случаях, это требование удовлетворяется наследованием одного или более назначенных инициализаторов от суперкласса.

Вспомогательные инициализаторы являются вторичными, поддерживающими инициализаторами для класса. Вы можете определить вспомогательный инициализатор для вызова назначенного инициализатора из того же класса, что и вспомогательный инициализатор с некоторыми параметрами назначенного инициализатора с установленными начальными значениями. Вы не обязаны обеспечивать вспомогательные инициализаторы, если ваш класс не нуждается в них.

Синтаксис назначенных и вспомогательных инициализаторов

Назначенные инициализаторы	<code>init(параметры) { выражения }</code>
Вспомогательные инициализаторы	<code>convenience init(параметры) { выражения }</code>

Делегирование инициализатора для классовых типов

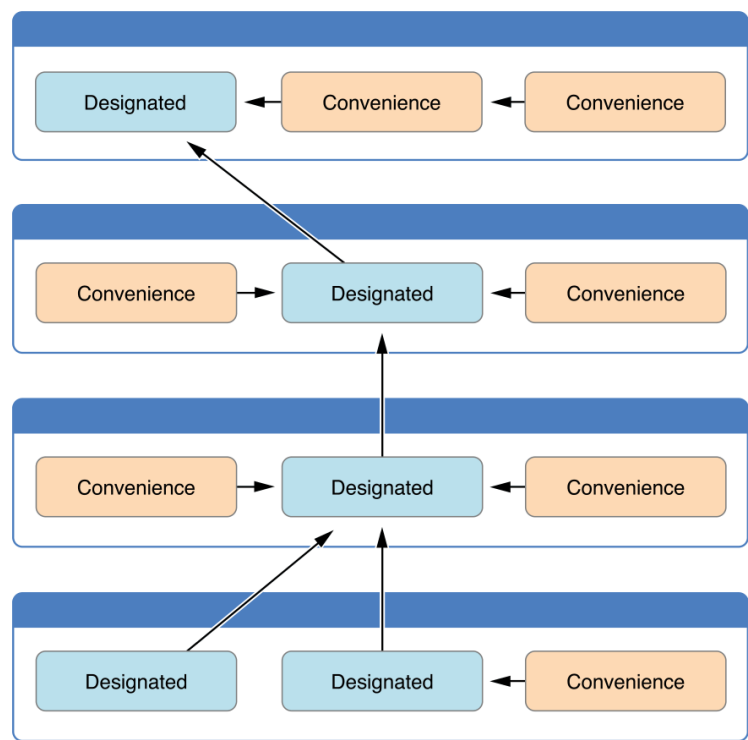
Для простоты отношений между назначенными и вспомогательными инициализаторами, **Swift использует следующие три правила** для делегирования вызовов между инициализаторами:

- **Правило 1:** Назначенный инициализатор должен вызывать назначенный инициализатор из суперкласса.
- **Правило 2:** Вспомогательный инициализатор должен вызывать другой инициализатор из того же класса.
- **Правило 3:** Вспомогательный инициализатор в конечном счете должен вызывать назначенный инициализатор.

Вот как можно просто это запомнить:

- Назначенные инициализаторы должны делегировать наверх
- Вспомогательные инициализаторы должны делегировать по своему уровню (классу).

Эти правила никак не относятся к тому, как пользователи ваших классов создают экземпляры каждого класса. Любой инициализатор из схемы выше может быть использован для создания полностью инициализированного экземпляра класса, которому он принадлежит. Правила влияют лишь на то, как вы будете писать реализацию класса.



Двухфазная инициализация

Инициализация класса в Swift является двухфазным процессом. На первой фазе каждое свойство хранения получает начальное значение от класса, в котором оно представлено. Как только первоначальные значения для свойств хранения были определены, начинается вторая фаза, и каждому классу предоставляется возможность изменить свои свойства еще до того как будет считаться, что созданный экземпляр можно использовать.

Использование двухфазного процесса инициализации делает инициализацию безопасной, в то же время обеспечивая полную гибкость классов в классовой иерархии. Двухфазная инициализация предотвращает доступ к значениям свойств до того, как они будут инициализированы и не допускает случайную установку значения свойства другим инициализатором.

Двухфазный процесс инициализации в Swift **аналогичен** инициализации в Objective-C. **Основное отличие** между ними проходит на первой фазе в том, что в Objective-C свойства получают значения 0 или nil. В Swift же этот процесс более гибкий и позволяет устанавливать пользовательские начальные значения и может обработать типы, для которых значения 0 или nil, являются некорректными.

Компилятор Swift проводит четыре полезные проверки безопасности для подтверждения того, что ваша двухфазная инициализация прошла без ошибок:

- **Проверка 1.** Назначенный инициализатор должен убедиться в том, что все свойства представленные его классом инициализированы до того, как он делегирует наверх, в инициализатор суперкласса. Память для объекта считается полностью инициализированной только для полностью инициализированного объекта, где все значения хранимых свойств известны. Для того чтобы удовлетворить этому правилу, назначенный инициализатор должен убедиться, что все его собственные свойства инициализированы до того, как будут переданы вверх по цепочке.
- **Проверка 2.** Назначенный инициализатор должен делегировать суперклассу инициализатор до присваивания значений унаследованным свойствам. Если этого сделано не будет, то новое значение, которое присвоит назначенный инициализатор будет переписано суперклассом, как часть инициализации суперкласса.
- **Проверка 3.** Вспомогательный инициализатор должен делегировать другому инициализатору до того, как будут присвоены значения любым свойствам (включая свойства определенные тем же классом). Если этого сделано не будет, то новое значение, которое присваивает вспомогательный инициализатор, будет перезаписано его собственным назначенным инициализатором класса.
- **Проверка 4.** Инициализатор не может вызывать методы экземпляра, читать значения любого свойства экземпляра или ссылаться на self как на значение до тех пор, пока не будет закончена первая фаза инициализации.

Экземпляр класса является не совсем корректным до тех пор, пока не закончится первая фаза. К свойствам можно получить доступ и можно вызывать методы только тогда, как стало известно, что экземпляр валиден (корректен) к концу первой фазы.

Вот как проходит двухфазная инициализация, основанная на четырех проверках:

Фаза первая:

- Назначенный или вспомогательный инициализатор вызывается в классе.
- Память под новый экземпляр этого класса выделяется. Но она еще не инициализирована.
- Назначенный инициализатор для этого класса подтверждает, что все свойства, представленные этим классом, имеют значения. Память под эти свойства теперь инициализирована.
- Назначенный инициализатор передает инициализатору суперкласса, что пора проводить те же действия, только для его собственных свойств.
- Так продолжается по цепочке до самого верхнего суперкласса.
- После того как верхушка этой цепочки достигнута и последний класс в цепочке убедился в том, что все его свойства имеют значение, только тогда считается, что память для этого экземпляра полностью инициализирована. На этом первая фаза кончается.

Фаза вторая:

- Двигаясь вниз по цепочке, каждый назначенный инициализатор в этой цепочке имеет такую возможность, как настраивать экземпляр. Теперь инициализаторы получают доступ к self и могут изменять свои свойства, создавать экземпляры и вызывать методы и т.д.
- И наконец, каждый вспомогательный инициализатор в цепочке имеет возможность настраивать экземпляр и работать с self.

Наследование и переопределение инициализатора

В отличие от подклассов в Objective-C, **подклассы в Swift не наследуют инициализаторов их суперклассов по умолчанию**. Такой подход в Swift предотвращает ситуации, когда простой инициализатор суперкласса наследуется более специфичным подклассом, а потом используется для создания экземпляра подкласса, который не полностью или не правильно инициализирован.

Инициализаторы суперкласса наследуются в определенных обстоятельствах, но только когда это безопасно и когда это имеет смысл делать.

Если вы хотите, **чтобы у вашего подкласса были один или более инициализаторов его суперклассов**, вы можете сделать свою реализацию этих инициализаторов внутри подкласса. Когда вы пишете инициализатор подкласса, который совпадает с назначенным инициализатором суперкласса, вы фактически переопределяете назначенный инициализатор. Таким образом **вы должны писать модификатор override** перед определением инициализатора подкласса. Это верно даже если вы переопределяете автоматически предоставляемый инициализатор.

Вы всегда пишете модификатор override, когда переписываете назначенный инициализатор суперкласса, даже если ваша реализация инициализатора подкласса является вспомогательным инициализатором.

Если вы пишете инициализатор подкласса, который совпадает с вспомогательным инициализатором суперкласса, то этот вспомогательный инициализатор суперкласса никогда не сможет быть вызван напрямую вашим подклассом, в соответствии с правилами указанными выше. Таким образом ваш подкласс не проводит переопределение инициализатора суперкласса. И в результате, **вы не пишете модификатор override**, когда проводите совпадающую реализацию вспомогательного инициализатора суперкласса.

Подклассы могут менять унаследованные переменные свойства в процессе инициализации, но **нельзя менять** неизменяемые унаследованные свойства.

Родительский класс	<pre>class Vehicle { var numberOfWheels = 0 var description: String { return "\(numberOfWheels) колес(o)" } } let vehicle = Vehicle() print("Транспортное средство \(vehicle.description)") //Транспортное средство 0 колес(o)</pre>
Переопределение инициализатора в подклассе и вызов родительского инициализатора в переопределенном подклассе	<pre>class Bicycle: Vehicle { override init() { super.init() numberOfWheels = 2 } } let bicycle = Bicycle() print("Велосипед: \(bicycle.description)") //Велосипед: 2 колес(a)</pre>

Автоматическое наследование инициализатора

Если вы предоставляете значения по умолчанию любому новому свойству, представленному в подклассе, то применяются два правила:

- **Правило 1.** Если ваш подкласс не определяет ни одного назначенного инициализатора, он автоматически наследует все назначенные инициализаторы суперкласса.
- **Правило 2.** Если у вашего класса *есть реализация всех назначенных инициализаторов его суперкласса*, либо они были унаследованы как по правилу 1 или же предоставлены как часть пользовательской реализации определения подкласса, то тогда этот подкласс автоматически наследует все вспомогательные инициализаторы суперкласса.

Эти правила применимы даже, если ваш подкласс позже добавляет вспомогательные инициализаторы.

Подкласс может реализовать назначенный инициализатор суперкласса как вспомогательный инициализатор подкласса в качестве части удовлетворяющей правилу 2.

Назначенные и вспомогательные инициализаторы в действии

 <pre>class Food var name: String Convenience init() Designated init(name)</pre>	<pre>class Food { var name: String init(name: String) { self.name = name } convenience init() { self.init(name: "[Unnamed]") } } let namedMeat = Food(name: "Бекон") //имя namedMeat является "Бекон" let mysteryMeat = Food() //mysteryMeat называется "[Unnamed]"</pre>
 <pre>class Food var name: String Convenience init() Designated init(name) class RecipeIngredient: Food var quantity: Int Inherited init() Convenience init(name) Designated init(name, quantity)</pre>	<pre>class RecipeIngredient: Food { var quantity: Int init(name: String, quantity: Int) { self.quantity = quantity super.init(name: name) } override convenience init(name: String) { self.init(name: name, quantity: 1) } } let oneMysteryItem = RecipeIngredient() let oneBacon = RecipeIngredient(name: "Bacon") let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)</pre>
 <pre>class Food var name: String Convenience init() Designated init(name) class RecipeIngredient: Food var quantity: Int Inherited init() Convenience init(name) Designated init(name, quantity) class ShoppingListItem: RecipeIngredient var purchased = false Inherited init() Inherited init(name) Inherited init(name, quantity)</pre>	<pre>class ShoppingListItem: RecipeIngredient { var purchased = false var description: String { var output = "\(quantity) x \(name)" output += purchased ? " ✓" : " X" return output } } var breakfastList = [ShoppingListItem(), ShoppingListItem(name: "Bacon"), ShoppingListItem(name: "Eggs", quantity: 6)] breakfastList[0].name = "Orange juice" breakfastList[0].purchased = true for item in breakfastList { print(item.description) } // 1 x Orange juice ✓ // 1 x Bacon X // 6 x Eggs X</pre>

Проваливающиеся инициализаторы

Для того чтобы справиться с условиями инициализации, которые могут провалиться, определите один или несколько проваливающихся инициализаторов как часть определения класса, структуры или перечисления. Вы можете написать проваливающийся инициализатор поместив вопросительный знак после ключевого слова `init` (*init?*).

Проваливающийся инициализатор создает опциональное значение типа, который он инициализирует. Вы пишете *return nil* внутри проваливающегося инициализатора для индикации точки, где инициализация может провалиться.

Вы не можете определить проваливающийся инициализатор и обычные инициализаторы с одними и теми же именами и типами параметров.

Строго говоря, **инициализаторы не возвращают значений**. Их роль заключается в том, что они проверяют, что `self` полностью и корректно инициализирован, до того, как инициализация закончится. Несмотря на то, что вы пишете `return nil` для указания неудачи инициализации, **вы не пишете слово `return` в случае**, если инициализация прошла успешно.

<p>Например, проваливающиеся инициализатора</p> <p>Пустая строка "" это не то же самое что nil.</p>	<pre> struct Animal { let species: String init?(species: String) { if species.isEmpty { return nil } self.species = species } } let someCreature = Animal(species: "Жираф") // someCreature имеет тип Animal?, но не Animal if let giraffe = someCreature { print("Мы инициализировали животное типа \(giraffe.species) ") } // Выведет "Мы инициализировали животное типа Жираф " let anonymousCreature = Animal(species: "") // anonymousCreature имеет тип Animal?, но не Animal if anonymousCreature == nil { print("Неизвестное животное не может быть инициализировано") } // Выведет "Неизвестное животное не может быть инициализировано" </pre>
---	--

Проваливающиеся инициализаторы для перечислений

Вы можете использовать проваливающийся инициализатор для выбора подходящего члена перечисления основываясь на одном или более параметров. Инициализатор может провалиться, если предоставленные параметры не будут соответствовать подходящему члену перечисления.

<p>Проваливающийся инициализатор используется для того, чтобы найти подходящий член перечисления для значения типа Character, которое представляет символ температуры</p>	<pre> enum TemperatureUnit { case kelvin, celsius, fahrenheit init?(symbol: Character) { switch symbol { case "K": self = .kelvin case "C": self = .celsius case "F": self = .fahrenheit default: return nil } } } let fahrenheitUnit = TemperatureUnit(symbol: "F") if fahrenheitUnit != nil { print("Эта единица измерения температуры определена, а значит наша инициализация прошла успешно!") } // Выведет "Эта единица измерения температуры определена, а значит наша инициализация прошла успешно!" let unknownUnit = TemperatureUnit(symbol: "X") if unknownUnit == nil { print("Единица измерения температуры не определена, таким образом мы зафейлили инициализацию") } // Выведет "Единица измерения температуры не определен а, таким образом мы зафейлили инициализацию" </pre>
--	--

Проваливающиеся инициализаторы для перечислений с начальными значениями

Перечисления с начальными значениями по умолчанию получают проваливающийся инициализатор `init?(rawValue:)`, который принимает параметр `rawValue` подходящего типа и выбирает соответствующий член перечисления, если он находит подходящий, или срабатывает сбой инициализации, если существующее значение не находит совпадения среди членов перечисления.

Пример TemperatureUnit из примера выше для использования начальных значений типа Character и использовать инициализатор init?(rawValue:)	<pre>enum TemperatureUnit: Character { case kelvin = "K", celsius = "C", fahrenheit = "F" } let fahrenheitUnit = TemperatureUnit(rawValue: "F") if fahrenheitUnit != nil { print("Эта единица измерения температура определена, а значит наша инициализация прошла успешно!") } // Выведет "Эта единица измерения температура определена, а значит наша инициализация прошла успешно!" let unknownUnit = TemperatureUnit(rawValue: "X") if unknownUnit == nil { print("Единица измерения температуры не определена, таким образом мы зафейлили инициализацию.") } // Выведет "Единица измерения температуры не определена, таким образом мы зафейлили инициализацию."</pre>
---	---

Распространение проваливающегося инициализатора

Проваливающийся инициализатор класса, структуры, перечисления **может быть делегирован** к другому проваливающемуся инициализатору из того же класса, структуры, перечисления. Аналогично проваливающийся инициализатор подкласса может быть делегирован вверх в проваливающийся инициализатор суперкласса.

В любом случае, если вы делегируете другому инициализатору, который проваливает инициализацию то и **весь процесс инициализации проваливается немедленно за ним**, и далее никакой код инициализации уже не исполняется.

Проваливающийся инициализатор может также делегировать к непроваливающемуся инициализатору. Используя такой подход, вам следует добавить потенциальное состояние провала в существующий процесс инициализации, который в противном случае не провалится.

Два класса проваливающегося инициализаторами, один наследует другой	<pre>class Product { let name: String init?(name: String) { if name.isEmpty { return nil } self.name = name } } class CartItem: Product { let quantity: Int init?(name: String, quantity: Int) { if quantity < 1 { return nil } self.quantity = quantity super.init(name: name) } }</pre>
Если вы создаете экземпляр CartItem с name не равной пустой строке и quantity равному 1 или более, то инициализация проходит успешно	<pre>if let twoSocks = CartItem(name: "sock", quantity: 2) { print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)") } // Выведет "Item: sock, quantity: 2"</pre>
Если вы попытаетесь создать экземпляр CartItem с quantity со значением 0, то инициализация провалится	<pre>if let zeroShirts = CartItem(name: "shirt", quantity: 0) { print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)") } else { print("Невозможно инициализировать ноль футболок") } // Выведет "Невозможно инициализировать ноль футболок"</pre>
Если вы попытаетесь создать экземпляр CartItem с name равным пустой строке, то инициализатор суперкласса Product вызовет неудачу инициализации	<pre>if let oneUnnamed = CartItem(name: "", quantity: 1) { print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)") } else { print("Невозможно инициализировать товар без имени") } // Выведет "Невозможно инициализировать товар без имени"</pre>

Переопределение проваливающегося инициализатора

Вы можете переопределить проваливающийся инициализатор суперкласса в подклассе, так же как любой другой инициализатор. Или **вы можете переопределить проваливающий инициализатор суперкласса непроваливающимся инициализатором подкласса**. Это позволяет вам определить подкласс, для которого инициализация не может провалиться, даже когда инициализация суперкласса позволяет это сделать.

Если вы переопределяете проваливающийся инициализатор суперкласса не проваливающимся инициализатором подкласса, то **единственным способом делегировать в инициализатор суперкласса** - принудительное извлечение результата из проваливающего инициализатора суперкласса.

Вы можете переопределить проваливающийся инициализатор непроваливающимся инициализатором, **но не наоборот**.

Переопределение проваливающегося инициализатора	<pre>class Document { var name: String? //Этот инициализатор создает документ со значением nil свойства name init(){} //Этот инициализатор создает документ с не пустым свойством name init?(name: String) { if name.isEmpty { return nil } self.name = name } } class AutomaticallyNamedDocument: Document { override init() { super.init() self.name = "[Untitled]" } override init(name: String) { super.init() if name.isEmpty { self.name = "[Untitled]" } else { self.name = name } } } class UntitledDocument: Document { override init() { super.init(name: "[Untitled]")! } }</pre>
---	--

Проваливающийся инициализатор init!

Обычно вы определяете проваливающийся инициализатор, который создает опциональный экземпляр соответствующего типа путем размещения знака вопроса после ключевого слова init (init?). Альтернативно, вы можете определить **проваливающийся инициализатор, который создает экземпляр неявно извлекаемого опционала соответствующего типа**. Сделать это можно, если вместо вопросительного знака поставить восклицательный знак после ключевого слова init (**init!**).

Вы можете делегировать от init? в init! и наоборот, а так же вы можете переопределить init? с помощью init! и наоборот. Вы так же можете делегировать от init в init!, хотя, делая таким образом, мы заставим сработать утверждение, если init! провалит инициализацию.

Требуемые инициализаторы

Напишите **required** перед определением инициализатора класса, если вы хотите, чтобы каждый подкласс этого класса был обязан реализовывать этот инициализатор

Вы также должны писать модификатор required перед каждой реализацией требуемого инициализатора класса для индикации того, что последующий подкласс так же должен унаследовать этот инициализатор по цепочке. **Вы не пишете override**, когда переопределяете требуемый инициализатор.

Вы не должны обеспечивать явную реализацию требуемого инициализатора, если вы можете удовлетворить требование унаследованным инициализатором.

Требуемые инициализаторы	<pre> class SomeClass { required init() { //пишем тут реализацию инициализатора } } class SomeSubclass: SomeClass { required init() { //пишем тут реализацию инициализатора подкласса } } </pre>
--------------------------	--

Начальное значение свойства в виде функции или замыкания

Если начальное значение свойства требует какой-то настройки или структуризации, то вы можете использовать замыкание или глобальную функцию, которая будет предоставлять значение для этого свойства. Как только создается новый экземпляр, вызывается функция или замыкание, которая возвращает значение, которое присваивается в качестве начального значения свойства.

После закрывающей фигурной скобки замыкания идут пустая пара круглых скобок. Это означает, что нужно исполнить это замыкание немедленно. Если вы пропустите эти скобки, то вы присваиваете само значение замыкания свойству, а не возвращаете значения замыкания.

Если вы используете замыкание для инициализации свойства, **помните, что остальная часть экземпляра еще не инициализирована**, на тот момент когда исполняется замыкание. Это значит, что вы не можете получить доступ к значениям других свойств из вашего замыкания, даже если эти свойства имеют начальное значение. Вы так же не можете использовать неявное свойство self и не можете вызвать какой-либо метод вашего экземпляра.

Начальное значение свойства в виде функции или замыкания	<pre> class SomeClass { let someProperty: SomeType = { // создаем начальное значения для SomeProperty внутри э // someValue должен быть того же типа, что и SomeType return someValue }() } </pre>
Начальное значение свойства в виде функции или замыкания	<pre> struct Chessboard { let boardColors: [Bool] = { var temporaryBoard = [Bool]() var isBlack = false for i in 1...8 { for j in 1...8 { temporaryBoard.append(isBlack) isBlack = !isBlack } isBlack = !isBlack } return temporaryBoard }() func squareIsBlackAt(row: Int, column: Int) -> Bool { return boardColors[(row * 8) + column] } } let board = Chessboard() print(board.squareIsBlackAt(row: 0, column: 1)) // Выведет "true" print(board.squareIsBlackAt(row: 7, column: 7)) // Выведет "false" </pre>

15. Деинициализация

Деинициализаторы вызываются автоматически прямо перед тем как освобождается экземпляр.

Вы пишете деинициализаторы с ключевого слова ***deinit***.

Деинициализаторы ***доступны*** только для классовых типов.

В декларировании класса можно прописать максимум ***один деинициализатор*** на один класс.

