Оглавление

1.Getting started ……..2
2. Переменные и некоторые манипуляции с ними……...4
3. Числа……7
4. Строки и символы (Characters)……...9
5. Ветвления……12
6. Циклы……..14
7. Массивы (array)……16
8. Кортежи (tuples)……..22
9. Структуры (Structs)……..24
10. Множества (Sets)……..26
11. Хеши (hash)……...28
12. Переключатель switch – case……30
13. Enum……..33
14. Тип Optional……39
15. Функции……...45
16. Extension (переводится как расширение)…...49
17. Классы……...51
18. Генерики (Generics)……56
19. Closures (замыкания)……..60
20. Протоколы (protocols)……61
21. Subscript……66
22. Модификаторы……69
23. Создание новых операторов……...73

Программирование на Swift
(для любопытных)

Существует мнение, что язык программирования Swift скоро станет настолько популярным, что потеснит Ruby и Python. Значит, есть смысл познакомиться с ним поближе. Не будем начинать с общих характеристик языка, с ними можно познакомиться в википедии, а сразу приступим к его изучению. По ходу дела и увидим особенности и возможности языка Swift.

1.Getting started

Традиционная программа hello на Swift выглядит предельно просто:

print("Hello, World!")

Поместим этот код в файл ***prog.swift*** (дальше я все файлы примеров буду называть одним этим именем, поскольку хранить эти ничтожно малые программы нет смысла, а при желании какую-то из них выполнить повторно, проще всего скопировать код из этого текста). Теперь надо расположить файл там, где нравится, в командной строке перейти в эту директорию и выполнить команду:

swift prog.swift → Hello, World!

(Дальше я всюду стрелкой → буду заменять слова «получим», «будет выведено на терминал» и тому подобное. Сама программа этой стрелки не выводит).

Ясно, что при этом Swift должен быть установлен, а в переменной ***path*** должен быть записан адрес папки ***bin***.

Как видим, программа на Swift выполняется последовательно строчка за строчкой, как это обычно делают языки процедурного стиля. Не требуется объявлять ни пакеты, ни классы, ни функцию ***main***. Для оператора ***print*** не потребовался импорт никакого пакета или модуля, на самом деле это стандартная функция самого языка.

Команда:

swiftc prog.swift

позволяет получит рабочий файл ***prog.exe***. Конечно, команда ***swift*** тоже создаёт рабочий файл, но он записывается во временную папку и после выполнения не сохраняется. Обе команды ***swift*** и ***swiftc*** могут

выполняться с дополнительными ключами, позволяющими получить разную дополнительную информацию. Разобраться с этими ключами нетрудно самостоятельно.

Составим теперь программу приветствия, вставив в неё созданную нами (пользовательскую, не люблю этого слова, но что поделаешь) функцию:

```
func f(x: String, y: String) {
    print("Hello \(x) \(y)!")
}
let a = "Victor"
let b = "Borisov"
f(x: a, y: b)
```

и выполним её:

swift prog.swift → Hello Victor Borisov!

Значит, функция создаётся с использование служебного слова ***func*** а аргументы задаются с обязательным указанием их типа. Тип результата (в нашем случае ***Void***) тоже может быть указан с применением стрелки:

```
func f(x: String, y: String) -> Void {...}
```

Функция ***f*** только выводит полученные значения аргументов на терминал, но ничего не возвращает. При выводе текста в двойных кавычках выполняется интерполяция, для этого надо интерполируемое выражение заключить в круглые скобки с обратным слешем перед ними. Можно здесь применять и управляющие символы, например ***\n*** – перевод строки. Кстати, Swift не имеет оператора println, а оператор ***print*** сам выполняет перевод строки.

Конечно, мы могли бы обойтись и без интерполяции, например так:

```
print("Hello, " + x + " " + y + "!\n")
```

В этом контексте + знак конкатенации.