Деинициализатор ***не принимает*** ни одного параметра и пишется без круглых скобок.

У вас ***нет возможности*** вызывать деинициализатор самостоятельно. Деинициализаторы суперкласса ***наследуются*** их подклассами, и деинициализаторы суперкласса вызываются автоматически в конце реализации деинициализатора подкласса. Деинициализаторы суперклассов ***всегда вызываются***, даже если подкласс не имеет своего деинициализатора.

Так как экземпляр не освобождается до тех пор пока не будет вызван деинициализатор, то деинициализатор ***может получить доступ*** ко всем свойствам экземпляра, который он вызывает, и может изменить свое поведение, основываясь на этих свойствах

Синтаксис деинициализатора	<pre>deinit { // проведение деинициализации }</pre>
Пример деинициализатора	<pre>class Player { var playerName: String deinit { print("PlayerOne has left the game") } } var playerOne: Player? = Player(playerName: "John") playerOne = nil // Выведет "PlayerOne has left the game"</pre>

16. Опциональная последовательность

Опциональная цепочка (optional chaining) - процесс запросов и вызовов свойств, методов, сабскриптов (индексов) у опционала, который может быть nil. Если опционал содержит какое-либо значение, то вызов свойства, метода или сабскрипта успешен, и наоборот, если опционал равен nil, то вызов свойства, метода или сабскрипта возвращает nil.

Множественные запросы **могут быть соединены** вместе, и вся цепочка этих запросов не срабатывает, если хотя бы один запрос равен nil.

Вы обозначаете опциональную последовательность, когда ставите **вопросительный знак (?)** опционального значения, свойство, метод или индекс которого вы хотите вызвать, если опционал не nil.

Опциональная последовательность не исполняется, если опционал равен nil

Принудительное извлечение (восклицательный знак (!) после опционального значения) приводит к runtime ошибке, когда опционал равен nil.

Факт того, что опциональная последовательность может быть вызвана и на значение nil, отражается в том, что **результатом работы опциональной последовательности** всегда является опциональная величина, даже в том случае, если свойство, метод или сабскрипт, к которым вы обращаетесь, возвращает неопциональное значение. Вы можете использовать это значение опционального возврата для проверки успеха (если возвращенный опционал содержит значение) или неудачи (если возвращенное значение опционала nil).

Можно использовать **опциональную последовательность для вызовов** свойств, методов, сабскриптов, которые находятся более чем на один уровень глубже. Это позволяет вам пробираться через подсвойства, внутри сложных моделей вложенных типов, и проверять возможность доступа свойств, методов и сабскриптов этих подсвойств.

Можете использовать опциональную последовательность **для доступа к свойству** опционального значения и проверить результат доступа к этому свойству на успешность.

Можно использовать опциональную последовательность **для вызова метода** опциональной величины, и проверить сам вызов метода на успешность. Вы можете сделать это, даже если этот метод не возвращает значения.

Если вы вызовете этот метод на опциональном значении в опциональной последовательности, то он **вернет тип не Void, а Void?**, потому что возвращаемые значения всегда опционального типа, когда они вызываются через опциональную последовательность.

Любая попытка установить свойство через опциональную последовательность возвращает значение Void?, которое **позволяет вам сравнивать** его с nil, для того, чтобы увидеть логический результат установки значения свойству (успех, провал).

Вы можете использовать опциональную последовательность для того, чтобы попробовать получить и установить значения из индекса опционального значения, и проверить успешность выполнения **вызова сабскрипта**.

Когда вы получаете доступ к опциональному значению через опциональную последовательность, вы размещаете **вопросительный знак до скобок сабскрипта (индекса)**, а не после. Вопросительный знак опциональной последовательности следует сразу после части выражения, которая является опционалом.

Если сабскрипт возвращает значение опционального типа, например ключ словаря типа Dictionary в Swift, то мы должны поставить **вопросительный знак после закрывающей скобки сабскрипта**, для присоединения его опционального возвращаемого значения

Вы можете **соединить несколько уровней** опциональных последовательностей вместе для того, чтобы пробраться до свойств, методов, сабскриптов, которые находятся глубже в модели. Однако многоуровневые опциональные последовательности **не добавляют новых уровней** опциональности к возвращаемым значениям:

- Если тип, который вы пытаетесь получить **не опциональный**, то он станет опциональным из-за опциональной последовательности.
- Если тип, который вы пытаетесь получить, **уже опциональный**, то более опциональным он уже не станет, даже по причине опциональной последовательности.

Связывание методов в опциональной последовательности с опциональными возвращаемыми значениями: можно использовать опциональную последовательность для вызова метода, который возвращает значение опционального типа, а затем к этой опциональной последовательности может прикрепить и возвращаемое значение самого метода, если это нужно.

Пример ОП	<pre> class Person { var residence: Residence? } class Residence { var numberOfRooms = 1 } let john = Person() let roomCount = john.residence!.numberOfRooms // ошибка runtime, так как residence не имеет значения для извлечения if let roomCount = john.residence?.numberOfRooms { print("John's residence has \\$(roomCount) room(s).") } else { print("Unable to retrieve the number of rooms.") } // Выведет "Unable to retrieve the number of rooms." john.residence = Residence() if let roomCount = john.residence?.numberOfRooms { print("John's residence has \\$(roomCount) room(s).") } else { print("Unable to retrieve the number of rooms.") } // Выведет "John's residence has 1 room(s)."</pre>
Содержимое классов для примеров ниже	<pre> class Person { var residence: Residence? } class Residence { var rooms = [Room]() var numberOfRooms: Int { return rooms.count } subscript(i: Int) -> Room { get { return rooms[i] } set { rooms[i] = newValue } } func printNumberOfRooms() { print("Общее количество комнат равно \\$(numberOfRooms)") } var address: Address? } class Room { let name: String init(name: String) { self.name = name } } class Address { var buildingName: String? var buildingNumber: String? var street: String? func buildingIdentifier() -> String? { if let buildingNumber = buildingNumber, let street = street { return "\\$(buildingNumber) \\$(street)" } else if buildingName != nil { return buildingName } else { return nil } } } }</pre>
Доступ к свойствам через ОП	<pre> let john = Person() if let roomCount = john.residence?.numberOfRooms { print("John's residence has \\$(roomCount) room(s).") } else { print("Unable to retrieve the number of rooms.") } // Выведет "Unable to retrieve the number of rooms."</pre>

Установка свойств с помощью функции через ОП	<pre> func createAddress() -> Address { print("Function was called.") let someAddress = Address() someAddress.buildingNumber = "29" someAddress.street = "Acacia Road" return someAddress } john.residence?.address = createAddress() </pre>
Вызов методов через ОП	<pre> func printNumberOfRooms() { print("Общее количество комнат равно \(numberOfRooms)") } if john.residence?.printNumberOfRooms() != nil { print("Есть возможность вывести общее количество комнат.") } else { print("Нет возможности вывести общее количество комнат.") } // Выведет "Нет возможности вывести общее количество комнат." </pre>
Сравнение результата установки значения через ОП с nil	<pre> if (john.residence?.address = someAddress) != nil { print("Была возможность установить адрес.") } else { print("Не было возможности установить адрес.") } // Выведет "Не было возможности установить адрес." </pre>
Доступ к сабскриптам через ОП	<pre> if let firstRoomName = john.residence?[0].name { print("Название первой комнаты \(firstRoomName).") } else { print("Никак не получить название первой комнаты.") } // Выведет "Никак не получить название первой комнаты." </pre>
Установка значения через сабскрипт через ОП	<pre> john.residence?[0] = Room(name: "Bathroom") </pre>
Получение доступа к сабскрипту (индексу) опционального типа	<pre> var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]] testScores["Dave"]?[0] = 91 testScores["Bev"]?[0] += 1 testScores["Brian"]?[0] = 72 // массив "Dave" [91, 82, 84], массив "Bev" [80, 94, 81] </pre>
Соединение нескольких уровней ОП	<pre> if let johnsStreet = john.residence?.address?.street { print("John's street name is \(johnsStreet).") } else { print("Unable to retrieve the address.") } // Выведет "Unable to retrieve the address." </pre>
Связывание методов в ОП с опциональными возвращаемыми значениями	<pre> if let beginsWithThe = john.residence?.address?.buildingIdentifier()?.hasPrefix("The") { if beginsWithThe { print("John's building identifier begins with \"The\".") } else { print("John's building identifier does not begin with \"The\".") } } // Выведет "John's building identifier begins with \"The\"." </pre>

17. Обработка ошибок

Обработка ошибок - это процесс реагирования на возникновение ошибок и восстановление после появления ошибок в программе.

Обработка ошибок в Swift **перекликается с шаблонами обработки ошибок**, которые используются в классе NSError в Cocoa и Objective-C.

В Swift ошибки отображаются значениями типов, которые **соответствуют протоколу Error**. Этот пустой протокол является индикатором того, что это перечисление может быть использовано для обработки ошибок.

Перечисления в Swift особенно хорошо подходят для группировки схожих между собой условий возникновения ошибок и соответствующих им значений, что позволяет получить дополнительную информацию о природе самой ошибки.

Для того чтобы «сгенерировать» ошибку, вы используете инструкцию **throw**.

Когда генерируется ошибка, то фрагмент кода, окружающий ошибку, должен быть ответственным за ее обработку. В Swift существует **четыре способа** обработки ошибок.

- можно **передать (propagate) ошибку из функции в код**, который вызывает саму эту функцию;
- можно обработать ошибку, используя инструкцию **do-catch**;
- можно обработать ошибку, как значение **опционала**;
- можно поставить **утверждение**, что ошибка в данном случае исключена.

Когда функция генерирует ошибку, последовательность выполнения вашей программы меняется, поэтому **важно сразу обнаружить место в коде**, которое может генерировать ошибки. Для того, чтобы выяснить где именно это происходит, напишите ключевое слово **try** - или варианты try? или try! - до куска кода, вызывающего функцию, метод или инициализатор, который может генерировать ошибку.

Обработка ошибок в Swift **напоминает обработку исключений (exceptions)** в других языках, с использованием ключевых слов try, catch и throw. В отличие от обработки исключений во многих языках, в том числе и в Objective-C- обработка ошибок в Swift не включает разворачивание стека вызовов, то есть процесса, который может быть дорогим в вычислительном отношении. Таким образом, производительные характеристики инструкции throw сопоставимы с характеристиками оператора return.

Декларирование ошибок в перечислении	<pre>enum VendingMachineError: Error { case invalidSelection case insufficientFunds(coinsNeeded: Int) case outOfStock }</pre>
Генерация ошибки	<pre>throw VendingMachineError.insufficientFunds(coinsNeeded: 5)</pre>

Передача ошибки с помощью генерирующей функции

Чтобы указать, что функция, метод или инициализатор могут генерировать ошибку, вам нужно написать **ключевое слово throws** в реализации функции после ее параметров. Функция, отмеченная throws называется **генерирующей функцией**. Если у функции установлен возвращаемый тип, то вы пишете ключевое слово throws перед стрелкой возврата (->).

Только генерирующая ошибку функция может передавать ошибки. Любые ошибки, сгенерированные внутри **non-throwing функции**, должны быть обработаны внутри самой функции.

Генерирующие ошибку инициализаторы могут распространять ошибки таким же образом, как генерирующие ошибку функции.

В примере метод vend(itemNamed:) генерирует соответствующую VendingMachineError	<pre>struct Item { var price: Int var count: Int } class VendingMachine { var inventory = ["Candy Bar": Item(price: 12, count: 7), "Chips": Item(price: 10, count: 4), "Pretzels": Item(price: 7, count: 11)] var coinsDeposited = 0 func vend(itemNamed name: String) throws { guard let item = inventory[name] else {</pre>
--	---

	<pre> throw VendingMachineError.invalidSelection } guard item.count > 0 else { throw VendingMachineError.outOfStock } guard item.price <= coinsDeposited else { throw VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited) } coinsDeposited -= item.price var newItem = item newItem.count -= 1 inventory[name] = newItem print("Dispensing \(name)") } } </pre>
В примере метод <code>buyFavoriteSnack(person: vendingMachine:)</code> это генерирующая функция	<pre> let favoriteSnacks = ["Alice": "Chips", "Bob": "Licorice", "Eve": "Pretzels"] func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws { let snackName = favoriteSnacks[person] ?? "Candy Bar" try vendingMachine.vend(itemNamed: snackName) } </pre>
Генерирующий ошибку инициализатор	<pre> struct PurchasedSnack { let name: String init(name: String, vendingMachine: VendingMachine) throws { try vendingMachine.vend(itemNamed: name) self.name = name } } </pre>

Обработка ошибок с использованием do-catch

Используйте **инструкцию do-catch** для обработки ошибок, запуская блок кода. Если выдается ошибка в коде условия do, она соотносится с условием catch для определения того, кто именно сможет обработать ошибку.

Вы пишете **шаблон** после ключевого слова catch, чтобы указать какие ошибки могут обрабатываться данным пунктом этого обработчика. **Если условие catch не имеет своего шаблона**, то оно подходит под любые ошибки и связывает ошибки к локальной константе error.

Если генерируется ошибка, выполнение немедленно переносится в условия catch, которые принимают решение о продолжении передачи ошибки. **Если ошибка не генерируется**, остальные операторы do выполняются.

В условии catch не нужно обрабатывать все возможные ошибки, которые может вызвать код в условии do. Если ни одно из условий catch не обрабатывает ошибку, ошибка распространяется на **окружающую область**.

Синтаксис обработки ошибок с использованием do-catch	<pre> do { try выражение выражение } catch шаблон 1 { выражение } catch шаблон 2 where условие { выражение } catch шаблон 3, шаблон 4 where условие { выражение } catch { выражение } </pre>
--	--

Пример обработки ошибок с использованием do-catch	<pre> var vendingMachine = VendingMachine() vendingMachine.coinsDeposited = 8 do { try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine) } catch VendingMachineError.invalidSelection { print("Ошибка выбора.") } catch VendingMachineError.outOfStock { print("Нет в наличии.") } catch VendingMachineError.insufficientFunds(let coinsNeeded) { print("Недостаточно средств. Вставьте еще \(coinsNeeded) монетки.") } catch { print("Неожиданная ошибка: \(error).") } // Выведет "Недостаточно средств. Вставьте еще 2 монетки. </pre>
Пример где любая ошибка, которая не является VendingMachineError, захватывается вызывающей функцией	<pre> func nourish(with item: String) throws { do { try vendingMachine.vend(itemNamed: item) } catch is VendingMachineError { print("Некорректный вывод, нет в наличии или недостаточно денег.") } } do { try nourish(with: "Beet-Flavored Chips") } catch { print("Unexpected non-vending-machine-related error: \(error)") } // Выведет "Некорректный вывод, нет в наличии или недостаточно денег." </pre>

Преобразование ошибок в опциональные значения

Вы можете *использовать try?* для обработки ошибки, преобразовав ее в опциональное значение. Если ошибка генерируется при условии try?, то значение выражения вычисляется как nil.

Использование try? позволяет написать краткий код обработки ошибок, если вы хотите обрабатывать все ошибки таким же образом.

В примере x и y имеют одинаковые значения и поведение	<pre> func someThrowingFunction() throws -> Int { // ... } let x = try? someThrowingFunction() let y: Int? do { y = try someThrowingFunction() } catch { y = nil } </pre>
Код использует несколько попыток для извлечения данных или возвращает nil, если попытки неудачные	<pre> func fetchData() -> Data? { if let data = try? fetchDataFromDisk() { return data } if let data = try? fetchDataFromServer() { return data } return nil } </pre>

Запрет на передачу ошибок

Когда вы знаете, что функции throw или методы *не сгенерируют ошибку* во время исполнения. В этих случаях, вы можете написать *try!* перед выражением для запрета передачи ошибки и завернуть вызов в утверждение того, что ошибка точно не будет сгенерирована

Пример запрета на передачу ошибки	<pre> let photo = try! loadImage(atPath: "../Resources/John Appleseed.jpg") </pre>
-----------------------------------	--

Установка действий по очистке (Cleanup)

Вы используете *оператор defer* для выполнения набора инструкций перед тем как исполнение кода оставит текущий блок. Это позволяет *сделать любую необходимую очистку*, которая должна быть выполнена, независимо от того, как именно это произойдет — либо он покинет из-за сгенерированной ошибки или из-за оператора, такого как break или return.

Оператор defer *откладывает выполнение*, пока не происходит выход из текущей области. Этот оператор состоит из ключевого слова defer и выражений, которые должны быть выполнены позже.

Отложенные действия **выполняются в обратном порядке**, как они указаны, то есть, код в первом операторе defer выполняется после кода второго, и так далее.

Вы **можете использовать** оператор defer, даже если не используете кода обработки ошибок.

Пример использования оператора defer	<pre>func processFile(filename: String) throws { if exists(filename) { let file = open(filename) defer { close(file) } while let line = try file.readline() { // работаем с файлом. } // close(file) вызывается здесь, в конце зоны видимости. } }</pre>
--------------------------------------	--

18. Согласованность

Swift имеет встроенную поддержку для структурированного написания асинхронного и параллельного кода.

Асинхронный код можно приостановить и возобновить позже, хотя одновременно выполняется только одна часть программы.

Параллельный код означает одновременное выполнение нескольких фрагментов кода - например, компьютер с четырехъядерным процессором может одновременно запускать четыре фрагмента кода, при этом каждое ядро выполняет одну из задач.

Если вы раньше писали параллельный код, возможно, вы привыкли работать с потоками. Модель параллелизма в Swift построена на основе потоков, но вы не взаимодействуете с ними напрямую. Асинхронная функция в Swift может отказаться от потока, в котором она выполняется, что позволяет другой асинхронной функции работать в этом потоке, пока первая функция заблокирована.

Определение и вызов асинхронных функций

Асинхронная функция или асинхронный метод - это особый вид функции или метода, которые можно приостановить на полпути выполнения. Это отличается от обычных синхронных функций и методов, которые либо выполняются до завершения, либо вызывают ошибку, либо никогда не возвращаются. Асинхронная функция или метод по-прежнему выполняет одно из этих трех действий, но может также останавливаться посередине, когда чего-то ожидает.

Внутри тела асинхронной функции или метода вы отмечаете каждое из этих мест, где выполнение может быть приостановлено.

Чтобы указать, что функция или метод является асинхронным, вы пишете ключевое слово **async** в его объявлении после его параметров

Пример асинхронной функции	<pre>func listPhotos(inGallery name: String) async -> [String] { let result = // ... some asynchronous networking code ... return result }</pre>
----------------------------	---

Для функции или метода, которые одновременно являются как асинхронными, так и исключаящими (**throw**), вы пишете **async перед throw**.

При вызове асинхронного метода выполнение приостанавливается до тех пор, пока этот метод не вернется. Вы пишете **await** перед вызовом, чтобы отметить возможную точку приостановки.

Внутри асинхронного метода поток выполнения приостанавливается только тогда, когда вы вызываете другой асинхронный метод - **приостановка никогда не бывает неявной или упреждающей** - это означает, что каждая возможная точка приостановки помечается с помощью **await**.

Пример асинхронной функции	<pre>let photoNames = await listPhotos(inGallery: "Summer Vacation") let sortedNames = photoNames.sorted() let name = sortedNames[1] let photo = await downloadPhoto(named: name) show(photo)</pre>
----------------------------	---

Точки приостановки в вашем коде, **отмеченные значком await, указывают на то**, что текущий фрагмент кода может приостановить выполнение, ожидая возврата асинхронной функции или метода. Это также называется уступкой потока, потому что за кулисами Swift приостанавливает выполнение вашего кода в текущем потоке и вместо этого запускает другой код в этом потоке. Поскольку код с **await** должен иметь возможность приостанавливать выполнение, только определенные места в вашей программе могут вызывать асинхронные функции или методы:

- Код в теле асинхронной функции, метода или свойства.
- Код в статическом методе **main()** структуры, класса или перечисления, помеченных **@main**.
- Код в отдельной дочерней задаче, как показано в разделе «Неструктурированный параллелизм».

Асинхронные последовательности

Цикл for-await-in потенциально приостанавливает выполнение в начале каждой итерации, когда он ожидает, когда будет доступен следующий элемент.

Точно так же, как вы можете использовать свои **собственные типы в цикле for-in**, добавив соответствие протоколу **Sequence**, вы можете использовать свои **собственные типы в цикле for-await-in**, добавив соответствие протоколу **AsyncSequence**.

Пример последовательности	асинхронной	<pre>import Foundation let handle = FileHandle.standardInput for try await line in handle.bytes.lines { print(line) }</pre>
---------------------------	-------------	---

Параллельный вызов асинхронных функций

Вызов асинхронной функции с помощью ***await*** **запускает только один фрагмент кода за раз**. Пока выполняется асинхронный код, вызывающая сторона ожидает завершения этого кода, прежде чем перейти к выполнению следующей строки кода.

У примера справа есть важный недостаток: несмотря на то, что загрузка является асинхронной и позволяет выполнять другую работу во время ее выполнения, одновременно выполняется только один вызов функции <code>downloadPhoto(named: :)</code> . Каждая фотография полностью загружается до начала загрузки следующей. Однако этим операциям не нужно ждать - каждую фотографию можно загружать независимо или даже одновременно.	<pre>let firstPhoto = await downloadPhoto(named: photoNames[0]) let secondPhoto = await downloadPhoto(named: photoNames[1]) let thirdPhoto = await downloadPhoto(named: photoNames[2]) let photos = [firstPhoto, secondPhoto, thirdPhoto] show(photos)</pre>
---	--

Чтобы вызвать асинхронную функцию и позволить ей работать параллельно с кодом вокруг нее, напишите `async` перед `let` при определении константы, а затем напишите `await` каждый раз, когда вы используете константу.

В этом примере все три вызова <code>downloadPhoto(named: :)</code> запускаются без ожидания завершения предыдущего. Если доступно достаточно системных ресурсов, они могут работать одновременно. Ни один из этих вызовов функций не помечен как <code>await</code> , потому что код не приостанавливается в ожидании результата функции. Вместо этого выполнение продолжается до тех пор, пока не будет определена строка, в которой определены фотографии - в этот момент программе требуются результаты этих асинхронных вызовов, поэтому вы пишете <code>await</code> , чтобы приостановить выполнение, пока не завершится загрузка всех трех фотографий.	<pre>async let firstPhoto = downloadPhoto(named: photoNames[0]) async let secondPhoto = downloadPhoto(named: photoNames[1]) async let thirdPhoto = downloadPhoto(named: photoNames[2]) let photos = await [firstPhoto, secondPhoto, thirdPhoto] show(photos)</pre>
---	--

Различия между этими двумя вышеописанными подходами:

- Вызов асинхронных функций с помощью `await`, когда код в следующих строках зависит от результата этой функции. Это создает работу, которая выполняется последовательно.
- Вызывайте асинхронные функции с помощью `async-let`, если вам не нужен результат до тех пор, пока не появится код. Это создает работу, которую можно выполнять параллельно.
- Оба `await` и **`async-let`** позволяют запускать другой код, пока они приостановлены.
- В обоих случаях вы отмечаете возможную точку приостановки с помощью `await`, чтобы указать, что выполнение будет приостановлено, если необходимо, до тех пор, пока асинхронная функция не вернется.

Задачи и группы задач

Задача - это единица работы, которая может выполняться асинхронно как часть вашей программы. Весь асинхронный код выполняется как часть некоторой задачи.

Вы также можете создать **группу задач** и добавить в нее **дочерние задачи**, что дает вам больше контроля над приоритетом и отменой, а также позволяет создавать динамическое количество задач.

Задачи расположены в иерархии. Каждая задача в группе задач имеет одну и ту же **родительскую задачу**, и каждая задача может иметь **дочерние задачи**. Из-за явной взаимосвязи между задачами и группами задач этот подход называется **структурированным параллелизмом**.

Примеры задач и группы задач	<pre>await withTaskGroup(of: Data.self) { taskGroup in let photoNames = await listPhotos(inGallery: "Summer Vacation") for name in photoNames { taskGroup.async { await downloadPhoto(named: name) } } }</pre>
------------------------------	--

Неструктурированный параллелизм

Неструктурированная задача не имеет родительской задачи

Чтобы создать неструктурированную задачу, выполняемую текущим актором, вызовите функцию **`async(priority: operation :)`**. Чтобы создать неструктурированную задачу, которая не является частью текущего актора, более конкретно называемую отдельной задачей, вызовите **`asyncDetached(priority: operation :)`**. Обе эти функции возвращают дескриптор задачи, который позволяет вам взаимодействовать с задачей, например, дождаться ее результата или отменить его.

Неструктурированная задача	<pre>let newPhoto = // ... какие-то данные по фото ... let handle = async { return await add(newPhoto, toGalleryNamed: "Spring Adventures") } let result = await handle.get()</pre>
----------------------------	---

Отмена задачи

Параллелизм в Swift использует **кооперативную модель отмены**. Каждая задача проверяет, была ли она отменена в соответствующие моменты ее выполнения, и реагирует на отмену любым подходящим способом. В зависимости от выполняемой вами работы это обычно означает одно из следующего:

- Выдает ошибку, например `CancellationError`
- Возврат `nil` или пустой коллекции
- Возврат частично выполненной работы

Чтобы проверить отмену, либо вызовите **`Task.checkCancellation()`**, который выбрасывает **`CancellationError`**, если задача была отменена, либо проверьте значение **`Task.isCancelled`** и обработайте отмену в своем собственном коде.

Чтобы распространить отмену вручную, вызовите **`Task.Handle.cancel()`**.

Акторы

Как и классы, **акторы являются ссылочными типами**, поэтому сравнение типов значений и ссылочных типов применяется как к акторам, так и к классам. В отличие от классов, акторы позволяют только одной задаче получать доступ к своему изменяемому состоянию за раз, что делает безопасным взаимодействие кода в нескольких задачах с одним и тем же экземпляром актора.

Вы вводите актора с ключевым словом `actor`, за которым следует его определение в фигурных скобках. Вы создаете экземпляр актора, используя тот же синтаксис инициализатора, что и структуры и классы. Когда вы обращаетесь к свойству или методу актора, вы используете `await`, чтобы отметить потенциальную точку приостановки. Акторы позволяют одновременно взаимодействовать со своим изменяемым состоянием только одной задаче: некоторые обновления состояния актора временно нарушают инварианты.

Актор, который записывает температуру	<pre>actor TemperatureLogger { let label: String var measurements: [Int] private(set) var max: Int init(label: String, measurement: Int) { self.label = label self.measurements = [measurement] self.max = measurement } } let logger = TemperatureLogger(label: "Outdoors", measurement: 25) print(await logger.max) // Выведет "25" extension TemperatureLogger { func update(with measurement: Int) { measurements.append(measurement) if measurement > max { max = measurement } } } print(logger.max) // Ошибка</pre>
---------------------------------------	--

Доступ к `logger.max` без записи `await` завершается ошибкой, поскольку свойства субъекта являются частью изолированного локального состояния этого субъекта. Swift гарантирует, что только код внутри актора может получить доступ к локальному состоянию актора. Эта гарантия известна как изоляция актора.

19. Приведение типов

Приведение типов - это способ проверить тип экземпляра и/или способ обращения к экземпляру так, как если бы он был экземпляром суперкласса или подкласса откуда-либо из своей собственной классовой иерархии.

Приведение типов в Swift реализуется с помощью операторов **is** и **as**.

Можно использовать приведение типов **для проверки** соответствия типа протоколу.

Определение классовой иерархии для приведения типов

Можно использовать приведение типов **с иерархией классов и подклассов**, чтобы проверить тип конкретного экземпляра класса и преобразовать тип этого экземпляра в тип другого класса в той же иерархии.

<p>Тип library выведен во время инициализации массива литералом массива. Механизм проверки типов Swift делает вывод, что Movie, Song имеют общий суперкласс MediaItem, так что тип массива library становится [MediaItem]</p> <p>Элементы, которые хранятся в library все еще экземпляры Movie и Song на самом деле. Однако, если вы переберете элементы массива, то они все будут одного типа MediaItem, а не Movie или Song. Для того чтобы работать с ними как с исходными типами, вам нужно проверить их типы или привести к другому типу, как указано далее.</p>	<pre>class MediaItem { var name: String init(name: String) { self.name = name } } class Movie: MediaItem { var director: String init(name: String, director: String) { self.director = director super.init(name: name) } } class Song: MediaItem { var artist: String init(name: String, artist: String) { self.artist = artist super.init(name: name) } } let library = [Movie(name: "Casablanca", director: "Michael Curtiz"), Song(name: "Blue Suede Shoes", artist: "Elvis Presley"), Movie(name: "Citizen Kane", director: "Orson Welles"), Song(name: "The One And Only", artist: "Chesney Hawkes"), Song(name: "Never Gonna Give You Up", artist: "Rick Astley")] // тип "library" выведен как [MediaItem]</pre>
--	---

Проверка типа

Используйте **оператор проверки типа (is)** для проверки того, соответствует ли тип экземпляра типам какого-то определенного подкласса. Оператор проверки типа **возвращает** true, если экземпляр имеет тип конкретного подкласса, false, если нет.

<p>Пример использования оператора проверки типа (is)</p>	<pre>var movieCount = 0 var songCount = 0 for item in library { if item is Movie { movieCount += 1 } else if item is Song { songCount += 1 } } print("В Media содержится \(movieCount) фильма и \(songCount) песни") // Выведет "В Media библиотеке содержится 2 фильма и 3 песни"</pre>
---	---

Понижающее приведение

Можно попробовать **привести тип к типу подкласса** при помощи оператора понижающего приведения (**as?** или **as!**).

Из-за того, что **понижающее приведение** может провалиться, оператор приведения имеет **две формы**. Опциональная форма (**as?**), которая возвращает опциональное значение типа, к которому вы пытаетесь привести. И принудительная форма (**as!**), которая принимает попытки понижающего приведения и принудительного разворачивания результата в рамках одного составного действия.

Используйте **опциональную форму оператора понижающего приведения (*as?*)**, когда вы не уверены, что ваше понижающее приведение выполнится успешно. В этой форме оператор всегда будет возвращать опциональное значение, и значение будет nil, если понижающее приведение будет не выполнимо.

Используйте **принудительную форму оператора понижающего приведения (*as!*)**, но только в тех случаях, когда вы точно уверены, что понижающее приведение будет выполнено успешно.

Приведение **не изменяет** экземпляра или его значений. Первоначальный экземпляр остается тем же. Просто после приведения типа с экземпляром можно обращаться (и использовать свойства) именно так как с тем типом, к которому его привели.

Пример использования понижающего приведения	<pre>for item in library { if let movie = item as? Movie { print("Movie: \(movie.name), dir. \(movie.director)") } else if let song = item as? Song { print("Song: \(song.name), by \(song.artist)") } } // Movie: Casablanca, dir. Michael Curtiz // Song: Blue Suede Shoes, by Elvis Presley // Movie: Citizen Kane, dir. Orson Welles // Song: The One And Only, by Chesney Hawkes // Song: Never Gonna Give You Up, by Rick Astley</pre>
---	--

Приведение типов для Any и AnyObject

Swift предлагает **две версии псевдонимов типа** для работы с неопределенными типами:

- Any может отобразить экземпляр любого типа, включая функциональные типы.
- AnyObject может отобразить экземпляр любого типа класса.

Используйте Any и AnyObject только тогда, когда вам явно нужно поведение и особенности, которые они предоставляют. Всегда лучше быть конкретным насчет типов, с которыми вы ожидаете работать в вашем коде.

Тип Any представляет собой значения любого типа, включая и опциональные типы. Swift предупредит вас, если вы используете опциональное значение в том месте, где ожидается тип Any. Если вы действительно хотите использовать опциональное значение в виде значения типа Any, то вы можете использовать оператор **as**, чтобы явно привести опционал к Any.

Пример использования Any с различными типами	<pre>var things = [Any]() things.append(0) things.append(0.0) things.append(42) things.append(3.14159) things.append("hello") things.append((3.0, 5.0)) things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman")) things.append({ (name: String) -> String in "Hello, \(name)" }) for thing in things { switch thing { case 0 as Int: print("zero as an Int") case 0 as Double: print("zero as a Double") case let someInt as Int: print("an integer value of \(someInt)") case let someDouble as Double where someDouble > 0: print("a positive double value of \(someDouble)") case is Double: print("some other double value that I don't want to print") case let someString as String: print("a string value of \"\(someString)\"") case let (x, y) as (Double, Double): print("an (x, y) point at \(x), \(y)") case let movie as Movie: print("a movie called \(movie.name), dir. \(movie.director)") case let stringConverter as (String) -> String: print(stringConverter("Michael")) default: print("something else") } }</pre>
--	--

	<pre>} } // zero as an Int // zero as a Double // an integer value of 42 // a positive double value of 3.14159 // a string value of "hello" // an (x, y) point at 3.0, 5.0 // a movie called Ghostbusters, dir. Ivan Reitman // Hello, Michael</pre>
Приведение опциона к Any, через оператор as	<pre>let optionalNumber: Int? = 3 things.append(optionalNumber) // Warning things.append(optionalNumber as Any) // No warning</pre>

20. Вложенные типы

Вложенные типы это типы в которые вкладываются вспомогательные перечисления, классы и структуры, внутри определения типа, которые они поддерживают.

Для того, чтобы использовать вложенные типы **снаружи определяющего их контекста**, нужно поставить префикс имени типа, внутри которого он вложен, затем его имя

Пример использования вложенных типов	<pre>struct BlackjackCard { // nested Suit enumeration enum Suit: Character { case spades = "♠", hearts = "♥", diamonds = "♦", clubs = "♣" } // nested Rank enumeration enum Rank: Int { case two = 2, three, four, five, six, seven, eight, nine, ten case jack, queen, king, ace struct Values { let first: Int, second: Int? } var values: Values { switch self { case .ace: return Values(first: 1, second: 11) case .jack, .queen, .king: return Values(first: 10, second: nil) default: return Values(first: self.rawValue, second: nil) } } } // BlackjackCard properties and methods let rank: Rank, suit: Suit var description: String { var output = "suit is \(suit.rawValue)," output += " value is \(rank.values.first)" if let second = rank.values.second { output += " or \(second)" } return output } } let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades) print("theAceOfSpades: \(theAceOfSpades.description)") // Выведет "theAceOfSpades: suit is ♠, value is 1 or 11"</pre>
Использование вложенных типов снаружи определяющего их контекста	<pre>let heartsSymbol = BlackjackCard.Suit.hearts.rawValue // heartsSymbol равен "♥"</pre>

21. Расширения

Расширения добавляют новую функциональность существующему типу класса, структуры или перечисления. Это включает в себя возможность расширять типы, к исходным кодам которых у вас нет доступа (известно как **ретроактивное моделирование**).

Расширения в Swift могут:

- Добавлять вычисляемые свойства и вычисляемые свойства типа
- Определять методы экземпляра и методы типа
- Предоставлять новые инициализаторы
- Определять сабскрипты (индексы)
- Определять новые вложенные типы
- Обеспечить соответствие существующего типа протоколу

Расширения **могут** добавлять новую функциональность типу, но они **не могут** переписать существующую функциональность.

Расширение объявляется с помощью ключевого слова **extension**.

Если вы определяете расширение для добавления новой функциональности **существующему типу**, то новая функциональность будет **доступна всем экземплярам** этого типа, даже если они были созданы до того, как было определено расширение.

Синтаксис расширения	<pre>extension SomeType { // описываем новую функциональность для типа SomeType }</pre>
Расширение протокола что бы он соответствовал протоколу	<pre>extension SomeType: SomeProtocol, AnotherProtocol { // реализация требования протокола тут }</pre>

Вычисляемые свойства в расширениях

Расширения могут добавлять новые вычисляемые свойства, но они **не могут добавить** свойства хранения или наблюдателей свойства к уже существующим свойствам.

Добовление вычисляемых свойств типу через расширение	<pre>extension Double { var km: Double { return self * 1_000.0 } var m: Double { return self } var cm: Double { return self / 100.0 } var mm: Double { return self / 1_000.0 } var ft: Double { return self / 3.28084 } } let oneInch = 25.4.mm print("Один дюйм - это \(oneInch) метра") // Выведет "Один дюйм- это 0.0254 метра" let threeFeet = 3.ft print("Три фута - это \(threeFeet) метра") // Выведет "Три фута - это 0.914399970739201 метра" let aMarathon = 42.km + 195.m print("Марафон имеет длину \(aMarathon) метров") // Выведет "Марафон имеет длину 42195.0 метров"</pre>
--	---

Инициализаторы в расширениях

Расширения могут добавлять **вспомогательные инициализаторы** классу, но они не могут добавить новый **назначенный инициализатор** или **деинициализатор классу**. Назначенные инициализаторы и деинициализаторы должны всегда предоставляться реализацией исходного класса.

Если вы используете расширения для того, чтобы добавить инициализатор к типу значений, который обеспечивает значения по умолчанию для всех своих хранимых свойств и не определяет какого-либо пользовательского инициализатора, то **вы можете вызвать дефолтный инициализатор и почленный инициализатор** для того типа значений изнутри инициализатора вашего расширения. Это **не будет работать**, если вы уже написали инициализатор как часть исходной реализации значения типа.

Если вы используете расширение для добавления инициализатора в структуру, которая была **объявлена в другом модуле**, новый инициализатор не может получить доступ к себе до тех пор, пока он не вызовет инициализатор из модуля определения.

Если вы предоставляете новый инициализатор вместе с расширением, **вы все еще ответственны за то**, что каждый экземпляр должен быть полностью инициализирован, к моменту, когда инициализатор заканчивает свою работу.

Добавление инициализатора через расширение.	<pre>struct Size { var width = 0.0, height = 0.0 } struct Point { var x = 0.0, y = 0.0 } struct Rect { var origin = Point() var size = Size() } let defaultRect = Rect() let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0), size: Size(width: 5.0, height: 5.0)) extension Rect { init(center: Point, size: Size) { let originX = center.x - (size.width / 2) let originY = center.y - (size.height / 2) self.init(origin: Point(x: originX, y: originY), size: size) } } let centerRect = Rect(center: Point(x: 4.0, y: 4.0), size: Size(width: 3.0, height: 3.0)) // исходная точка centerRect (2.5, 2.5) и его размер (3.0, 3.0)</pre>
---	---

Методы в расширениях

Методы экземпляров, добавленные в расширении так же **могут менять и сам экземпляр**. Методы структуры и перечисления, которые изменяют self или его свойства, **должны быть отмечены как mutating**.

Добавление метода через расширение	<pre>extension Int { func repetitions(task: () -> Void) { for _ in 0..<self 2.repetitions="" hello!="" hello!<="" pre="" print("hello!")="" task()="" {="" }=""></self></pre>
Добавление метода изменяющий экземпляр	<pre>extension Int { mutating func square() { self = self * self } } var someInt = 3 someInt.square() // теперь переменная someInt имеет значение 9</pre>

Сабскрипты в расширениях

Добавление сабскрипта через расширение	<pre>extension Int { subscript(digitIndex: Int) -> Int { var decimalBase = 1 for _ in 0..<digitindex %="" (self="" *="10" 10="" 746381295[0]<="" decimalbase="" decimalbase)="" pre="" return="" {="" }=""></digitindex></pre>
--	---

```
// возвращает 5
746381295[1]
// возвращает 9
746381295[2]
// возвращает 2
746381295[8]
// возвращает 7
```

Вложенные типы в расширениях

Добавление вложенных типов через расширение

`number.kind` имеет тип `Int.Kind`. Значит все значения членов `Int.Kind` могут быть записаны в короткой форме внутри конструкции `switch`, как `.negative`, а не `Int.Kind.negative`

```
extension Int {
  enum Kind {
    case negative, zero, positive
  }
  var kind: Kind {
    switch self {
    case 0:
      return .zero
    case let x where x > 0:
      return .positive
    default:
      return .negative
    }
  }
}

func printIntegerKinds(_ numbers: [Int]) {
  for number in numbers {
    switch number.kind {
    case .negative:
      print("- ", terminator: "")
    case .zero:
      print("0 ", terminator: "")
    case .positive:
      print("+ ", terminator: "")
    }
  }
  print("")
}

printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// Выведет "+ + - 0 - 0 + "
```

22. Непрозрачные типы

Функция или метод с непрозрачным типом возвращаемого значения скрывает информацию о типе своего возвращаемого значения. Вместо того, чтобы указывать конкретный тип в качестве типа возвращаемого значения функции, возвращаемое значение описывается в терминах поддерживаемых им протоколов.

Скрытие информации о типе **полезно** на границах между модулем и кодом, который вызывает модуль, поскольку базовый тип возвращаемого значения может оставаться закрытым.

В отличие от возврата значения, тип которого является типом протокола, непрозрачные типы **сохраняют** идентичность типа - компилятор имеет доступ к информации о типе, но клиенты модуля - нет.

Непрозрачный тип позволяет реализации функции выбирать тип для возвращаемого значения таким образом, чтобы абстрагироваться от кода, вызывающего функцию.

<p>Проблема, которую решают непрозрачные типы</p> <p>Использование универсального шаблона в FlippedShape дает ограничение: перевернутый результат показывает точные универсальные типы, которые использовались для его создания.</p> <p>Этот подход к определению структуры JoinedShape<T: Shape, U: Shape>, которая объединяет две фигуры вместе по вертикали, как показано в приведенном коде, приводит к таким типам, как JoinedShape<FlippedShape<Triangle>, Triangle></p> <p>Предоставление подробной информации о создании формы позволяет типам, которые не предназначены для использования в общедоступном интерфейсе, просачиваться наружу из-за необходимости указывать полный тип возвращаемого значения.</p> <p>Код внутри модуля может создавать одну и ту же форму различными способами, и другой код вне модуля, который использует эту форму, не должен учитывать детали реализации списка преобразований. Оберточные типы, такие как JoinedShape и FlippedShape, не имеют значения для пользователей модуля и не должны быть видны.</p>	<pre>protocol Shape { func draw() -> String } struct Triangle: Shape { var size: Int func draw() -> String { var result: [String] = [] for length in 1...size { result.append(String(repeating: "*", count: length)) } return result.joined(separator: "\n") } } let smallTriangle = Triangle(size: 3) print(smallTriangle.draw()) // * // ** // *** struct FlippedShape<T: Shape>: Shape { var shape: T func draw() -> String { let lines = shape.draw().split(separator: "\n") return lines.reversed().joined(separator: "\n") } } let flippedTriangle = FlippedShape(shape: smallTriangle) print(flippedTriangle.draw()) // *** // ** // * struct JoinedShape<T: Shape, U: Shape>: Shape { var top: T var bottom: U func draw() -> String { return top.draw() + "\n" + bottom.draw() } } let joinedTriangles = JoinedShape(top: smallTriangle, bottom: flippedTriangle) print(joinedTriangles.draw()) // * // ** // *** // *** // ** // *</pre>
---	--

Можно комбинировать непрозрачные возвращаемые типы с универсальными.

<p>Примеры универсальных функций с непрозрачными возвращаемыми типами</p>	<pre>func flip<T: Shape>(_ shape: T) -> some Shape { return FlippedShape(shape: shape) } func join(_ top: T, _ bottom: U) -> some Shape { JoinedShape(top: top, bottom: bottom) }</pre>
--	--

```

}

let opaqueJoinedTriangles = join(smallTriangle, flip(smallTriangle))
print(opaqueJoinedTriangles.draw())
// *
// **
// ***
// ***
// **
// *

```

Если функция с возвращаемым непрозрачным типом **возвращается из нескольких мест**, все возможные возвращаемые значения должны иметь один и тот же тип. Для универсальной функции этот возвращаемый тип может использовать параметры универсального типа функции, но он все равно должен быть одного типа.

Если вы вызываете эту функцию с помощью Square, она возвращает Square; в противном случае она возвращает FlippedShape. Это нарушает требование возвращать значения только одного типа и делает код invalidFlip(_ :) недопустимым	<pre> func invalidFlip<T: Shape>(_ shape: T) -> some Shape { if shape is Square { return shape // Ошибка: несоответствующий возвращаемый тип } return FlippedShape(shape: shape) // Ошибка: несоответствующий возвращаемый тип } </pre>
---	--

Требование всегда возвращать один тип не мешает вам **использовать универсальные шаблоны** в непрозрачном возвращаемом типе.

В этом случае базовый тип возвращаемого значения зависит от T: какая бы фигура ни была передана, repeat(shape: count :) создает и возвращает массив этой формы. Тем не менее, возвращаемое значение всегда имеет один и тот же базовый тип [T]	<pre> func `repeat`<T: Shape>(shape: T, count: Int) -> some Collection { return Array(repeating: shape, count: count) } </pre>
--	---

Различия между типом протокола и непрозрачным типом

Возврат непрозрачного типа очень похож на использование типа протокола в качестве типа возвращаемого значения функции, но эти два вида возвращаемого типа различаются тем, что **по-разному работают с идентичностью типа**.

Непрозрачный тип **относится к одному конкретному типу**, хотя вызывающая функция не может видеть конкретно что это за тип.

Тип протокола может **относиться к любому типу**, который соответствует протоколу.

Типы протоколов дают вам **больше гибкости** в отношении базовых типов значений, которые они хранят, а непрозрачные типы позволяют вам делать **более строгие гарантии** в отношении этих базовых типов.