Далее в программе объявлены и инициированы текстовыми значениями две переменных: ***a*** и ***b***. Служебное слово ***let*** определяет неизменяемость (immutable) переменных, им нельзя дальше присвоить новое значение. Для изменяемых (mutable) переменных соответственно применяется слово ***var***. В нашем примере замена ***let*** на ***var*** по существу ничего бы не изменила. С одним служебным словом можно объявить сразу несколько переменных:

```
let a = "Victor", b = "Borisov"
```

Поскольку при объявлении переменной её тип не указан, значит Swift способен выводить тип по значению. Однако, тип можно было бы и объявить явно:

let a: String= "Victor", b: String = "Borisov"

При вызове функции и передаче ей значений требуется указывать имя аргументов ***f(x: a, y: b)*** и можно было бы подумать, что здесь именованные аргументы. Однако, это не так и поменять местами значения нельзя, то-есть вызов ***f(y: b, x: a)*** приведёт к ошибке. От необходимости указания имени аргументов можно избавиться, если в списке аргументов функции перед идентификаторами поставить знак (_). Между этим знаком и идентификаторами должен быть пробел:

```
func f(_ x: String, _ y: String) {
        print("Hello, \(x) \(y)!")
}
let a = "Victor"
let b = "Borisov"
f(a, b) → Hello, Victor Borisov!
```

Отметим ещё, что объявления функций в Swift должны обязательно предшествовать их вызовам.

Вообще-то вводить дополнительные переменные в нашем примере было необязательно и можно было бы сделать так:

```
func f(x: String, y: String) -> Void { print("Hello \(x) \(y)!\n") }
f(x: "Victor", y: "Borisov")
```

Мне нравится форматировать программу так, чтобы код был как можно компактнее и хорошо, если он весь помещается на одной странице.

2. Переменные и некоторые манипуляции с ними

При инициализации переменной к значению можно добавить блок, содержащий своеобразные инструкции ***willSet*** и ***didSet***. Этот блок будет всегда выполняться при присваивании переменной нового значения. Естественно, что переменная должна быть объявлена с ***var***. Посмотрим на примере, в чём тут фокус:

```
var x = 5 {
        willSet { print("x will set to \(newValue). x was previously \(x)") }
```

```
        didSet { print("x did set to \(x). x was previously \(oldValue)") }
}
```

x = 6 → x will set to 6. x was previously 5
x did set to 6. x was previously 5
x = 77 → x will set to 77. x was previously 6
x did set to 77. x was previously 6

Итак, присваивание переменной нового значения заставляет блок выполниться, при этом в теле инструкции ***willSet*** исходный идентификатор переменной (у нас ***x***) указывает на старое её значение, а новое значение доступно по стандартному имени ***newValue***. Соответственно, в теле инструкции ***didSet*** новое значение имеет исходный идентификатор (***x***), а старое значение доступно по стандартному имени ***oldValue***. В блоках этих инструкций можно запрограммировать какие-то действия с новыми и старыми значениями переменной. В документации такие переменные называются ***Property Observers***, что-то вроде свойства — наблюдатели. Пока, правда, не очень ясно, для чего их можно использовать.

Имеется ещё две подобные инструкции: ***get*** и ***set***, которые применяются при инициализации переменной. Они также имеют свои блоки, в которых можно запрограммировать какие-то действия. Инструкция ***get*** возвращает результат вычисления блока с помощью оператора ***return***, а ***set*** можно использовать для каких-нибудь действий в своём блоке при присваивании исходной переменной нового значения. Это новое значение доступно в блоке под именем ***newValue***. Посмотрим на примере:

```
let pi = 3.14; var r:Double = 0
var s: Double {
        get {return  pi * r * 2 }
        set { r = newValue / pi / 2.0 }
}
```

Сначала присвоим новое значение переменной ***r*** и выведем значение переменной ***s*** на печать.

```
r = 1
print("s = \(s) ")
```
→ s = 6.28

Блок инструкции ***get*** при обращении к переменной ***s*** выполнился и она получила значение. Попробуем теперь присвоить новое значение самой переменной ***s***:

s = 14
print("r = \(r)") → r = 2.229…

Теперь выполнился блок инструкции ***set***, в котором было вычислено значение для ***r*** с использованием нового значения ***s***.