Пример функции, которая использует тип протокола в качестве возвращаемого типа вместо непрозрачного типа возврата	<pre> func protoFlip<T: Shape>(_ shape: T) -> Shape { return FlippedShape(shape: shape) } </pre>
Так как функция возвращает тип протокола, то она может возвращать значения разных типов, но соответствующих одному протоколу	<pre> func protoFlip<T: Shape>(_ shape: T) -> Shape { if shape is Square { return shape } return FlippedShape(shape: shape) } </pre>

Использование типа протокола в качестве типа возвращаемого значения для функции дает вам возможность возвращать любой тип, соответствующий протоколу. **Однако цена такой гибкости** заключается в том, что некоторые операции с возвращаемыми значениями невозможны. Например, **оператор == недоступен** - это зависит от конкретной информации о типе, которая не сохраняется при использовании типа протокола.

Еще одна проблема с этим подходом заключается в том, что преобразования формы **не вкладываются**. Результатом переворота треугольника является значение типа Shape, а функция protoFlip(_ :) принимает аргумент некоторого типа, который соответствует протоколу Shape. Однако значение типа протокола не соответствует этому протоколу; значение, возвращаемое protoFlip(_ :), не соответствует Shape. Это означает, что такой код, как protoFlip(protoFlip (smallTriange)), который применяет несколько преобразований, недействителен, поскольку перевернутая форма не является допустимым аргументом для protoFlip(_ :).

Swift может определять связанные типы, что позволяет использовать непрозрачное возвращаемое значение в тех местах, где тип протокола не может использоваться в качестве возвращаемого значения.

<p>Вы не можете использовать Container в качестве возвращаемого типа функции, потому что у этого протокола есть связанный тип.</p> <p>Вы также не можете использовать его в качестве ограничения в универсальном возвращаемом типе, потому что за пределами тела функции недостаточно информации, чтобы сделать вывод, каким должен быть универсальный тип.</p>	<pre>protocol Container { associatedtype Item var count: Int { get } subscript(i: Int) -> Item { get } } extension Array: Container { }</pre> <p>// Ошибка: Протоколы со связанными типами не могут быть использованы в качестве возвращаемого типа.</p> <pre>func makeProtocolContainer(item: T) -> Container { return [item] }</pre> <p>// Ошибка: Не достаточно информации для определения типа C.</p> <pre>func makeProtocolContainer<T, C: Container>(item: T) -> C { return [item] }</pre>
<p>Использование непрозрачного типа some Container в качестве возвращаемого типа выражает желаемый контракт API - функцию возвращающую контейнер, но не указывающую его тип.</p> <p>Тип значения twelve считается Int, что иллюстрирует тот факт, что вывод типа работает с непрозрачными типами. В реализации makeOpaqueContainer(item :) базовый тип непрозрачного контейнера - [T]</p>	<pre>func makeOpaqueContainer<T>(item: T) -> some Container { return [item] } let opaqueContainer = makeOpaqueContainer(item: 12) let twelve = opaqueContainer[0] print(type(of: twelve)) // Выведет "Int"</pre>

23. Протоколы

Протокол определяет образец методов, свойств или другие требования, которые соответствуют определенному конкретному заданию или какой-то функциональности. **Протокол фактически** не предоставляет реализацию для любого из этих требований, он только описывает как реализация должна выглядеть.

Протокол может быть принят классом, структурой или перечислением для обеспечения фактической реализации этих требований.

Синтаксис протокола

Определение протокола	<pre>protocol SomeProtocol { // определение протокола... }</pre>
Принятие протокола	<pre>struct SomeStructure: FirstProtocol, AnotherProtocol { // определение структуры... }</pre>
Если у класса есть суперкласс, то вписывайте имя суперкласса до списка протоколов, которые он принимает	<pre>class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol { // определение класса... }</pre>

Требование свойств

Протокол требует у соответствующего ему типа предоставить свойство экземпляра или свойство типа, и это свойство должно иметь конкретное имя и тип. **Протокол не уточняет** какое должно быть свойство, хранимое или вычисляемое, только лишь указывает на требование имени свойства и типа. Протокол уточняет должно свойство ли быть доступным, или оно должно быть доступным и устанавливаемым.

Требуемые свойства всегда объявляются как переменные свойства, с префиксом **var**. Свойства, значения которых вы можете получить или изменить маркируются **{ get set }** после объявления типа свойства, а свойства, значения которых мы можем только получить, но не изменить **{ get }**.

Требование по предоставлению свойств экземпляра	<pre>protocol SomeProtocol { var mustBeSettable: Int { get set } var doesNotNeedToBeSettable: Int { get } }</pre>
Требование по предоставлению свойств типов	<pre>protocol AnotherProtocol { static var someTypeProperty: Int { get set } }</pre>
Пример структуры, которая принимает и полностью соответствует протоколу	<pre>struct Person: FullyNamed { var fullName: String } let john = Person(fullName: "John Appleseed") // john.fullName равен "John Appleseed"</pre>

Требование методов

Протоколы могут требовать реализацию определенных методов экземпляра и методов типа, соответствующими типами протоколу. Эти методы написаны как часть определения протокола в точности в такой же форме как и методы экземпляра или типа, но только в них отсутствуют фигурные скобки или тело метода целиком. Вариативные параметры допускаются точно так же как и в обычных методах. Дефолтные значения, однако, не могут быть указаны для параметров метода внутри определения протокола.

Требование к методу экземпляра	<pre>protocol RandomNumberGenerator { func random() -> Double }</pre>
Требование к методу типа	<pre>protocol SomeProtocol { static func someTypeMethod() }</pre>
Пример класса который реализует алгоритм генератора псевдослучайных чисел, известный как алгоритм линейного конгруэнтного генератора	<pre>class LinearCongruentialGenerator: RandomNumberGenerator { var lastRandom = 42.0 let m = 139968.0 let a = 3877.0 let c = 29573.0 func random() -> Double { lastRandom = ((lastRandom * a + c).truncatingRemainder(dividingBy:m)) } }</pre>

	<pre> return lastRandom / m } } let generator = LinearCongruentialGenerator() print("Here's a random number: \"(generator.random())\"") // Выведет "Случайное число: 0.37464991998171" print("And another one: \"(generator.random())\"") // Выведет "Другое случайное число: 0.729023776863283"</pre>
--	---

Требования изменяющих методов

Иногда необходимо для метода изменить (или мутировать) экземпляр, которому он принадлежит. Для методов экземпляра типа значения (структура, перечисление) вы располагаете ключевое слово mutating до слова метода func, для индикации того, что этому методу разрешено менять экземпляр, которому он принадлежит, и/или любое свойство этого экземпляра.

Если вы определяете требуемый протоколом метод экземпляра, который предназначен менять экземпляры любого типа, которые принимают протокол, то поставьте ключевое слово mutating перед именем метода, как часть определения протокола. Это позволяет структурам и перечислениями принимать протокол и удовлетворять требованию метода.

Протокол с методом который может изменить экземпляр которому принадлежит этот метод	<pre>protocol Togglable { mutating func toggle() }</pre>
Пример реализации протокола с mutating	<pre>enum OnOffSwitch: Togglable { case off, on mutating func toggle() { switch self { case .off: self = .on case .on: self = .off } } } var lightSwitch = OnOffSwitch.off lightSwitch.toggle() // lightSwitch теперь равен .on</pre>

Требование инициализатора

Пример протокола который требует реализацию конкретного инициализатора типами соответствующими протоколу	<pre>protocol SomeProtocol { init(someParameter: Int) }</pre>
--	---

Вы можете реализовать требуемый инициализатор в классе, соответствующем протоколу, в качестве **назначенного** инициализатора или **вспомогательного**.

Использование модификатора required гарантирует, что вы проведете явную или унаследованную реализацию требуемого инициализатора на всех подклассах соответствующего класса протоколу, так, чтобы они тоже соответствовали протоколу.

Вам не нужно обозначать реализацию инициализаторов протокола модификатором required в классах, **где стоит модификатор final**, потому что конечные классы не могут иметь подклассы.

Реализация класса соответствующего протоколу с требованием инициализатора	<pre>class SomeClass: SomeProtocol { required init(someParameter: Int) { // реализация инициализатора... } }</pre>
Нужно отметить этот инициализатор ключевым словом required	
Если подкласс переопределяет назначенный инициализатор суперкласса и так же реализует соответствующий протоколу инициализатор, то обозначьте реализацию инициализатора сразу двумя модификаторами required и override	<pre>protocol SomeProtocol { init() } class SomeSuperClass { init() { // реализация инициализатора... } } class SomeSubClass: SomeSuperClass, SomeProtocol { // "required" от соответствия протоколу SomeProtocol; // "override" от суперкласса SomeSuperClass required override init() { // реализация инициализатора... } }</pre>

	}
	}

Протоколы могут определять требования **проваливающихся инициализаторов** для соответствующих протоколу типов.

Требование проваливающегося инициализатора может быть удовлетворено проваливающимся инициализатором или непроваливающимся инициализатором соответствующего протоколу типа.

Требование непроваливающегося инициализатора может быть удовлетворено непроваливающимся инициализатором или неявно развернутым проваливающимся инициализатором.

Протоколы как типы

Любой протокол, который вы создаете становится полноправным **типом**. Из-за того что протоколы являются типами, то их имена начинаются **с заглавной буквы**. Вы можете использовать протокол во многих местах, где можно использовать другие типы:

- Как тип параметра или возвращаемый тип в функции, методе, инициализаторе
- Как тип константы, переменной или свойства
- Как тип элементов массива, словаря или другого контейнера

Пример использования протокола в качестве типа	<pre>protocol RandomNumberGenerator { func random() -> Double } class Dice { let sides: Int let generator: RandomNumberGenerator init(sides: Int, generator: RandomNumberGenerator) { self.sides = sides self.generator = generator } func roll() -> Int { return Int(generator.random() * Double(sides)) + 1 } } var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator()) for _ in 1...5 { print("Бросок игральной кости равен \(d6.roll())") } // Бросок игральной кости равен 3 // Бросок игральной кости равен 5 // Бросок игральной кости равен 4</pre>
--	--

Делегирование

Делегирование - это шаблон, который позволяет классу или структуре передавать (или делегировать) некоторую ответственность экземпляру другого типа. **Этот шаблон реализуется** определением протокола, который инкапсулирует делегируемые полномочия, таким образом, что соответствующий протоколу тип (делегат) гарантировано получит функциональность, которая была ему делегирована.

Пример делегирования	<pre>protocol DiceGame { var dice: Dice { get } func play() } protocol DiceGameDelegate: AnyObject { func gameDidStart(_ game: DiceGame) func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) func gameDidEnd(_ game: DiceGame) } class SnakesAndLadders: DiceGame { let finalSquare = 25 let dice = Dice(sides: 6, generator: LinearCongruentialGenerator()) var square = 0 var board: [Int] init() { board = Array(repeating: 0, count: finalSquare + 1) board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02 board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08 } weak var delegate: DiceGameDelegate? func play() {</pre>
----------------------	---

```

square = 0
delegate?.gameDidStart(self)
gameLoop: while square != finalSquare {
    let diceRoll = dice.roll()
    delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
    switch square + diceRoll {
    case finalSquare:
        break gameLoop
    case let newSquare where newSquare > finalSquare:
        continue gameLoop
    default:
        square += diceRoll
        square += board[square]
    }
}
delegate?.gameDidEnd(self)
}

class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Начали новую игру Змеи и лестницы")
        }
        print("У игровой кости \ \(game.dice.sides) граней")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Выкинули \ \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("Длительность игры \ \(numberOfTurns) хода")
    }
}

let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Начали новую игру Змеи и лестницы
// У игровой кости 6 граней
// Выкинули 3
// Выкинули 5
// Выкинули 4
// Выкинули 5
// Длительность игры 4 хода

```

Добавление реализации протокола через расширение

Вы можете **расширить существующий тип** для того, чтобы он соответствовал протоколу, даже если у вас нет доступа к источнику кода для существующего типа. Расширения могут добавлять новые свойства, методы и сабскрипты существующему типу, что таким образом может удовлетворить любым требованиям протокола.

Существующие экземпляры типа **автоматически принимают и отвечают** требованиям протокола, когда опции, необходимые для соответствия добавляются через расширение типа.

Пример добавления реализации протокола через расширение

```

protocol TextRepresentable {
    var textualDescription: String { get }
}

extension Dice: TextRepresentable {
    var textualDescription: String {
        return "Игровая кость с \ \(sides) гранями"
    }
}

let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
print(d12.textualDescription)
// Выведет "Игровая кость с 12 гранями"

```

Условное соответствие протоколу

Шаблонный тип может удовлетворять требованиям протокола только при определенных условиях, например, когда общий параметр типа соответствует протоколу. Вы можете сделать **общий тип условно соответствующим протоколу**, указав ограничения при расширении типа. Напишите эти ограничения после имени протокола, который вы используете, написав оговорку **where**.

Общий тип условно соответствующий протоколу	<pre>extension Array: TextRepresentable where Element: TextRepresentable { var textualDescription: String { let itemsAsText = self.map { \$0.textualDescription } return "[" + itemsAsText.joined(separator: ", ") + "]" } } let myDice = [d6, d12] print(myDice.textualDescription) // Prints "[A 6-sided dice, A 12-sided dice]"</pre>
---	--

Принятие протокола через расширение

Если тип уже соответствует всем требованиям протокола, но еще не заявил, что он принимает этот протокол, то вы можете сделать это через **пустое расширение**. Типы не принимают протоколы автоматически, если они удовлетворяют их требованиям. Принятие протокола должно быть объявлено **в явной форме**.

Принятие протокола через расширение	<pre>struct Hamster { var name: String var textualDescription: String { return "Хомяка назвали \(name)" } } extension Hamster: TextRepresentable {} let simonTheHamster = Hamster(name: "Фруша") let somethingTextRepresentable: TextRepresentable = simonTheHamster print(somethingTextRepresentable.textualDescription) // Выведет "Хомяка назвали Фруша"</pre>
-------------------------------------	---

Принятие протокола через синтезированную реализацию

Swift может **автоматически предоставлять соответствие** таких протоколов как **Equatable**, **Hashable** и **Comparable** в большинстве простых случаев. Использование синтезированной реализации означает для нас, что мы не должны будем писать повторяющийся шаблонный код, для того, чтобы реализовать требования протокола.

Swift предоставляет синтезированную реализацию протокола **Equatable** для следующих кастомных типов:

- Структуры, которые имеют только свойства хранения и соответствуют протоколу Equatable
- Перечисления, которые имеют только ассоциативные типы и соответствуют протоколу Equatable
- Перечисления, которые не имеют ассоциативных типов

Чтобы получить **синтезированную реализацию оператора ==**, вам нужно объявить о соответствии протоколу Equatable в файле, который содержит оригинальное объявление без реализации оператора ==. По умолчанию Equatable предоставит свою дефолтную реализацию оператора !=.

Swift предоставляет синтезированную реализацию протокола **Hashable** для следующих кастомных типов:

- Структуры имеют только свойства хранения, которые соответствуют протоколу Hashable
- Перечисления, которые имеют только ассоциативные типы, которые соответствуют протоколу Hashable
- Перечисления, которые не имеют ассоциативных типов

Для получения **синтезированной реализации метода hash(into:)**, нужно объявить о соответствии протоколу Hashable в файле, который содержит оригинальное объявление без реализации метода hash(into:).

Swift предоставляет синтезированную реализацию **Comparable** для перечислений, у которых нет сырого значения (rawValue). Если перечисление имеет ассоциативные типы, то они все должны соответствовать протоколу Comparable. Для получения синтезированной реализации оператора <, объявите о соответствии протоколу Comparable в файле, который содержит оригинальное объявление перечисления, без реализации оператора <. Дефолтная реализация операторов протокола Comparable <=, > и >= предоставляет реализацию остальных операторов сравнения.

Принятие протокола через синтезированную реализацию	<pre>enum SkillLevel: Comparable { case beginner case intermediate case expert(stars: Int)</pre>
---	--


```

}
var levels = [SkillLevel.intermediate, SkillLevel.beginner,
              SkillLevel.expert(stars: 5), SkillLevel.expert(stars: 3)]
for level in levels.sorted() {
    print(level)
}
// Выведет "beginner"
// Выведет "intermediate"
// Выведет "expert(stars: 3)"
// Выведет "expert(stars: 5)"

```

Коллекции типов протокола

Протоколы могут использоваться в качестве типов, которые хранятся в таких коллекциях как массивы или словари

Создание коллекции типов протокола	<pre> protocol TextRepresentable { var textualDescription: String { get } } let things: [TextRepresentable] = [game, d12, simonTheHamster] for thing in things { print(thing.textualDescription) } // Игра Змеи и Лестницы с полем в 25 клеток // Игральная кость с 12 гранями // Хомяка назвали Фруша </pre>
------------------------------------	---

Наследование протокола

Протокол может наследовать один или более других протоколов и может добавлять требования поверх тех требований протоколов, которые он наследует. Синтаксис наследования протокола аналогичен синтаксису наследования класса, но с возможностью наследовать сразу несколько протоколов, которые разделяются между собой запятыми.

Протокол наследует несколько других протоколов	<pre> protocol InheritingProtocol: SomeProtocol, AnotherProtocol { // определение протокола... } </pre>
Наследование протокола и добавление требований	<pre> protocol PrettyTextRepresentable: TextRepresentable { var prettyTextualDescription: String { get } } extension SnakesAndLadders: PrettyTextRepresentable { var prettyTextualDescription: String { var output = textualDescription + ":\n" for index in 1...finalSquare { switch board[index] { case let ladder where ladder > 0: output += "▲ " case let snake where snake < 0: output += "▼ " default: output += "○ " } } return output } } print(game.prettyTextualDescription) // Игра Змеи и Лестницы с полем в 25 клеток: // ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ○ ▼ ○ </pre>

Классовые протоколы

Вы можете ограничить протокол так, чтобы его могли принимать только классы (но не структуры или перечисления), добавив **AnyObject** протокол к списку реализации протоколов.

Используйте протоколы class-only, когда поведение, определяемое протоколом, предполагает или требует, что соответствующий протоколу тип должен быть ссылочного типа, а не типом значения.

Объявление протокола который может быть принят только классом	<pre> protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol { // определение протокола типа class-only } </pre>
---	--

Композиция протоколов

Вы можете скомбинировать несколько протоколов в одно единственное требование при помощи **композиции протоколов**. Композиции протоколов ведут себя так, как будто вы определили временный локальный протокол, который имеет комбинированные требования ко всем протоколам в композиции. Композиции протоколов не определяют новых типов протоколов.

Композиции протоколов имеют форму `SomeProtocol & AnotherProtocol`. Вы можете перечислить столько протоколов, сколько нужно, разделяя их между собой знаком амперсанда (&). В дополнение к списку протоколов, композиция протокола также может содержать один тип класса, который можно использовать для указания требуемого суперкласса.

Пример композиции протоколов	<pre>protocol Named { var name: String { get } } protocol Aged { var age: Int { get } } struct Person: Named, Aged { var name: String var age: Int } func wishHappyBirthday(to celebrator: Named & Aged) { print("С Днем Рождения, \(celebrator.name)! Тебе уже \(celebrator.age)!") } let birthdayPerson = Person(name: "Сашка", age: 21) wishHappyBirthday(to: birthdayPerson) // Выведет "С Днем Рождения, Сашка! Тебе уже 21!"</pre>
------------------------------	--

Проверка соответствия протоколу

Вы можете использовать операторы `is` и `as`, для проверки соответствия протоколу и приведению к определенному протоколу. Приведение к протоколу проходит точно так же как и приведение к типу:

- Оператор `is` возвращает значение `true`, если экземпляр соответствует протоколу и возвращает `false`, если нет.
- Опциональная версия оператора понижающего приведения `as?` возвращает опциональное значение типа протокола, и это значение равно `nil`, если оно не соответствует протоколу.
- Принудительная версия оператора понижающего приведения `as` осуществляет принудительное понижающее приведение, и если оно не завершается успешно, то высказывает runtime ошибка.

Объявление протокола который может быть принят только классом	<pre>protocol HasArea { var area: Double { get } } class Circle: HasArea { let pi = 3.1415927 var radius: Double var area: Double { return pi * radius * radius } init(radius: Double) { self.radius = radius } } class Country: HasArea { var area: Double init(area: Double) { self.area = area } } class Animal { var legs: Int init(legs: Int) { self.legs = legs } } let objects: [AnyObject] = [Circle(radius: 2.0), Country(area: 243_610), Animal(legs: 4)] for object in objects { if let objectWithArea = object as? HasArea { print("Площадь равна \(objectWithArea.area)") } else { print("Что-то такое, что не имеет площади") } } // Площадь равна 12.5663708 // Площадь равна 243610.0</pre>
---	---

Опциональные требования протокола

Вы можете определить опциональные требования для протокола. Эти требования не обязательно должны быть реализованы для соответствия протоколу. Опциональные требования должны иметь префиксный модификатор ***optional*** в качестве части определения протокола. Таким образом вы можете писать код, который взаимодействует с кодом на Objective-C. Имеется в виду, что без ***@objc*** код не будет компилироваться, и при этом наличие ***@objc*** позволяет коду Swift взаимодействовать с кодом Objective-C. И протокол, и опциональное требование должны иметь атрибут ***@objc***. Обратите внимание, что протоколы с маркировкой ***@objc*** могут приниматься только классами, но не структурами или перечислениями.

Когда вы используете опциональное требование свойства или метода, то их тип ***автоматически становится опциональным***. Например, тип метода `(Int) -> String` становится `((Int) -> String)?`

Опциональное требование протокола может быть вызвано при помощи опциональной цепочки, чтобы учесть возможность того, что требование не будет реализовано типом, который соответствует протоколу.

Опциональные требования протокола	требования
	<pre> @objc protocol CounterDataSource { @objc optional func increment(forCount count: Int) -> Int @objc optional var fixedIncrement: Int { get } } class Counter { var count = 0 var dataSource: CounterDataSource? func increment() { if let amount = dataSource?.increment?(forCount: count) { count += amount } else if let amount = dataSource?.fixedIncrement { count += amount } } } class ThreeSource: NSObject, CounterDataSource { let fixedIncrement = 3 } var counter = Counter() counter.dataSource = ThreeSource() for _ in 1...4 { counter.increment() print(counter.count) } // 3 // 6 // 9 // 12 class TowardsZeroSource: NSObject, CounterDataSource { func increment(forCount count: Int) -> Int { if count == 0 { return 0 } else if count < 0 { return 1 } else { return -1 } } } counter.count = -4 counter.dataSource = TowardsZeroSource() for _ in 1...5 { counter.increment() print(counter.count) } // -3 // -2 // -1 // 0 // 0 </pre>

Расширение протоколов

Протоколы могут быть расширены для обеспечения метода и реализации свойства соответствующими типами. Это позволяет вам самостоятельно определить поведение по протоколам, а не по индивидуальному соответствию каждого типа или глобальной функции.

Создавая расширение по протоколу, все соответствующие типы автоматически получают эту реализацию метода без каких-либо дополнительных изменений.

Расширения протоколов могут добавлять реализацию к соответствующим типам данных, но не могут расширить протокол или унаследовать от другого протокола. Наследование протокола всегда указывается в самом объявлении протокола.

Расширение протоколов	<pre>extension RandomNumberGenerator { func randomBool() -> Bool { return random() > 0.5 } } let generator = LinearCongruentialGenerator() print("Рандомное число: \(generator.random())") // Выведет "Рандомное число: 0.37464991998171" print("Рандомное логическое значение: \(generator.randomBool())") // Выведет "Рандомное логическое значение: true"</pre>
-----------------------	---

Обеспечение реализации по умолчанию (дефолтной реализации)

Вы можете использовать расширение протокола, чтобы обеспечить реализацию по умолчанию для любого метода или требования свойства этого протокола. Если соответствующий тип предоставляет свою собственную реализацию требуемого метода или свойства, то реализация будет использоваться вместо той, которая предоставляется расширением.

Требования протокола с реализацией по умолчанию, предоставляемой расширениями, отличаются от опциональных требований протокола. Хотя соответствующие типы не должны предоставлять свою собственную реализацию, требования с реализацией по умолчанию могут быть вызваны без опциональных последовательностей.

Обеспечение реализации по умолчанию (дефолтной реализации)	<pre>protocol TextRepresentable { var textualDescription: String { get } } protocol PrettyTextRepresentable: TextRepresentable { var prettyTextualDescription: String { get } } extension PrettyTextRepresentable { var prettyTextualDescription: String { return textualDescription } }</pre>
--	--

Добавление ограничений к расширениям протоколов

Когда вы определяете расширение протокола, вы можете указать ограничения для принимающих типов, которые они должны удовлетворить до того, как будут доступны методы и свойства расширения. Вы записываете эти ограничения сразу после имени протокола, при помощи оговорки where.

Если подписанный тип удовлетворяет требованиям нескольких ограничивающих расширений, которые предоставляют реализации для одного и того же метода или свойства, то Swift будет использовать самое строгое ограничение.

Добавление ограничений к расширениям протоколов	<pre>extension Collection where Element: Equatable { func allEqual() -> Bool { for element in self { if element != self.first { return false } } return true } } let equalNumbers = [100, 100, 100, 100, 100] let differentNumbers = [100, 100, 200, 100, 200] print(equalNumbers.allEqual()) // Prints "true" print(differentNumbers.allEqual()) // Prints "false"</pre>
---	---

24. Универсальные шаблоны

Универсальный код позволяет писать общего назначения функции и типы, которые могут работать с любыми другими типами, с учетом требований, которые определены.

Коллекции Swift Array или Dictionary являются универсальными.

Универсальные функции

Универсальные функции могут работать с любыми типами, для этого они используют **заполнитель имени типа**.

Пример функции	универсальной	<pre>func swapTwoValues<T>(_ a: inout T, _ b: inout T) { let temporaryA = a a = b b = temporaryA }</pre>
Вызов функции	универсальной	<pre>var someString = "hello" var anotherString = "world" swapTwoValues(&someString, &anotherString) // someString равна "world", а anotherString равна "hello"</pre>

Параметры типа

Заполнитель имени типа **T** пример параметра типа. **Параметры типа** определяют и называют тип наполнителя, и пишутся сразу после имени функции, между угловыми скобками (например, <T>).

Именование параметров типа

В большинстве случаев параметры типа **имеют описательные имена**, такие как Key и Value в Dictionary<Key, Value> и Element в Array<Element>, которые помогут читающему код определить взаимосвязь между параметром типа и универсальным типом или функцией, в которых он используется. Тем не менее, когда между ними **нет значимых отношений**, то по традиции именами становятся отдельные буквы, такие как T, U, V.

Всегда давайте параметрам типа имена **"горбатого" верхнего регистра** (например, T и MyTypeParameter), чтобы указать, что они являются заполнителем для типа, а не значением.

Универсальные типы

Универсальные типы это к примеру универсальные классы, структуры и перечисления, которые могут работать с любыми типами, наподобие тому, как работают Array или Dictionary.

Пример универсального типа на примере структуры	<pre>struct Stack<Element> { var items = [Element]() mutating func push(_ item: Element) { items.append(item) } mutating func pop() -> Element { return items.removeLast() } } var stackOfStrings = Stack<String>() stackOfStrings.push("uno") stackOfStrings.push("dos") stackOfStrings.push("tres") stackOfStrings.push("cuatro") // stack содержит 4 строки let fromTheTop = stackOfStrings.pop() // fromTheTop равен "cuatro", а stack содержит 3 строки</pre>
---	--

Расширяем универсальный тип

Когда вы расширяете универсальный тип, вы не обеспечиваете список параметров в качестве определения расширения. Вместо этого, список параметров типа, из **исходного определения типа**, доступен внутри тела расширения, а имена исходных параметров типа используются для ссылки на параметры типа из исходного определения.

Пример расширения универсального типа	<pre>extension Stack { var topItem: Element? { return items.isEmpty ? nil : items[items.count - 1] } } if let topItem = stackOfStrings.topItem { print("The top item on the stack is \(topItem).") } // Выведет "The top item on the stack is tres."</pre>
---------------------------------------	--

Ограничения типа

Ограничение типа это внедрение определенных ограничений типа на типы, которые могут быть использованы вместе с универсальными функциями или универсальными типами. Ограничения типа указывают на то, что параметры типа должны наследовать от определенного класса или соответствовать определенному протоколу или композиции протоколов.

Вы **пишете ограничения типа**, поместив ограничение единственного класса или протокола после имени параметра типа, и разделив их между собой запятыми, обозначая их в качестве части списка параметров.

Синтаксис ограничения типа T, требует чтобы T, было подклассом класса SomeClass. U, требует чтобы U соответствовал протоколу SomeProtocol.	<pre>func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) { // тело функции... }</pre>
Пример ограничения типа	<pre>func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? { for (index, value) in array.enumerated() { if value == valueToFind { return index } } return nil } let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25]) // doubleIndex опциональный Int не имеющий значения, потому что значения 9.3 нет в массиве let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"]) // stringIndex опциональный Int равный 2</pre>

Связанные типы

При определении протокола бывает нужно определить еще один или более **связанных типов** в качестве части определения протокола. Связанный тип дает **плейсхолдер имени типу**, который используется как часть протокола. Фактический тип, который будет использоваться связанным типом не указывается до тех пор, пока не будет принят протокол.

Связанные типы указываются при помощи ключевого слова **associatedtype**.

Можно **расширить** существующий тип для того, чтобы добавить соответствие протоколу. Можно расширить существующий тип с помощью **пустого расширения**.

Можно **добавить ограничение к связанному типу** в протоколе, чтобы требовать, чтобы соответствующие типы удовлетворяли этим ограничениям

пример протокола Container, который объявляет связанный тип Item	<pre>protocol Container { associatedtype Item mutating func append(_ item: Item) var count: Int { get } subscript(i: Int) -> Item { get } }</pre>
универсальный тип Stack, который соответствует протоколу Container тип параметра Element использован в качестве параметра item метода append(_:) и в качестве возвращаемого типа сабскрипта. Таким образом Swift может вывести, что Element подходящий тип для использования его в качестве типа Item для этого конкретного контейнера.	<pre>struct Stack<Element>: Container { // исходная реализация Stack<Element> var items = [Element]() mutating func push(_ item: Element) { items.append(item) } mutating func pop() -> Element { return items.removeLast() } }</pre>

	<pre>// удовлетворение требований протокола Container mutating func append(_ item: Element) { self.push(item) } var count: Int { return items.count } subscript(i: Int) -> Element { return items[i] } }</pre>
Пример пустого расширения	<pre>extension Array: Container {}</pre>
Добавление ограничений в связанный тип	<pre>protocol Container { associatedtype Item: Equatable mutating func append(_ item: Item) var count: Int { get } subscript(i: Int) -> Item { get } }</pre>

Использование протокола в ограничениях связанного типа и Оговорка where

Протокол может выступать *как часть собственных требований*.

Оговорка where позволяет требовать, чтобы связанный тип, соответствовал определенному протоколу, и/или чтобы конкретные параметры типа и связанные типы были одними и теми же.

Вы *пишете оговорку where*, поместив ключевое слово where сразу после списка параметров типа, за которым следует одно или более ограничений для связанных типов, и/или один или более отношений равенства между типами и связанными типами.

Можно использовать оговорку **where в расширениях**.

Можно написать **универсальную оговорку where** во время объявления, которая не будет иметь своих собственных универсальных ограничений по типу, когда вы уже работаете в контексте универсального типа (**контекстуальная оговорка Where**).

Если вы хотите написать этот код без контекстуальной оговорки where, вам **нужно написать расширения**, по одному для каждой оговорки where.

Можно включать универсальную оговорку where **в связанный тип**.

<p>Пример где протокол выступает как часть собственных требований</p> <p>Suffix имеет два ограничения: он должен соответствовать протоколу SuffixableContainer (протокол, который в настоящее время определяется), а его тип Item должен быть таким же, как тип Item контейнера.</p>	<pre>protocol SuffixableContainer: Container { associatedtype Suffix: SuffixableContainer where Suffix.Item == Item func suffix(_ size: Int) -> Suffix }</pre>
<p>расширение типа Stack, которое добавляет соответствие протоколу SuffixableContainer</p>	<pre>extension Stack: SuffixableContainer { func suffix(_ size: Int) -> Stack { var result = Stack() for index in (count-size)..<count { result.append(self[index]) } return result } // Определено, что Suffix является Stack. } var stackOfInts = Stack() stackOfInts.append(10) stackOfInts.append(20) stackOfInts.append(30) let suffix = stackOfInts.suffix(2) // suffix содержит 20 и 30</pre>
<p>Расширения с оговоркой where</p>	<pre>extension Stack where Element: Equatable { func isTop(_ item: Element) -> Bool { guard let topItem = items.last else { return false } return topItem == item } }</pre>

	<pre> } if stackOfStrings.isTop("tres") { print("Top element is tres.") } else { print("Top element is something else.") } } // Выведет "Top element is tres." </pre>
Еще пример расширения с оговоркой where	<pre> extension Container where Item == Double { func average() -> Double { var sum = 0.0 for index in 0..<count "648.9"="" +="self[index]" 1200.0,="" 37.0].average())="" 98.6,="" <="" double(count)="" pre="" print([1260.0,="" return="" sum="" {="" }="" выведет=""> </count></pre>
Контекстуальная оговорка Where Структура Container является универсальной, и оговорка where в примере определяет какого типа ограничения должны быть выполнены, чтобы эти новые методы были доступны контейнеру.	<pre> extension Container { func average() -> Double where Item == Int { var sum = 0.0 for index in 0..<count "648.75"="" "true"="" &&="" >="1" +="Double(self[index])" ->="" 1200,="" 37]="" 98,="" <="" bool="" count="" double(count)="" endswith(_="" equatable="" func="" item="" item)="" item:="" let="" numbers="[1260," pre="" print(numbers.average())="" print(numbers.endswith(37))="" return="" self[count-1]="=" sum="" where="" {="" }="" выведет=""> </count></pre>
Связанные типы с универсальной оговоркой where Универсальная оговорка where в Iterator требует, чтобы итератор должен поддерживать тот же самый тип элементов, что и тип элементов контейнера, не смотря на тип самого итератора.	<pre> protocol Container { associatedtype Item mutating func append(_ item: Item) var count: Int { get } subscript(i: Int) -> Item { get } associatedtype Iterator: IteratorProtocol where Iterator.Element == Item func makeIterator() -> Iterator } protocol ComparableContainer: Container where Item: Comparable { } </pre>

Универсальные сабскрипты

Сабскрипты могут быть универсальными, и они могут включать в себя универсальную оговорку where. Вы можете написать имя-плейсхолдер внутри угловых скобок после ключевого слова subscript, и вы пишете универсальную оговорку where прямо до открывающей фигурной скобки тела сабскрипта.

Пример универсального сабскрипта	<pre> extension Container { subscript<Indices: Sequence>(indices: Indices) -> [Item] where Indices.Iterator.Element == Int { var result = [Item]() for index in indices { result.append(self[index]) } return result } } </pre>
----------------------------------	--

25. Автоматический подсчет ссылок (ARC)

Swift использует **automatic reference counting (ARC)** (автоматический подсчет ссылок) для отслеживания и управления памятью вашего приложения. ARC автоматически освобождает память, которая использовалась экземплярами класса, когда эти экземпляры больше нам не нужны.

ARC применима **только для экземпляров класса**. Структуры и перечисления являются типами значений, а не ссылочными типами, и они не хранятся и не передают свои значения по ссылке.

Работа ARC

Каждый раз, **когда вы создаете экземпляр класса**, ARC выделяет фрагмент памяти для хранения информации этого экземпляра. Этот фрагмент памяти содержит информацию о типе экземпляра, о его значении и любых свойствах хранения, связанных с ним.

Дополнительно, **когда экземпляр больше не нужен**, ARC освобождает память, использованную под этот экземпляр, и направляет эту память туда, где она нужна.

ARC ведет учет количества свойств, констант, переменных, которые ссылаются на каждый экземпляр класса. **ARC не освободит экземпляр**, если есть хотя бы одна активная ссылка (**strong reference** (сильная ссылка)).

Пример того, как работает ARC	<pre>class Person { let name: String init(name: String) { self.name = name print("\(name) инициализируется") } deinit { print("\(name) деинициализируется") } } var reference1: Person? var reference2: Person? var reference3: Person? reference1 = Person(name: "John Appleseed") // Выведет "John Appleseed инициализируется" reference2 = reference1 reference3 = reference1 // Присвоение другим переменным экземпляра reference1, добавит // две сильные ссылки к экземпляру reference1 reference1 = nil reference2 = nil reference3 = nil // Выведет "John Appleseed деинициализируется"</pre>
--------------------------------------	--

Циклы сильных ссылок между экземплярами классов

Существует возможность написать код, в котором экземпляр класса **никогда** не будет иметь нулевое число сильных ссылок. Это может случиться, если экземпляры классов имеют сильные связи друг с другом, что не позволяет им освободиться. Это известно как **цикл сильных ссылок**.

Пример цикла сильных ссылок	<pre>class Person { let name: String init(name: String) { self.name = name } var apartment: Apartment? deinit { print("\(name) освобождается") } } class Apartment { let unit: String init(unit: String) { self.unit = unit } var tenant: Person? deinit { print("Апартаменты \(unit) освобождаются") } } var john: Person? var unit4A: Apartment? // Рисунок 1 john = Person(name: "John Appleseed") unit4A = Apartment(unit: "4A") john!.apartment = unit4A unit4A!.tenant = john // Рисунок 2 john = nil</pre>
------------------------------------	---

```
unit4A = nil
// Рисунок 3
```

Рисунок 1



Рисунок 2

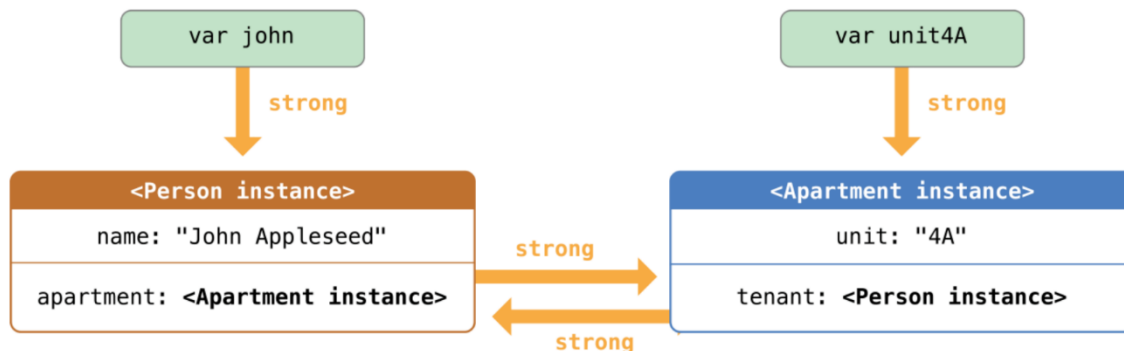
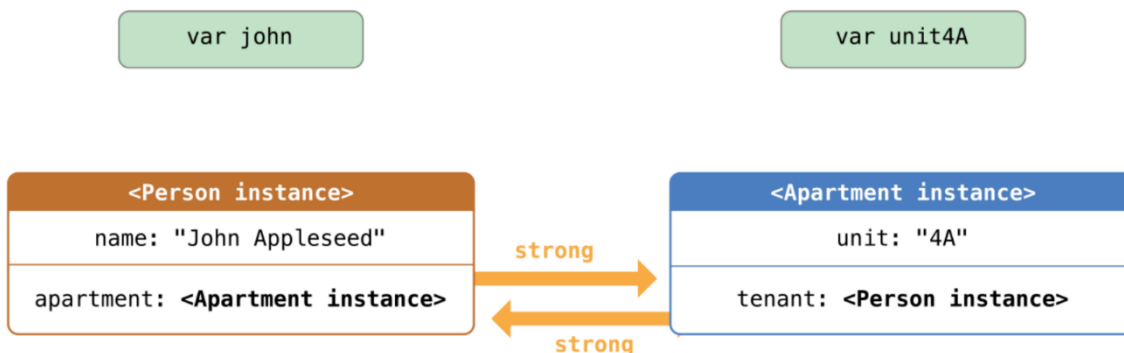


Рисунок 3



Замена циклов сильных ссылок между экземплярами классов

Swift предлагает два способа переопределить ссылку, чтобы она была не сильной, а **слабой** или **бесхозной**.

Слабые и бесхозные ссылки позволяют одному экземпляру в цикле ссылок ссылаться на другой экземпляр без сильного прикрепления. Экземпляры могут ссылаться друг на друга без создания цикла сильных связей.

Используйте слабую ссылку, если другой экземпляр имеет более короткое время жизни, то есть когда другой экземпляр может быть освобожден первым.

Используйте бесхозные ссылки, если другой экземпляр имеет одинаковое время жизни или более длительный срок службы.

Слабые (weak) ссылки

Слабые ссылки не удерживаются за экземпляр, на который они указывают, так что ARC не берет их во внимание, когда считает ссылки экземпляра. Вы указываете слабую ссылку ключевым словом **weak** перед объявлением имени свойства или переменной.

ARC автоматически присваивает **слабой ссылке nil**, когда экземпляр, на который она указывает, освобождается. И поскольку слабые ссылки должны позволять изменять их значение до nil во время выполнения, они всегда **объявляются как переменные**, а не как константы опционального типа.

Вы можете **проверить существование значения в слабой ссылке** точно так же как и с любыми другими опциональными значениями, и вы никогда не будете иметь ссылку с недопустимым значением, например, указывающую на несуществующий экземпляр.

Когда ARC устанавливает слабую ссылку на nil, **наблюдатели свойств не вызываются**.

Там, где используются **сборщики "мусора"**, слабые указатели иногда используются для реализации простого механизма кеширования, потому что объекты без сильных связей сразу отпускаются, как только у памяти появляется необходимость избавиться от "мусора". Однако со включенной ARC значения удаляются только тогда, когда уходит последняя сильная связь на них, делая слабые связи не подходящими для текущей задачи.

Пример слабый (weak) ссылки

```
class Person {
  let name: String
  init(name: String) { self.name = name }
  var apartment: Apartment?
  deinit { print("\(name) деинициализируется") }
}
class Apartment {
  let unit: String
  init(unit: String) { self.unit = unit }
  weak var tenant: Person?
  deinit { print("Apartment \(unit) деинициализируется") }
}
var john: Person?
var unit4A: Apartment?
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
// Рисунок 4
john = nil
// Выведет "John Appleseed деинициализируется"
// Рисунок 5
unit4A = nil
// выводит "Апартаменты 4A деинициализируется"
// Рисунок 6
```

Рисунок 4

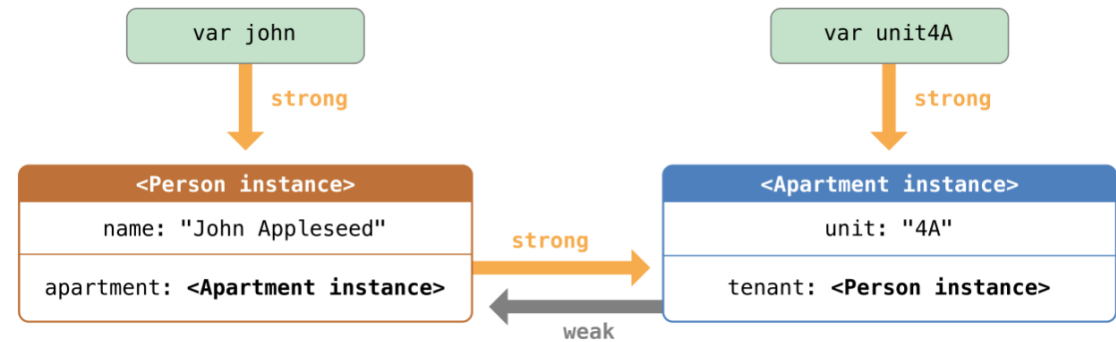
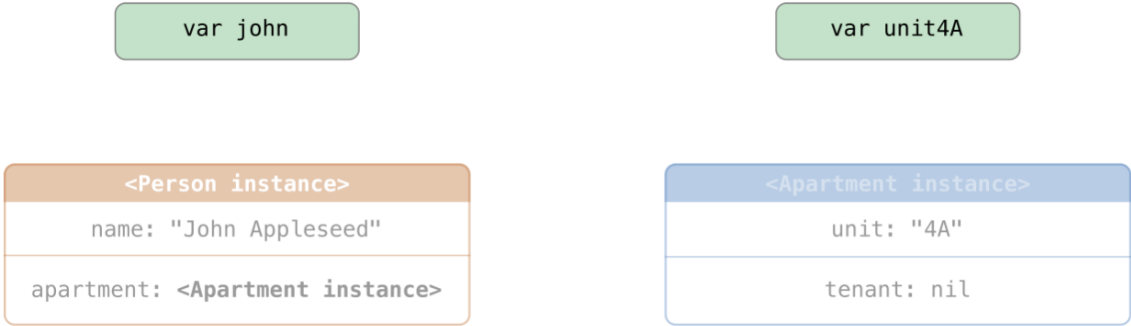


Рисунок 5



Рисунок 6



Бесхозные ссылки

Как и слабые ссылки, бесхозные ссылки *также не имеют сильной связи* с экземпляром, на который они указывают.