Инструкция ***get*** может применяться самостоятельно для инициализации переменной:

```
var s: Double {
    get { return pi * r * 2 }
}
```

Впрочем, здесь можно ограничиться и одним только оператором ***return***:

```
var s: Double { return pi * r * 2 }
```

Переменные, объявленные вне блока, функции, класса, структуры и так далее, считаются глобальными и могут быть использованы всюду в отличие от локальных переменных, доступных только внутри тех объектов, где они объявлены. Есть одна важная особенность у глобальных переменных — все они относятся к «ленивым» и до первого их использования не вычисляются. Поясним на примере, о чём тут речь:

```
let m = [1, 2, 3, 4, 5]
var x = m.reduce(0) { a, e in return a + e }
class A {
    var y: Int { return x }
}
let p = A()
print(p.y)
```
→ 15

Здесь переменная ***x*** объявлена вне класса ***A*** и, значит, она глобальна по отношению к классу. В строке

```
let x = m.reduce(0) { a, e in return a + e }
```

переменная ***x*** инициирована суммой массива ***m*** (позже мы это рассмотрим). Переменная **y** является переменной экземпляра класса A (в документации такие переменные называют свойствами класса), ей просто присваивается значение глобальной переменной ***x***.

“Ленивость” переменной ***x*** заключается в том, что она не вычисляется до первого её использования, то-есть, в нашем примере до строки ***print(p.y)***. Такая особенность переменных полезна в том случае, когда их вычисление связано с трудоёмкими (дорогостоящими)

вычислениями. Иногда это позволяет исключать ненужную работу компьютера и экономить время.

Переменные экземпляра (свойства) сами могут быть объявлены ленивыми с использование модификатора ***lazy***, например:
lazy var z: String
Ленивые переменные всегда должны объявляться со словом ***var***.

3. Числа

Swift поддерживает следующие типы чисел: ***Int*** и беззнаковое ***Uint***, ***Float32*** (можно ***Float***), ***Float64*** (или ***Double***), ***Int8***, ***Int16***, ***Int32,*** соответствующие беззнаковые ***UInt8*** и так далее. Кроме уже знакомого способа задания типа через двоеточие:
let x: Double = 67.098
это же можно делать с применением служебного слова ***as***:
let x = 67.098 as Double

Для удобства в больших (длинных) числах можно вставлять знак подчёркивания:
let x = 12_345_678_987
При выводе эти разделительные знаки удаляются.

Для шестнадцатеричных чисел впереди ставится два знака ***0x***, для восьмеричных — ***0o***, для двоичных — ***0b***.

Для чисел с плавающей запятой применима научная нотация:
let x = 1.234e2 или ***1.234E2***.

Конвертирование чисел к типу String выполняется с помощью функции String:
let s = String(0.078)
или можно использовать интерполяцию:
let y = "\(x)"; print(y) → 123.4 в текстовом виде
С обратным преобразованием есть некоторые проблемы, например:
var x = "123.098"
let y = Double(x); print(y) → Optional(123.098)
Здесь мы получили некоторый новый тип числа - ***Optional***. Если будет иметь место предупреждение, то надо добавить ***as Any***:
print(y as Any) → Optional(123.098)

Фактически это те же числа, но дополнительно этот тип может принимать значение ***nil***. Фактически ***nil*** это отсутствие значения,

отсюда и название типа ***Optional***, то-есть переменная, не обязательно имеющая значение. Выполнять арифметические операции с числами такого типа нельзя. Но получить из ***Optional*** обычный тип числа просто, достаточно поставить в конце восклицательный знак:
let z = y!; print(z) → 123.098
Можно, конечно всё сделать одновременно:
let y = Double(x)!; print(y) → 123.098
Позже мы рассмотрим этот тип ***Optional*** и зачем он нужен более подробно.