В отличие от слабых ссылок, бесхозные ссылки *всегда имеют значение*. Из-за этого бесхозные ссылки *имеют неопциональный тип*. Вы указываете на то, что ссылка бесхозная ключевым словом *unowned*, поставленным перед объявлением свойства или переменной.

Так как бесхозная ссылка не является опциональной, то вам не нужно и разворачивать ее каждый раз, когда вы собираетесь ее использовать. Вы *можете обратиться к бесхозной ссылке напрямую*.

ARC не может установить значение ссылки на nil, когда экземпляр, на который она ссылается, освобожден.

Используйте бесхозные ссылки только в том случае, если вы абсолютно уверены в том, что ссылка всегда будет указывать на экземпляр. Если вы попытаетесь получить доступ к бесхозной ссылке после того, как экземпляр, на который она ссылается освобожден, то выскочит *runtime ошибка*.

Пример слабый (weak) ссылки

В конце примера из-за того, что более сильных ссылок, ссылающихся на экземпляр Customer нет, то этот экземпляр освобождается. После того, как это происходит, у нас не остается больше сильных ссылок, указывающих на экземпляр CreditCard, так что он тоже освобождается

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) деинициализируется") }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Карта #\(number) деинициализируется") }
}

var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234567890123456, customer: john!)
// Рисунок 7
john = nil
// Рисунок 8
// Выведет "John Appleseed деинициализируется"
// Выведет "Карта #1234567890123456 деинициализируется"
```

Рисунок 7

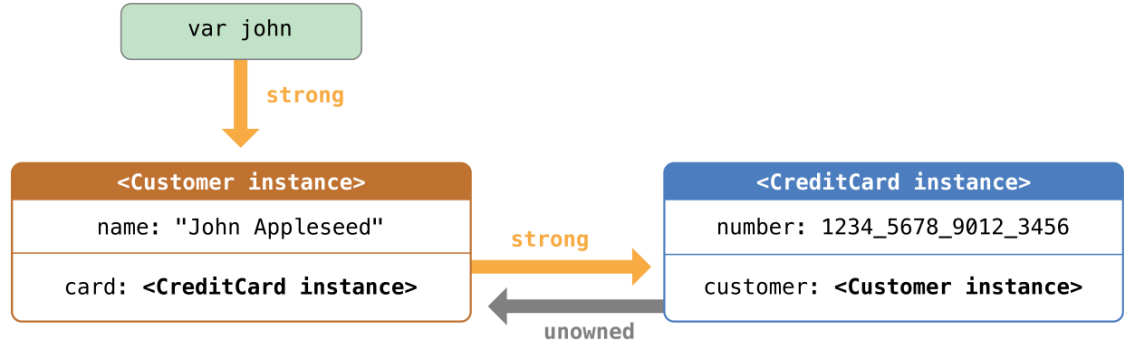
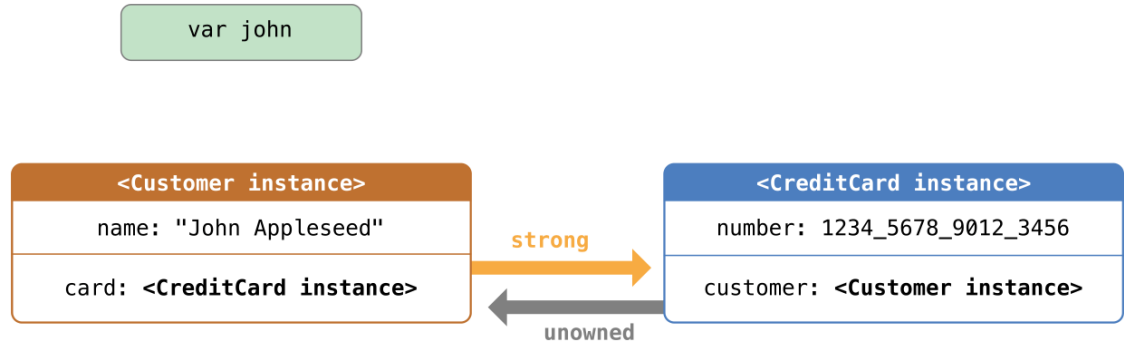


Рисунок 8



Примеры выше **показывают как использовать safe unowned связи**. Swift так же предоставляет unsafe unowned связи для случаев, где вам нужно отключить проверку безопасности во время исполнения, например в случае, когда вы хотите увеличить производительность. Как и со всеми небезопасными операциями, всю ответственность за проверку кода на безопасность вы берете на себя.

Чтобы показать, что вы будете использовать **unsafe unowned** связь, вам нужно написать **unowned(unsafe)**. Если вы попытаетесь получить доступ к unsafe unowned ссылке после того, как экземпляр был освобожден, ваша программа попытается получить доступ к памяти, где этот объект хранился ранее, что само по себе является небезопасной операцией.

Бесхозные опциональные ссылки

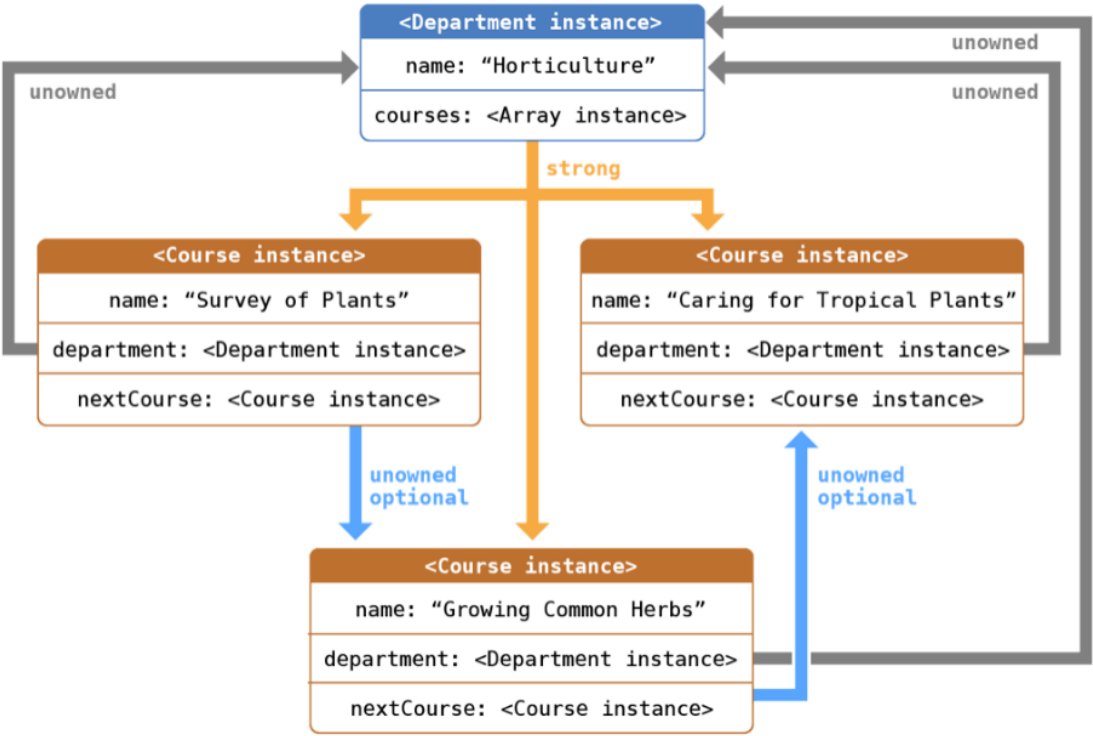
С точки зрения модели ARC опциональная бесхозная ссылка и слабая ссылка могут быть использованы в одних и тех же контекстах. **Разница лишь в том**, что когда вы используете опциональную бесхозную ссылку, вы ответственны за то, чтобы она ссылалась на валидный объект или была бы установлена на nil.

Бесхозная опциональная ссылка не имеет сильной связи с экземпляром класса, который она удерживает, так что она не удерживает ARC от освобождения экземпляра класса. Она ведет себя точно так же как бесхозная ссылка в ARC за исключением того, что бесхозная опциональная ссылка может быть nil.

Лежащий в основе опционального значения тип - Optional, который является по своей сути просто перечислением в стандартной библиотеке Swift. Однако, **опционалы являются исключением из правил**, так как типы значений не могут быть маркированы как unowned. **Опционал, который является оберткой** для класса не использует подсчет ссылок, так что вам не нужно поддерживать сильную ссылку на опционал.

<div>Пример использования бесхозных опциональных ссылок</div> <div>Схема ссылок на рисунке 9</div>	<pre>class Department { var name: String var courses: [Course] init(name: String) { self.name = name self.courses = [] } } class Course { var name: String unowned var department: Department unowned var nextCourse: Course? init(name: String, in department: Department) { self.name = name self.department = department self.nextCourse = nil } } let department = Department(name: "Horticulture") let intro = Course(name: "Survey of Plants", in: department) let intermediate = Course(name: "Growing Common Herbs", in: department) let advanced = Course(name: "Caring for Tropical Plants", in: department) intro.nextCourse = intermediate intermediate.nextCourse = advanced department.courses = [intro, intermediate, advanced]</pre>
--	--

Рисунок 9



Бесхозные ссылки и неявно извлеченные опциональные свойства

Пример с Person, Apartment показывает ситуацию, где два свойства, оба из которых могут иметь значение nil, имеют потенциальную возможность образования цикла сильных связей. **Этот случай лучше всего решается с помощью слабой связи.**

Пример с Customer, CreditCard демонстрирует ситуацию, где одному свойству разрешено иметь значение nil, другому - нет. Однако здесь так же существует потенциальная возможность образования цикла сильных ссылок. **Такой случай лучше всего разрешается с помощью бесхозных ссылок.**

Есть и третий вариант, в котором оба свойства должны всегда иметь значение, и ни одному из них нельзя иметь nil, после завершения инициализации. **В этом случае лучше всего скомбинировать бесхозное свойство одного класса с неявно извлеченным опциональным свойством другого класса.** Это позволяет получить доступ к обоим свойствам напрямую (без опционального извлечения) после завершения инициализации, так же позволяя избегать взаимных сильных ссылок.

<p>Комбинирование бесхозного свойства одного класса с неявно извлеченным опциональным свойством другого класса</p> <p>В примере экземпляры Country и City создаются единственным выражением, без создания цикла сильных ссылок друг на друга.</p> <p>Так как свойство capitalCity (неявно извлеченное опциональное свойство (City!)) при инициализации имеет значение по умолчанию nil</p>	<pre>class Country { let name: String var capitalCity: City! init(name: String, capitalName: String) { self.name = name self.capitalCity = City(name: capitalName, country: self) } } class City { let name: String unowned let country: Country init(name: String, country: Country) { self.name = name self.country = country } } var country = Country(name: "Россия", capitalName: "Москва") print("Столицей страны \(country.name) является \(country.capitalCity.name)") // Выведет "Столицей страны Россия является Москва"</pre>
---	--

Циклы сильных ссылок в замыканиях

Сильные ссылки так же могут образовываться, когда вы **присваиваете замыкание свойству экземпляра класса**, и тело замыкания захватывает экземпляр. Этот захват может случиться из-за того, что тело замыкания получает доступ к свойству экземпляра

Этот цикл возникает из-за того, что замыкания, как и классы, **являются ссылочными типами**. Когда вы присваиваете замыкание свойству, вы присваиваете ссылку на это замыкание.

В отличие от предыдущих примеров здесь не два экземпляра классов, а замыкание и один экземпляр класса, которые поддерживают существование друг друга. Swift предлагает элегантное решение этой проблемы, которое известно как **список захвата замыкания** (closure capture list).

Пример цикла сильных ссылок, когда мы используем замыкание, которое ссылается на self

Свойство asHTML объявлено как ленивое свойство, потому что оно нам нужно только тогда, когда элемент должен быть отображен в виде строкового значения для какого-либо HTML элемента выходного значения. Факт того, что свойство asHTML является ленивым, означает, что вы можете сослаться на self внутри дефолтного замыкания, потому что обращение к ленивому свойству невозможно до тех пор, пока инициализация полностью не закончится и не будет известно, что self уже существует.

Переменная paragraph определена как опциональный HTMLElement, так что он может быть и nil для демонстрации цикла сильных ссылок.

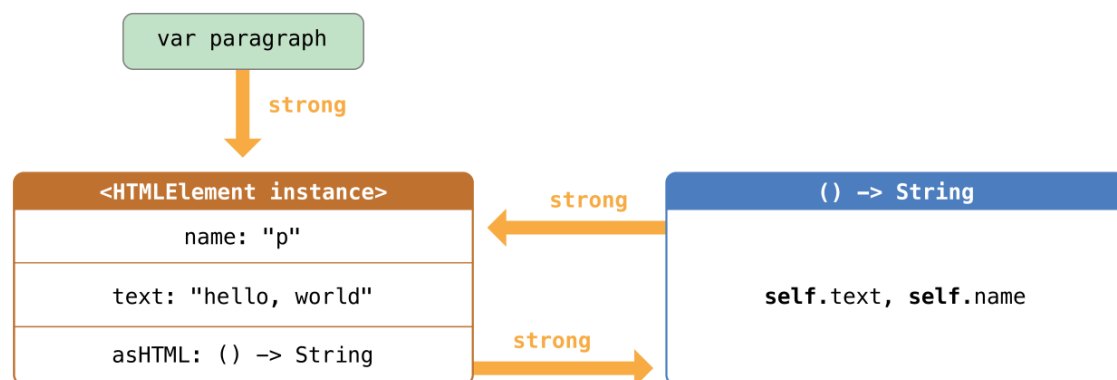
Если вы установите значение paragraph на nil, чем разрушите сильную ссылку на экземпляр HTMLElement, то ни экземпляр HTMLElement, ни его замыкание не будут освобождены из-за цикла сильных ссылок

Схема ссылок на рисунке 10

```
class HTMLElement {
    let name: String
    let text: String?
    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(\(self.name))>"
        } else {
            return "<\(self.name) />"
        }
    }
    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
    deinit {
        print("\(name) деинициализируется")
    }
}

let heading = HTMLElement(name: "h1")
let defaultText = "some default text"
heading.asHTML = {
    return "<\(heading.name)>\(heading.text ?? defaultText)</\(\(heading.name))>"
}
print(heading.asHTML())
// Выведет "<h1>some default text</h1>"
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// Выведет "<p>hello, world</p>"
paragraph = nil
```

Рисунок 10



Свойство asHTML экземпляра держит сильную ссылку на его замыкание. Однако из-за того, что **замыкание** ссылается на self внутри своего тела (`self.name`, `self.text`), оно захватывает self, что означает, что замыкание держит сильную ссылку обратно на экземпляр HTMLElement. **Даже несмотря на то**, что замыкание ссылается на self несколько раз, оно захватывает лишь одну сильную ссылку на экземпляр HTMLElement.

Замена циклов сильных ссылок в замыканиях

Заменить цикл сильных ссылок между замыканием и экземпляром класса можно путем определения **списка захвата** в качестве части определения замыкания. **Список захвата определяет правила**, которые нужно использовать при захвате одного или более ссылочного типа в теле замыкания.

Swift требует от вас написания `self.someProperty` или `self.someMethod()` (вместо `someProperty`, `someMethod()`), каждый раз, когда вы обращаетесь к члену свойства self внутри замыкания. Это помогает вам не забыть, что возможен случай случайного захвата self.

Каждый элемент в списке захвата является парой **ключевого слова weak или unowned и ссылки на экземпляр класса** (например, `self`) или переменную, инициализированную с помощью какого-либо значения

(например, `delegate = self.delegate!`). Эти пары вписываются в квадратные скобки и разделяются между собой запятыми.

Размещается список захвата перед списком параметров замыкания и его возвращаемым типом. Если у замыкания нет списка параметров или возвращаемого типа, так как они могут быть выведены из контекста, то разместите список захвата в самом начале замыкания, перед словом `in`

Примеры захвата	определения списка	<pre>lazy var someClosure: (Int, String) -> String = { [unowned self, weak delegate = self.delegate!] (index: Int, stringToProcess: String) -> String in // тело замыкания }</pre>
		<pre>lazy var someClosure: () -> String = { [unowned self, weak delegate = self.delegate!] in // тело замыкания }</pre>

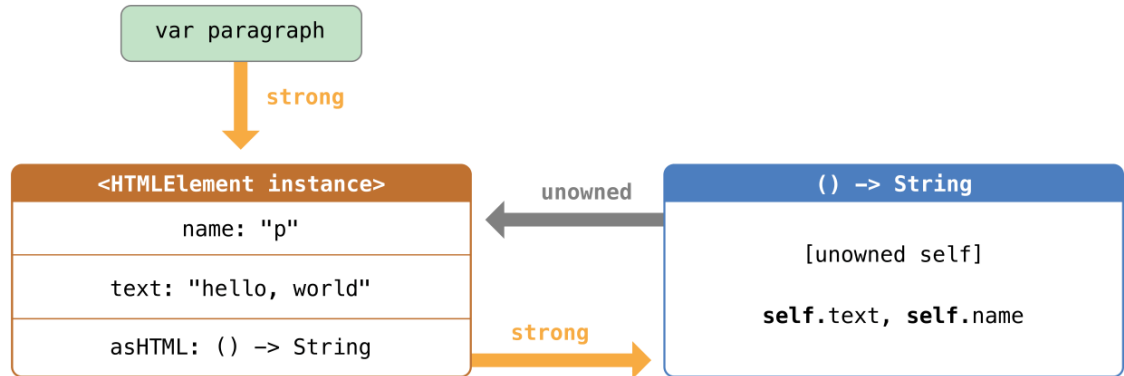
Определите список захвата в замыкании **как бесхозную ссылку в том случае**, когда замыкание и экземпляр, который оно захватывает, всегда будут ссылаться друг на друга, тогда они всегда будут освобождаться в одно и то же время.

Наоборот, определите список захвата **в качестве слабой ссылки**, когда захваченная ссылка может стать `nil` в какой-либо момент в будущем. Слабые ссылки всегда опционального типа и автоматически становятся `nil`, когда экземпляр, на который они ссылаются, освобождается. **Это позволяет** вам проверять их существование внутри тела замыкания.

Если захваченная ссылка никогда не будет nil, то она должна быть всегда захвачена как `unowned` ссылка, а не `weak` ссылка.

Пример использования цикла захвата	<pre>class HTMLElement { let name: String let text: String? lazy var asHTML: () -> String = { [unowned self] in if let text = self.text { return "<\(self.name)>\(text)</\(\self.name)>" } else { return "<\(self.name) />" } } init(name: String, text: String? = nil) { self.name = name self.text = text } deinit { print("\(name) освобождается") } } var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world") print(paragraph!.asHTML()) // Выведет "<p>hello, world</p>" paragraph = nil // Выведет "p освобождается"</pre>
---	---

Рисунок 11



26. Безопасность хранения

Доступ к памяти происходит в вашем коде, когда вы, например, устанавливаете значение переменной или передаете аргумент функции.

Конфликт доступа к памяти может возникнуть, когда разные части вашего кода пытаются одновременно получить доступ к одному и тому же фрагменту памяти.

В частности, конфликт возникает, если у вас есть два доступа, отвечающие полностью всем следующим условиям:

- По крайней мере, один из них является доступом на запись или неатомарным доступом.
- Они получают доступ к одному и тому же фрагменту в памяти.
- Их длительность перекрывается друг другом.

Длительность доступа к памяти - мгновенная или долгосрочная.

Доступ считается мгновенным (моментальным), если невозможно запустить другой код после того, пока не завершится уже запущенный код с доступом к памяти. По своей природе два моментальных (мгновенных) доступа не могут произойти одновременно. В большинстве случаев доступ к памяти происходит мгновенно.

Долгосрочный доступ может перекрываться с другими долгосрочными доступами и мгновенными доступами. **Перекрывающиеся обращения появляются**, прежде всего, в коде, который использует сквозные параметры в функциях и методах или методы структуры с модификатором mutating

Пример доступа к памяти	// Доступ к памяти с правами записи, где хранится данная переменная var one = 1 // Доступ к памяти с правами чтения, где хранится данная переменная print("We're number \\$(one)!")
Пример мгновенного доступа	func oneMore(than number: Int) -> Int { return number + 1 } var myNumber = 1 myNumber = oneMore(than: myNumber) print(myNumber) // Выведет "2"

Конфликт доступа к сквозным параметрам

Функция имеет долгосрочный доступ для записи ко всем ее сквозным параметрам. Доступ записи для сквозного параметра **начинается** после того, как все несквозные параметры были оценены и **длится** всю продолжительность вызова этой функции. Если имеется **несколько сквозных параметров**, то доступы на запись начинаются в том же порядке, в каком были объявлены параметры.

Одним из **последствий этого долгосрочного доступа** для записи является то, что вы не можете получить доступ к исходной переменной, которая была передана как сквозная переменная, даже если правила определения области видимости и контроля доступа это позволяют - любой доступ к оригиналу **создаст конфликт**.

Пример конфликта доступа к сквозным параметрам (получение доступа к исходной переменной, которая до этого была передана как сквозная)	<pre>var stepSize = 1 func increment(_ number: inout Int) { number += stepSize } increment(&stepSize) // Ошибка: conflicting accesses to stepSize</pre> <div><pre>func increment(_ number: inout Int){ number += <u>stepSize</u> }</pre></div>
Решение конфликта доступа через явную копию stepSize	<pre>var copyOfStepSize = stepSize increment(&copyOfStepSize) // Обновим оригинал stepSize = copyOfStepSize // stepSize равен 2</pre>

Другим следствием долгосрочного доступа для записи к сквозным параметрам является то, что передача одной переменной в качестве аргумента для нескольких сквозных параметров одной и той же функции **вызывает конфликт**.

Пример конфликта доступа к сквозным параметрам
(передача одного аргумента для нескольких сквозных параметров)

```
func balance(_ x: inout Int, _ y: inout Int) {  
    let sum = x + y  
    x = sum / 2  
    y = sum - x  
}  
  
var playerOneScore = 42  
var playerTwoScore = 30  
balance(&playerOneScore, &playerTwoScore) // OK  
balance(&playerOneScore, &playerOneScore)  
// Ошибка: Conflicting accesses to playerOneScore
```

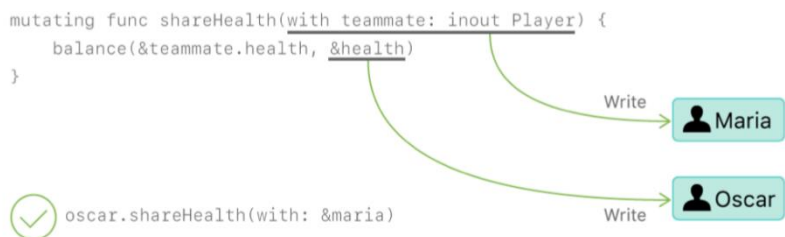
Так как операторы это функции, то они также могут иметь **долгосрочный доступ** к своим сквозным параметрам. Например, если `balance(_:_)` это операторная функция с именем `<^>`, то запись `playerOneScore <^> playerOneScore` приведет к такому же конфликту, что и `balance(&playerOneScore, &playerOneScore)`

Конфликт доступа к self в методах

Mutating метод в структуре имеет доступ для записи к self на время вызова метода.

Пример доступа к self в методах без конфликта

```
struct Player {  
    var name: String  
    var health: Int  
    var energy: Int  
    static let maxHealth = 10  
    mutating func restoreHealth() {  
        health = Player.maxHealth  
    }  
}  
  
extension Player {  
    mutating func shareHealth(with teammate: inout Player) {  
        balance(&teammate.health, &health)  
    }  
}  
  
var oscar = Player(name: "Oscar", health: 10, energy: 10)  
var maria = Player(name: "Maria", health: 5, energy: 10)  
oscar.shareHealth(with: &maria) // OK
```



Пример доступа к self в методах с конфликтом

```
oscar.shareHealth(with: &oscar)  
// Ошибка: conflicting accesses to oscar  
  
mutating func shareHealth(with teammate: inout Player) {  
    balance(&teammate.health, &health)  
}
```



Конфликт доступа к свойствам

Такие типы, как **структуры, кортежи и перечисления**, состоят из отдельных составляющих их значений, таких как свойства структуры или элементы кортежа. Поскольку они являются типами значений, **изменение любой части значения меняет все значение**, то есть доступ для чтения или записи к одному из свойств требует доступа для чтения или записи ко всему значению.

Если использовать локальную переменную вместо глобальной переменной, **то компилятор может доказать**, что перекрывающий доступ к сохраненным свойствам структуры безопасен

Пример конфликта доступа к свойствам (запись к элементам кортежа)	<pre>var playerInformation = (health: 10, energy: 20) balance(&playerInformation.health, &playerInformation.energy) // Ошибка: conflicting access to properties of playerInformation</pre>
Пример конфликта доступа к свойствам (запись свойств структуры, которая хранится в глобальной переменной)	<pre>var holly = Player(name: "Holly", health: 10, energy: 10) balance(&holly.health, &holly.energy) // Ошибка</pre>
Пример где нет конфликта доступа к свойствам, так как она локальная а не глобальная	<pre>func someFunction() { var oscar = Player(name: "Oscar", health: 10, energy: 10) balance(&oscar.health, &oscar.energy) // ОК }</pre>

Безопасность хранения - это желаемая гарантия, но **эксклюзивный доступ** является более строгим требованием, чем безопасность хранения, что означает, что код может сохранять безопасность хранения, даже несмотря на то, что он нарушает исключительный доступ к памяти.

Swift позволяет использовать этот безопасный для памяти код, если компилятор может доказать, что неисключительный доступ к памяти по-прежнему безопасен. В частности, он может доказать, что перекрывающий доступ к свойствам структуры безопасен, если применяются **следующие условия**:

- Вы получаете доступ только к сохраненным свойствам экземпляра, а не к вычисленным свойствам или свойствам класса.
- Структура - это значение локальной переменной, а не глобальной переменной.
- Структура либо не захватывается никакими замыканиями, либо захватывается только несбегающими замыканиями.

Если компилятор не может доказать, что доступ безопасен, он не разрешает доступ.

27. Контроль доступа

Контроль доступа ограничивает доступ к частям вашего кода из кода других исходных файлов и модулей. Эта особенность позволяет вам прятать детали реализации вашего кода и указывать на предпочтительный интерфейс, через который можно получить доступ к вашему коду.

Вы можете присвоить определенные уровни доступа как к индивидуальным типам (классы, структуры и перечисления), так и к свойствам, методам, инициализаторам и сабскриптам, принадлежащим этим типам.

Протоколы могут быть ограничены в определенном контексте, так же как могут быть ограничены глобальные переменные или функции.

Модули и исходные файлы

Модель контроля доступа Swift основывается **на концепции** модулей и исходных файлов.

Модуль представляет из себя единый блок распределения кода - фреймворк или приложение, которое построено и поставляется в качестве единого блока и которое может быть импортировано другим модулем с ключевым словом **import**.

Каждый таргет сборки (например, бандл приложения или фреймворк) в Xcode обрабатывается как отдельный модуль. Если вы объедините вместе аспекты кода вашего приложения в качестве отдельного фреймворка, то их возможно будет инкапсулировать и использовать заново во множестве других приложений. Таким образом, все, что вы определите в рамках этого фреймворка будет считаться частью отдельного модуля, когда это будет импортировано и использовано внутри приложения, или когда это будет использовано внутри другого фреймворка.

Исходный файл - исходный код файла в пределах одного модуля (в сущности это и есть один файл вашего приложения или фреймворка). Хотя в большинстве случаев определение типов происходит в отдельных исходных файлах, но фактически исходный файл может содержать определения множества различных типов, функций и т.д.

Уровни доступа

Swift предлагает **пять различных уровней доступа** для объектов вашего кода:

- **Открытый доступ** и **публичный доступ** (**open access** и **public access**). Этот уровень доступа позволяет использовать объекты внутри любого исходного файла из определяющего их модуля и так же в любом исходном файле из другого модуля, который импортирует определяющий модуль. Вы обычно используете открытый и публичный доступы, когда указываете общий интерфейс фреймворка. Отличия между этими двумя уровнями доступа будут описаны ниже.
- **Внутренний** (**internal access**). Этот уровень доступа позволяет использовать объекты внутри любого исходного файла из их определяющего модуля, но не исходного файла не из этого модуля. Вы обычно указываете внутренний доступ, когда определяете внутреннюю структуру приложения или фреймворка.
- **Файл-частный** (**file private**). Этот уровень доступа позволяет использовать объект в пределах его исходного файла. Используйте файл-частный уровень доступа для того, чтобы спрятать детали реализации определенной части функциональности, когда эти части функциональности будут использоваться внутри другого файла.
- **Частный** (**private**). Этот уровень доступа позволяет использовать сущность только в пределах области ее реализации. Используйте частный доступ для того, чтобы спрятать детали реализации конкретной части функциональности, когда они используются только внутри области объявления.

Открытый доступ - самый высокий уровень доступа (наименее строгий), и **частный уровень** доступа является самым низким уровнем доступа (самый строгий).

Открытый доступ применяется только к классам и членам класса и **отличается от public** доступа следующим:

- Классы, с уровнем доступа **public**, могут наследоваться только в том модуле, в котором они определены.
- Члены класса с уровнем доступа **public** или с более строгим уровнем доступа могут быть переопределены подклассами только в том модуле, в котором они определены.
- Открытые классы могут наследоваться внутри модуля, в котором они определены и внутри модуля, который импортирует модуль, в котором они определены.
- Открытые члены класса могут переопределяться подклассами внутри модуля, в котором они были определены или внутри модуля, который импортирует модуль, в котором они были определены.

Обозначая класс через маркер *open*, явно свидетельствует о том, что вы рассмотрели влияние этого класса на код других модулей, использующих его в качестве суперкласса.

Руководящий принцип по выбору уровня доступа

Уровни доступа в Swift следуют общему руководящему принципу: никакой объект не может быть определен в пределах другого объекта, который имеет более низкий (более строгий) уровень доступа.

Например:

- Переменная с уровнем доступа **public** не может быть определена как будто она имеет уровень доступа **private**, потому что этот уровень доступа не может быть использован везде, где доступен **public**.
- Функция не может иметь уровень доступа выше чем у ее параметров или возвращаемого типа, потому что функция не может использоваться там, где ее параметры не доступны.

Дефолтный уровень доступа

Все сущности вашего кода (кроме двух исключений) имеют дефолтный уровень доступа - **внутренний (internal)**, если вы явно не указываете другой уровень. В результате во многих случаях вам не нужно указывать явный уровень доступа в вашем коде.

Уровень доступа для простых однозадачных приложений

Когда вы пишете **простое однозадачное приложение**, то код вашего приложения обычно самодостаточен и не требует доступа к нему из внешних источников. По умолчанию уровень доступа стоит **внутренний**. Если вам все таки нужно, то вы можете некоторые части вашего кода обозначить как **fileprivate** или **private**, для того чтобы спрятать детали реализации от другого кода этого же модуля.

Уровень доступа для фреймворка

Когда вы разрабатываете **фреймворк**, обозначьте внешний интерфейс фреймворка как **open, public**, так чтобы его можно было посмотреть и получить к нему доступ из других модулей, так например, чтобы приложение могло импортировать его.

Внешний интерфейс - интерфейс прикладного программирования (API) для фреймворка.

Уровни доступа для модуля поэлементного тестирования (unit test target)

Когда вы пишете приложение с **модулем поэлементного тестирования**, то код вашего приложения должен быть доступным для модуля, чтобы он мог его проверить. По умолчанию только сущности с маркировкой **public могут быть доступны** для других модулей, однако этот модуль может получить доступ ко всем внутренним сущностям, если вы поставили входную маркировку объявления модуля продукта как **@testable** и компилируете со включенным режимом тестирования.

Синтаксис контроля доступа	<pre>public class SomePublicClass {} internal class SomeInternalClass {} fileprivate class SomeFilePrivateClass {} private class SomePrivateClass {} public var somePublicVariable = 0 internal let someInternalConstant = 0 fileprivate func someFilePrivateFunction() {} private func somePrivateFunction() {} class SomeInternalClass {} // неявно internal let someInternalConstant = 0 // неявно internal</pre>
----------------------------	---

Пользовательские типы

Если вы хотите **указать явно** уровень доступа для пользовательского типа, то делайте это на этапе определения типа. Новый тип может быть **использован там**, где позволяет его уровень доступа.

Контроль уровня доступа типа так же **влияет на уровень доступа для этих членов по умолчанию** (его свойств, методов, инициализаторов и сабскриптов). Если вы определяете уровень доступа типа как **fileprivate** или **private**, то дефолтный уровень доступа его членов так же будет **fileprivate** или **private**.

Если вы определяете уровень доступа типа как **fileprivate** или **private**, то дефолтный уровень доступа его членов так же будет **fileprivate** или **private**.

Если вы определите уровень доступа как **internal** или **public** (или будете использовать дефолтный уровень доступа, без явного указания **internal**), то уровень доступа членов типа по умолчанию будет **internal**. Если вы хотите чтобы члены типа имели уровень доступа **public**, то **вы должны явно указать его**. Такое **требование гарантирует**, что внешняя часть API - эта та часть, которую вы выбираете сами и исключает тот случай, когда вы можете по ошибке забыть указать **internal** для внутреннего кода.

Примеры уровней доступа	<pre>public class SomePublicClass { // явный public класс public var somePublicProperty = 0 // явный public член класса var someInternalProperty = 0 // неявный internal член класса fileprivate func someFilePrivateMethod() {} // явный file-private член класса private func somePrivateMethod() {} // явный private член класса } class SomeInternalClass { // неявный internal класс var someInternalProperty = 0 // неявный internal член класса fileprivate func someFilePrivateMethod() {} // явный file-private член класса private func somePrivateMethod() {} // явный private член класса } fileprivate class SomeFilePrivateClass { // явный file-private класс func someFilePrivateMethod() {} // неявный file-private член класса private func somePrivateMethod() {} // явный private член класса } private class SomePrivateClass { // явный private класс func somePrivateMethod() {} // неявный private член класса }</pre>
-------------------------------	---

Кортежи типов

Уровень доступа для кортежей типов имеет самый строгий уровень доступа типа из всех используемых типов в кортеже. **Например**, если вы скомпонуете кортеж из двух разных типов, один из которых будет иметь уровень доступа как internal, другой как private, то кортеж будет иметь уровень доступа как private.

Кортежи типов не имеют отдельного определения в отличии от классов, структур, перечислений или функций. **Уровень доступа кортежей типов вычисляется автоматически**, когда используется кортеж, и не может быть указан явно.

Типы функций

Уровень доступа для типов функции вычисляется как самый строгий уровень доступа из типов параметров функции и типа возвращаемого значения. **Вы должны указывать уровень** доступа явно как часть определения функции, если вычисляемый уровень доступа функции не соответствует контекстному по умолчанию.

Функция, которая описана справа не будет компилироваться:	<pre>func someFunction() -> (SomeInternalClass, SomePrivateClass) { // реализация функции... }</pre>
Возвращаемый тип функцией является кортежем, который составлен из двух пользовательских классов, один из этих классов был определен как internal, другой - как private. Таким образом, общий уровень доступа кортежа будет вычислен как private	
Из-за того, что уровень доступа функции private, то вы должны установить общий уровень доступа как private во время определения функции	<pre>private func someFunction() -> (SomeInternalClass, SomePrivateClass) { // реализация функции... }</pre>

Типы перечислений

Каждый кейс в перечислении автоматически получает тот же уровень доступа, что и само перечисление. Вы не можете указать другой уровень доступа для какого-то определенного кейса перечисления.

Пример уровня доступа у перечисления	<pre>public enum CompassPoint { case north case south case east case west }</pre>
--------------------------------------	---

Исходные значения и связанные значения

Типы, используемые для любых начальных значений или связанных значений в перечислении, **должны иметь** как минимум такой же высокий уровень доступа как и перечисление. **Вы не можете использовать** тип private для типа исходного значения перечисления, которое имеет internal уровень доступа.

Вложенные типы

Вложенные типы, определенные внутри типа с уровнем доступа **private**, автоматически получают уровень доступа **private**.

Вложенные типы внутри public типов или internal типов, автоматически получают уровень доступа как **internal**.

Если вы хотите, чтобы вложенный тип внутри public типа имел уровень доступа как public, то вам нужно явно **указать этот тип самостоятельно**.

Уровень доступа класса и подкласса

Подкласс не может иметь более высокого уровня доступа, чем его суперкласс. **Например**, вы не можете написать подклассу public, если его суперкласс имеет internal доступ.

В дополнение **вы можете переопределить любой член класса** (метод, свойство, инициализатор или сабскрипт), который будет виден в определенном контексте доступа. Переопределение может сделать член унаследованного класса **более доступным**, чем его версия суперкласса.

Пример переопределения, где член унаследованного класса более доступный	<pre>public class A { fileprivate func someMethod() {} } internal class B: A { override internal func someMethod() {} }</pre>
Член подкласса может вызвать член суперкласса, который имеет более низкий уровень доступа, чем член подкласса, до тех пор пока вызов члена суперкласса попадает под допустимый уровень доступа контекста	<pre>public class A { fileprivate func someMethod() {} } internal class B: A { override internal func someMethod() { super.someMethod() } }</pre>

Константы, переменные, свойства и сабскрипт

Константы, переменные, свойства не могут быть более открытыми, чем их тип. Это не правильно писать свойство public для private типа. **Аналогично дело обстоит и с сабскриптом**: сабскрипт не может быть более открытым, чем тип индекса или возвращаемый тип.

Если константа, переменная, свойство или сабскрипт используют тип private, то они должны быть отмечены ключевым словом private:

Пример определения типа у константы, переменных, свойств и сабскриптов	<pre>private var privateInstance = SomePrivateClass()</pre>
--	---

Геттеры и сеттеры

Геттеры и сеттеры для констант, переменных и сабскриптов **автоматически получают** тот же уровень доступа как и константа, переменная, свойство или сабскрипт, которому они принадлежат. Это правило **применяется как к свойствам хранения так и к вычисляемым свойствам**.

Вы можете задать сеттер более низкого уровня доступа чем его соответствующий геттер, для ограничения области read-write этой переменной, свойства или сабскрипта. Вы присваиваете более низкий уровень доступа написав **fileprivate(set)**, **private(set)** или **internal(set)** до вступительного var или subscript.

Можно присвоить **явный уровень доступа и к геттеру, и к сеттеру**.

Пример определения уровня доступа у сеттера Вы и можете обращаться к текущему значению свойства numberOfEdits в пределах другого исходного файла, но вы не можете изменять его из другого исходного файла	<pre>struct TrackedString { private(set) var numberOfEdits = 0 var value: String = "" { didSet { numberOfEdits += 1 } } } var stringToEdit = TrackedString() stringToEdit.value = "This string will be tracked." stringToEdit.value += " This edit will increment numberOfEdits." stringToEdit.value += " So will this one." print("Количество изменений равно \(stringToEdit.numberOfEdits)") // Выведет "Количество изменений равно 3"</pre>
Пример определения уровня доступа у сеттера	<pre>public struct TrackedString { public private(set) var numberOfEdits = 0</pre>

элементы структуры (включая свойство numberOfEdits) получают уровень доступа internal по умолчанию	<pre> public var value: String = "" { didSet { numberOfEdits += 1 } } public init() {} </pre>
--	--

Инициализаторы

Пользовательским инициализаторам может быть присвоен уровень доступа ниже или равный уровню доступа самого типа, который они инициализируют (Единственное **исключение** составляют требуемые инициализаторы).

Требуемый инициализатор должен иметь тот же уровень доступа как и класс, которому он принадлежит.

Что же касается параметров функций и методов, типов параметров инициализатора, то они не могут быть более частными, чем собственный уровень доступа инициализатора.

Дефолтные инициализаторы

Дефолтный инициализатор имеет тот же уровень доступа, что и тип, который он инициализирует, если только тип не имеет доступа **public**. Для типа, у которого уровень доступа установлен **public**, дефолтный инициализатор имеет уровень доступа **internal**.

Если вы хотите, чтобы открытый (public) тип был инициализируемым при помощи инициализатора, который не имеет аргументов, когда используется в другом модуле, то вы должны явно указать такой инициализатор как часть определения типа.

Дефолтные почленные инициализаторы для типов структур

Дефолтные почленные инициализаторы для типов структур считаются частными (private), если есть свойства, которые имеют уровень доступа как private. **В противном случае**, инициализатор имеет уровень доступа internal.

Как и с дефолтным инициализатором выше, **если вы хотите открытый тип структуры**, который может быть инициализирован при помощи почленного инициализатора, когда используется в другом модуле, то вы должны предоставить открытый почленный инициализатор самостоятельно, как часть определения типа.

Протоколы и уровень доступа

Если вы хотите **присвоить явный уровень доступа протоколу**, то вы должны указать его во время определения протокола.

Уровень доступа каждого требования в процессе определения протокола **устанавливается на тот же уровень**, что и сам протокол. **Вы не можете установить** уровень доступа требований протокола отличным от того, который поддерживает сам протокол. **Это гарантирует**, что все требования протокола будут видимы любому типу, который принимает протокол.

Если вы определяете public протокол, то требования протокола требуют public уровня доступа для тех требований, которые они реализуют. **Это поведение отличается** от поведений других типов, где определение открытого типа предполагает наличие уровня internal у элементов этого типа.

Наследование протокола

Если вы определяете новый протокол, который наследует из другого существующего протокола, то новый протокол может иметь уровень доступа не выше чем протокол, который он наследует.

Соответствие протоколу

Тип может соответствовать протоколу с более низким уровнем доступа, чем сам тип.

Контекст, в котором тип соответствует конкретному протоколу, является минимумом из доступов протокола и типа. Если тип является public, но протокол, которому он соответствует является internal, то соответствие типа этому протоколу будет тоже internal.

Когда вы пишете или расширяете тип для того, чтобы он соответствовал протоколу, **вы должны быть уверены**, что реализация этого типа каждому требованию протокола, по крайней мере имеет один и тот же уровень доступа, что и соответствие типа этому протоколу. **Например**, если тип public соответствует протоколу internal, то реализация каждого требования протокола должна быть как минимум internal.

В Swift как и в Objective-C **соответствие протоколу является глобальным**. И тип не может соответствовать протоколу двумя разными способами в пределах одной программы.

Расширения и уровни доступа

Вы можете расширить класс, структуру или перечисление в любом контексте, в котором класс, структура или перечисление доступны.

Любой элемент типа, добавленный в расширение, имеет тот же дефолтный уровень доступа, что и типы, объявленные в исходном типе, будучи расширенными.

Аналогично вы можете отметить расширение, явно указав модификатор уровня доступа (например, `private extension`), для того чтобы указать новый дефолтный уровень доступа, который будут иметь элементы, определенные в этом расширении. Этот новый уровень доступа может быть переопределен для отдельных элементов расширением

Вы не можете предоставлять явный модификатор уровня доступа для расширения, если вы используете расширение для добавления соответствия протоколу. Вместо этого, собственный уровень доступа протокола используется для предоставления дефолтного уровня доступа для каждой реализации требования протокола внутри расширения.

Private свойства и методы в расширениях

Расширения, которые находятся в том же файле, что и сам класс/структура/перечисление, который(ую/ое) они расширяют, ведут себя точно так, как будто они являются частью расширяемого типа. И в результате вы можете:

- Объявлять приватные члены в оригинальном объявлении и получать доступ к ним через расширение
- Объявлять приватные члены в одном расширении и получать доступ к ним через другие расширения, если они находятся в том же файле
- Объявлять приватные члены в расширении и получать доступ к ним в оригинальном объявлении

Это поведение означает, что вы можете использовать расширения для организации вашего кода, независимо от того, имеют ли ваши типы приватные члены или нет.

Вы можете использовать расширение, чтобы реализовать требование протокола	<pre>protocol SomeProtocol { func doSomething() } struct SomeStruct { private var privateVariable = 12 } extension SomeStruct: SomeProtocol { func doSomething() { print(privateVariable) } }</pre>
---	---

Универсальные шаблоны

Уровень доступа для универсального типа или универсальной функции вычисляется как минимальный уровень доступа универсального типа или самой функции и уровень доступа ограничений любого типа ограничений для параметров типа.

Алиасы типов

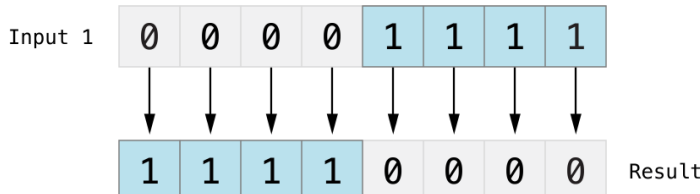
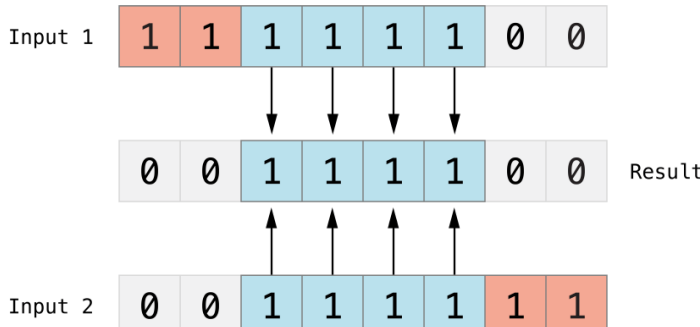
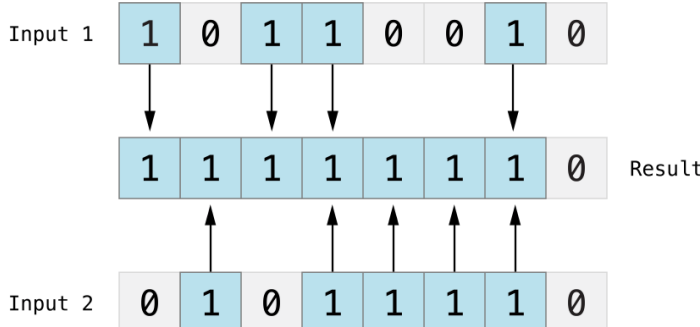
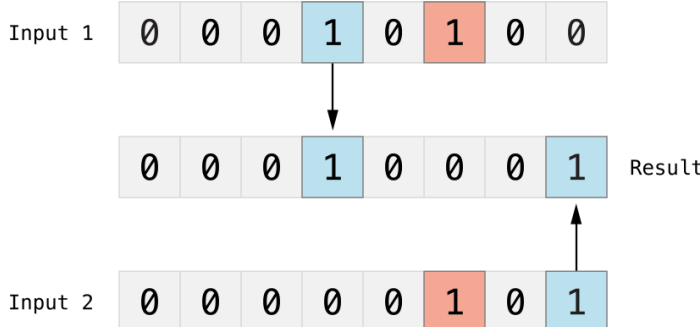
Любой алиас типа, который вы определяете, **рассматривается как отдельный тип для цели контроля доступа**. Алиас типа может иметь уровень доступа **такой же или ниже**, чем уровень доступа типа, псевдоним которого он создает.

Это правило так же применимо для **алиасов типа связанных типов**, используемых для удовлетворения несоответствий протоколу.

28. Продвинутое операторы

Побитовые операторы

Побитовые операторы позволяют манипулировать отдельными битами необработанных данных внутри структуры данных. Они часто используются в низкоуровневом программировании, например программирование графики или создание драйвера для устройства.

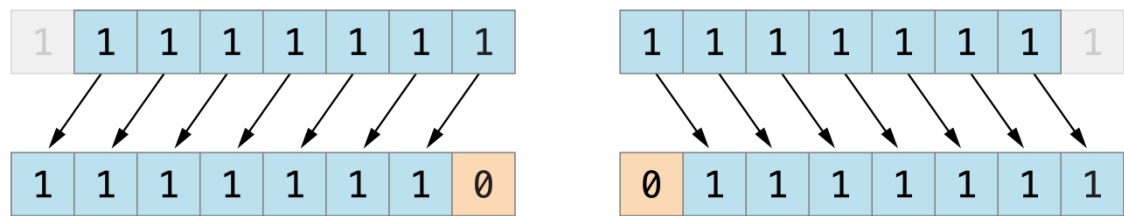
<p>Побитовый оператор NOT (~) инвертирует все битовые числа</p>	<pre>let initialBits: UInt8 = 0b00001111 let invertedBits = ~initialBits // равен 11110000</pre> 
<p>Побитовый оператор AND (&) комбинирует два бита двух чисел. Он возвращает новое число, чье значение битов равно 1, если только оба бита из входящих чисел были равны 1</p>	<pre>let firstSixBits: UInt8 = 0b11111100 let lastSixBits: UInt8 = 0b00111111 let middleFourBits = firstSixBits & lastSixBits // равен 00111100</pre> 
<p>Побитовый оператор OR () сравнивает биты двух чисел. Оператор возвращает новое число, чьи биты устанавливаются на 1, если один из пар битов этих двух чисел имеет бит равный 1</p>	<pre>let someBits: UInt8 = 0b10110010 let moreBits: UInt8 = 0b01011110 let combinedbits = someBits moreBits // равен 11111110</pre> 
<p>Побитовый оператор XOR или “оператор исключаящего OR” (^), который сравнивает биты двух чисел. Оператор возвращает число, которое имеет биты равные 1, когда биты входных чисел разные, и возвращает 0, когда биты одинаковые</p>	<pre>let firstBits: UInt8 = 0b00010100 let otherBits: UInt8 = 0b00000101 let outputBits = firstBits ^ otherBits // равен 00010001</pre> 

Операторы побитового левого и правого сдвига

Оператор побитового левого сдвига (<<) и **оператор побитового правого сдвига (>>)** двигают все биты числа влево или вправо на определенное количество мест, в зависимости от правил, которые определены ниже.

Поведение побитового сдвига (логический сдвиг) имеет следующие правила:

- Существующие биты сдвигаются вправо или влево на требуемое число позиций.
- Любые биты, которые вышли за границы числа, отбрасываются.
- На пустующие позиции сдвинутых битов вставляются нули.



Побитовый сдвиг в виде Swift кода

```
let shiftBits: UInt8 = 4
// 00001000 бинарный вид
shiftBits << 1      // 00001000
shiftBits << 2      // 00010000
shiftBits << 5      // 10000000
shiftBits << 6      // 00000000
shiftBits >> 2      // 00000001
```

Можно использовать побитовый сдвиг для кодирования и декодирования значений внутри других типов данных

Значение розового цвета #CC6699, что записывается в виде шестнадцатеричном представлении Swift как 0xCC6699. Этот цвет затем раскладывается на его красный(CC), зеленый(66) и голубой(99) компоненты при помощи побитового оператора AND (&) и побитового оператора правого сдвига (>>).

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16
// redComponent равен 0xCC, или 204
let greenComponent = (pink & 0x00FF00) > > 8
// greenComponent равен 0x66, или 102
let blueComponent = pink & 0x0000FF
// blueComponent равен 0x99, или 153
```

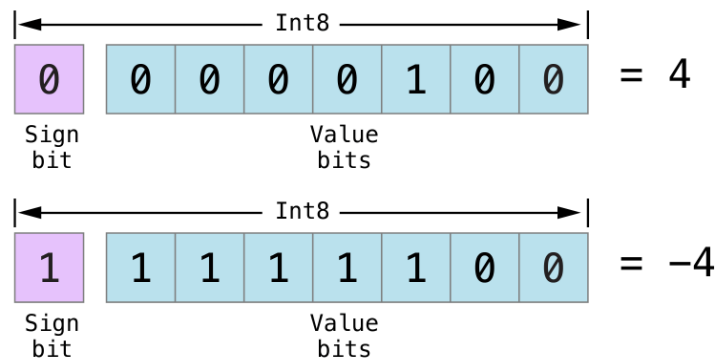
Красный компонент получен с помощью побитового оператора AND между числами 0xCC6699 и 0xFF0000. Нули в 0xFF0000 фактически являются “маской” для третьего и четвертого бита в 0xCC6699, тем самым заставляя игнорировать 6699, и оставляя 0xCC0000 в качестве результата.

После этого число сдвигается на 16 позиций вправо (>> 16). Каждая пара символов в шестнадцатеричном числе использует 8 битов, так что сдвиг вправо на 16 позиций преобразует число 0xCC0000 в 0x0000CC. Это то же самое, что и 0xCC, которое имеет целое значение равное 204.

Поведение побитового сдвига для знаковых целых чисел

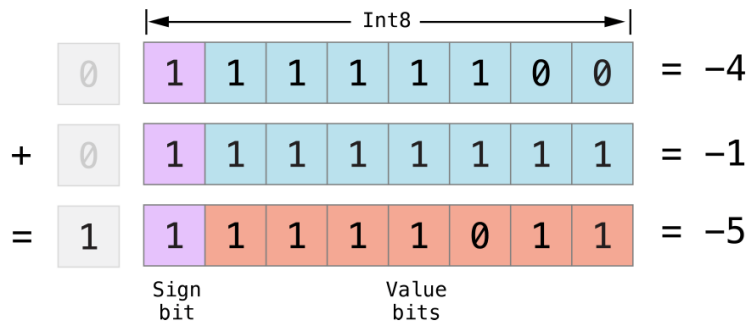
Знаковые целые числа используют первый бит (известный как знаковый бит) для индикации того, является ли число положительным или отрицательным. Значение знакового бита равное 0 свидетельствует о положительном числе, 1 - отрицательном. Остальные биты (известные как биты значения) хранят фактическое значение.

Отрицательные числа хранятся путем вычитания их абсолютного значения из 2 в степени n, где n - количество битов значения.



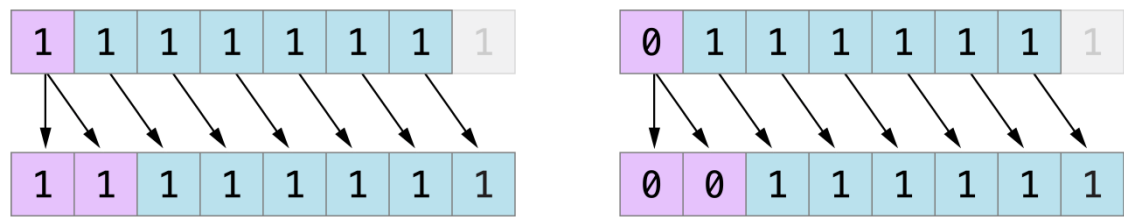
Кодирование отрицательных чисел известно под названием **дополнительный код**.

Вы можете добавить -1 к -4, просто выполняя стандартное сложение всех восьми битов (включая и восьмой бит), и отбрасывая все, что не поместится в ваши восемь бит.



Представление “дополнительного кода” также позволяет вам сдвигать биты отрицательных чисел влево и вправо, как в случае с положительными, и все так же умножая их при сдвиге влево или уменьшая их в два раза, при сдвиге на 1 место вправо. **Когда вы сдвигаете знаковое** число вправо, используйте то же самое правило, что и для беззнаковых чисел, но заполняйте освободившиеся левые биты знаковыми битами, а не нулями.

Эти действия гарантируют вам, что знаковые числа имеют тот же знак, после того как они сдвинуты вправо, и эти действия известны как **арифметический сдвиг**.



Операторы переполнения

Числа могут переполняться как в положительную, так и в отрицательную сторону. Все **операторы переполнения** начинаются с амперсанда (&).

Переполнение всегда переворачивает значение с самого большого на самое маленькое, **недополнение** всегда переворачивает самое маленькое число на самое большое.

<p>Пример того, что случится, когда беззнаковое значение позволяет переполнить себя, с использованием оператора (&+)</p>	<pre>var willOverflow = UInt8.max // willOverflow равняется 255, что является наибольшим числом, которое может держать UInt8 willOverflow = willOverflow &+ 1 // willOverflow теперь равно 0</pre> <p>UInt8</p> <p>0 1 1 1 1 1 1 1</p> <p>&+ 0 0 0 0 0 0 0 1</p> <p>= 1 0 0 0 0 0 0 0</p>
<p>пример с использованием оператора недополнения (&-)</p>	<pre>var unsignedOverflow = UInt8.min // unsignedOverflow равен 0, что является наименьшим возможным значением UInt8 unsignedOverflow = unsignedOverflow &- 1 // unsignedOverflow теперь равно 255</pre> <p>Int8</p> <p>0 0 0 0 0 0 0 0 = 0</p> <p>&- 0 0 0 0 0 0 0 1</p> <p>= 1 1 1 1 1 1 1 1 = 255</p>

Все **вычитание для знаковых целых чисел** проводится как прямое бинарное вычитание с учетом знакового бита, в качестве части вычитаемых чисел

```
var signedUnderflow = Int8.min
// signedUnderflow равняется -128, что является самым
// маленьким числом, которое может держать Int8
signedUnderflow = signedUnderflow &- 1
// signedUnderflow теперь равняется 127
```

Diagram illustrating the bit-level operation of signed underflow for Int8:

- Initial value: `signedUnderflow = Int8.min` (represented as `1 0 0 0 0 0 0 0` in binary, which is `-128`). The leftmost bit is the **Sign bit**, and the remaining seven are **Value bits**.
- Operation: `&- 1` (subtract 1).
- Result: `signedUnderflow` becomes `0 1 1 1 1 1 1 1` in binary, which is `127`.

Приоритет и ассоциативность

Оператор приоритета дает некоторым операторам более высокий приоритет по сравнению с остальными. В выражении сначала применяются эти операторы, затем все остальные.

Оператор ассоциативности определяет то, как операторы одного приоритета сгруппированы вместе (или ассоциированы друг с другом), то есть либо они сгруппированы слева, либо справа.

В Swift, как и в C, **оператор умножения (*) и оператор остатка (%)** имеют более высокий приоритет, чем **оператор сложения (+)**. Однако оператор умножения и **оператор остатка** имеют один и тот же приоритет по отношению друг к другу.

Порядок выполнения примера справа: <ul style="list-style-type: none"> 3 остаток от деления 4 равняется 3 3 умножается на 5 равно 15 2 плюс 15 и получаем 17 	<pre>2 + 3 % 4 * 5 // это равно 17, а не 5 // выражения выше и ниже эквивалентны 2 + ((3 % 4) * 5)</pre>
--	--

Правила приоритета и ассоциативности операторов Swift проще и более предсказуемые чем в C или Objective-C. Однако это означает, что **они ведут себя не так же** как они вели себя в этих C-языках. Будьте внимательны с тем, как ведут себя операторы взаимодействия при переносе кода в Swift.

Операторные функции

Классы и структуры могут предоставлять свои собственные реализации существующих операторов (**перегрузка существующего оператора**).

Нет такой возможности перегрузить оператор присваивания (=). Только составные операторы могут быть перегружены. **Тернарный оператор (a ? b : c) так же не может быть перегружен**.

Пользовательские классы и структуры **не получают дефолтной реализации эквивалентных операторов**, известных как “равен чему-то” оператор (==) или “не равен чему-то” (!=).

Swift предоставляет **синтезированные реализации операторов эквивалентности** для следующих пользовательских типов:

- Структур, имеющих только свойства хранения, соответствующие протоколу `Equatable`
- Перечислений, имеющих только ассоциированные типы, соответствующие протоколу `Equatable`
- Перечислений, не имеющих связанных типов

Пример отображает как можно реализовать арифметический оператор сложения (+) для пользовательской структуры.	<pre>struct Vector2D { var x = 0.0, y = 0.0 } extension Vector2D { static func + (left: Vector2D, right: Vector2D) -> Vector2D { return Vector2D(x: left.x + right.x, y: left.y + right.y) } } let vector = Vector2D(x: 3.0, y: 1.0) let anotherVector = Vector2D(x: 2.0, y: 4.0) let combinedVector = vector + anotherVector // combinedVector является экземпляром Vector2D, который имеет значения (5.0, 5.0)</pre>
---	---

Вы реализуете префиксный или постфиксный унарный оператор при помощи модификаторов prefix или postfix перед ключевым словом func , когда объявляете операторную функцию	<pre> extension Vector2D { static prefix func - (vector: Vector2D) -> Vector2D { return Vector2D(x: -vector.x, y: -vector.y) } } let positive = Vector2D(x: 3.0, y: 4.0) let negative = -positive // negative - экземпляр Vector2D со значениями (-3.0, -4.0) let alsoPositive = -negative // alsoPositive - экземпляр Vector2D со значениями (3.0, 4.0) </pre>
Оператор сложения-присваивания (+=) комбинирует в себе оператор добавления и оператор присваивания. Вы обозначаете левый входной параметр составного оператора как inout , потому что именно эта величина и будет изменена напрямую изнутри самой операторной функции.	<pre> extension Vector2D { static func += (left: inout Vector2D, right: Vector2D) { left = left + right } } var original = Vector2D(x: 1.0, y: 2.0) let vectorToAdd = Vector2D(x: 3.0, y: 4.0) original += vectorToAdd // original теперь имеет значения (4.0, 6.0) </pre>
Чтобы использовать операторы эквивалентности для проверки эквивалентности вашего собственного пользовательского типа, предоставьте реализацию для этих операторов тем же самым способом, что и для инфиксных операторов и добавьте соответствие протоколу стандартной библиотеки Equatable	<pre> extension Vector2D: Equatable { static func == (left: Vector2D, right: Vector2D) -> Bool { return (left.x == right.x) && (left.y == right.y) } } let twoThree = Vector2D(x: 2.0, y: 3.0) let anotherTwoThree = Vector2D(x: 2.0, y: 3.0) if twoThree == anotherTwoThree { print("Эти два вектора эквиваленты.") } // Выведет "Эти два вектора эквиваленты." </pre>
Пример так же реализует оператор “не равен чему-то” (!=) , который просто возвращает обратный результат оператора “равен чему-то”.	
Поскольку свойства x, y и z являются эквивалентными, Vector3D принимает стандартные реализации операторов эквивалентности	<pre> struct Vector3D: Equatable { var x = 0.0, y = 0.0, z = 0.0 } let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0) let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0) if twoThreeFour == anotherTwoThreeFour { print("These two vectors are also equivalent.") } // Выведет "These two vectors are also equivalent." </pre>

Пользовательские операторы

Вы можете объявить и реализовать ваши **собственные пользовательские операторы** в дополнение к стандартным операторам Swift.

Новые операторы объявляются на глобальном уровне при помощи ключевого слова **operator** и отмечаются модификатором **prefix, infix, postfix**.

Каждый пользовательский infix оператор принадлежит к своей приоритетной группе. **Группа приоритета** определяет приоритет оператора по отношению к другим инфиксным операторам, так же как и ассоциативность оператора.

Пользовательскому инфиксному оператору, который явно не размещен в приоритетной группе, предоставляется дефолтная группа приоритета, которая является по приоритету следующей после тернарного условного оператора.

Вы не указываете приоритет, когда определяете префиксный и постфиксные операторы. **Однако,** если вы воздействуете на операнд сразу двумя операторами (префиксным и постфиксным), то первым будет применен постфиксный оператор.

Пример определяет префиксный оператор <code>+++</code>	новый	<pre> prefix operator +++ extension Vector2D { static prefix func +++ (vector: inout Vector2D) -> Vector2D { vector += vector return vector } } var toBeDoubled = Vector2D(x: 1.0, y: 4.0) let afterDoubling = +++toBeDoubled // toBeDoubled имеет значения (2.0, 8.0) // afterDoubling так же имеет значения (2.0, 8.0) </pre>
Пример определяет инфиксный оператор <code>+ -</code> левой ассоциативности и с приоритетом <code>AdditionPrecedence</code> :	новый	<pre> infix operator + -: AdditionPrecedence extension Vector2D { static func +- (left: Vector2D, right: Vector2D) -> Vector2D { return Vector2D(x: left.x + right.x, y: left.y - right.y) } } let firstVector = Vector2D(x: 1.0, y: 2.0) let secondVector = Vector2D(x: 3.0, y: 4.0) let plusMinusVector = firstVector +- secondVector // plusMinusVector является экземпляром Vector2D со значениями (4.0, -2.0) </pre>

Result Builders

Result Builder - это определяемый вами тип, который добавляет *синтаксис для создания вложенных данных*, таких как список или дерево, естественным, декларативным образом.

Код, использующий result builder, может включать обычный синтаксис Swift, например `if` и `for`, для обработки условных или повторяющихся фрагментов данных.

Чтобы определить **Result Builder**, вы пишете атрибут **@resultBuilder** в объявлении типа.

В приведенном коде определены несколько типов рисования на одной линии с использованием звездочек и текста	<pre> protocol Drawable { func draw() -> String } struct Line: Drawable { var elements: [Drawable] func draw() -> String { return elements.map { \$0.draw() }.joined(separator: " ") } } struct Text: Drawable { var content: String init(_ content: String) { self.content = content } func draw() -> String { return content } } struct Space: Drawable { func draw() -> String { return " " } } struct Stars: Drawable { var length: Int func draw() -> String { return String(repeating: "*", count: length) } } struct AllCaps: Drawable { var content: Drawable func draw() -> String { return content.draw().uppercased() } } </pre>
Можно создать рисунок при помощи этих типов, вызвав их инициализаторы	<pre> let name: String? = "Ravi Patel" let manualDrawing = Line(elements: [Stars(length: 3), </pre>

	<pre> Text("Hello"), Space(), AllCaps(content: Text((name ?? "World") + "!")), Stars(length: 2),]) print(manualDrawing.draw()) // Выведет "***Hello RAVI PATEL!*" </pre>
<p>Result Builder позволяет вам переписать такой код, чтобы он выглядел как обычный код на Swift.</p> <p>Например, этот код определяет Result Builder под названием DrawingBuilder, который позволяет вам использовать декларативный синтаксис для описания рисунка</p>	<pre> @resultBuilder struct DrawingBuilder { static func buildBlock(_ components: Drawable...) -> Drawable { return Line(elements: components) } static func buildEither(first: Drawable) -> Drawable { return first } static func buildEither(second: Drawable) -> Drawable { return second } } </pre>
<p>Вы можете применить атрибут @DrawingBuilder к параметру функции, который превращает закрытие, переданное в функцию, в значение, которое построитель результатов создает из этого закрытия</p>	<pre> func draw(@DrawingBuilder content: () -> Drawable) -> Drawable { return content() } func caps(@DrawingBuilder content: () -> Drawable) -> Drawable { return AllCaps(content: content()) } func makeGreeting(for name: String? = nil) -> Drawable { let greeting = draw { Stars(length: 3) Text("Hello") Space() caps { if let name = name { Text(name + "!") } else { Text("World!") } } Stars(length: 2) } return greeting } let genericGreeting = makeGreeting() print(genericGreeting.draw()) // Выведет "***Hello WORLD!*" let personalGreeting = makeGreeting(for: "Ravi Patel") print(personalGreeting.draw()) // Выведет "***Hello RAVI PATEL!*" </pre>
<p>Например, Swift преобразует вызов caps(_:) в этом примере в код, подобный справо</p>	<pre> let capsDrawing = caps { let partialDrawing: Drawable if let name = name { let text = Text(name + "!") partialDrawing = DrawingBuilder.buildEither(first: text) } else { let text = Text("World!") partialDrawing = DrawingBuilder.buildEither(second: text) } return partialDrawing } </pre>
<p>Чтобы добавить поддержку записи для циклов в специальном синтаксисе рисования,</p>	<pre> extension DrawingBuilder { static func buildArray(_ components: [Drawable]) -> Drawable { return Line(elements: components) } } </pre>

добавьте buildArray(_ :).	метод	} let manyStars = draw { Text("Stars:") for length in 1...3 { Space() Stars(length: length) } }
------------------------------	-------	--