Конвертирование типов чисел выполняется функциями, одноимёнными с типами:
let x: Int = 34
let y = Double(x); print(y) → 34.0

Функция ***round*** выполняет округление чисел по обычному правилу, функция ***ceil*** увеличивает число до ближайшего целого, а функция ***floor*** наоборот уменьшает число до целого. Все математические функции расположены в пакете ***Foundation***, который надо предварительно импортировать:
import Foundation
let x = round(1.2); print(x) → 1
let y = ceil(1.2); print(y) → 2
let z = ceil(-2.7); print(z) → -2
let t = floor(-2.7); print(t) → -3
Наконец, функция ***Int***, конвертирующая дробные числа в целые просто отбрасывает дробную часть:
let s = Int(2.7); print(s) → 2
Функция ***pow*** выполняет возведение в степень:
let x = pow(5.0, 2.0); print(x) → 25.0

Все эти функции работают с типами ***Float32*** и ***Float64***.

Арифметическая операция ***%*** возвращает остаток от деления целых чисел:
print(17 % 5) → 2

Есть также встроенный оператор ***truncatingRemainder*** (ненавижу подобные идентификаторы), который возвращает остаток от деления двух чисел с плавающей точкой, он должен вызываться так:
x.truncatingRemainder(dividingBy: y) → остаток от деления x / y
print(15.7.truncatingRemainder(dividingBy: 4.5)) → 2.2

4. Строки и символы (Characters)

В Swift применяются строки в двойных кавычках и многострочные в тройных кавычках (три двойных кавычки). Тип ***Character*** представлен одиночными символами в двойных кавычках:
let x: Character = "@"; print(x) → @

Как мы уже видели, конкатенация строк выполняется с помощью знака плюс. Кроме того к существующей строке можно добавить текст с помощью функции ***append***:
var x = "Hello"
x.append(", " + "World!")
print(x) → Hello, World!

Метод ***joined()*** позволяет объединить массив строк в одну строку:
let x = ["a", "b", "c"].joined()
print(x) → abc

Функция ***Array*** позволяет превратить строку в массив символов:
let x = "Hello"
let y = Array(x)
print(y) → ["H", "e", "l", "l", "o"]

По символам строки можно выполнять цикл, как по элементам массива:
var y = "Hello"
for c in y {print(c)} →
H
e
l
l
o

Метод ***count*** позволяет узнать длину строки в символах:
let n = y.count; print(n) → 5

Метод ***isEmpty*** позволяет проверить, не является ли строка пустой:
var x = "", y = "Hello"
print(x.isEmpty) → true
print(y.isEmpty) → false

Методы ***hasPrefix*** и ***hasSuffix*** позволяют проверить наличие в начале или в конце строки группы заданных символов:
print("fortitude".hasPrefix("fort")) → true
print("Swift Language".hasSuffix("age")) → true

Метод ***reversed()*** выполняет реверс в строке, при этом требуется к результату применить функцию ***String***:

let x = "Hello"
let z = String(x.reversed())
print(z) → olleH

В Swift нельзя извлекать символы из строки непосредственно по индексу, например ***x[2]*** – это не работает и тут потребуется более сложный приём, но не будем пока его обсуждать.

При выводе чисел применяется обычный способ форматирования:

import Foundation
let x = 24.705
let y = String(format: "%10.5f", x)
print("x = " + y) → x = 24.70500

Здесь ***10*** — общее количество позиций, а ***5*** — число знаков после точки. Можно, конечно, не вводить промежуточную переменную ***y***:

print("x = " + String(format: "%10.5f", x))

При форматированном выводе необходимо импортировать модуль ***Foundation***.

Для целых чисел вместо ***f*** надо поставить ***d***, а если перед заданным числом позиций поставить ***0***, то лишние позиции перед числом будут заполнены нулями:

let x = 24
let y = String(format: "%05d", x)
print("x = " + y) → x = 00024

Для шестнадцатеричных чисел применяется буква ***x*** и даже можно делать форматированный вывод для чисел произвольной системы счисления, только не ясно, зачем это надо.

Метод ***uppercased()*** переводит все буквы в верхний регистр:

let x = "Hello"
let y = x.uppercased(); print(y) → HELLO

А метод ***lowercased()*** - в нижний регистр.

Метод ***filter*** позволяет просматривать текст посимвольно и выполнять над ним разные действия. Посмотрим, как работает этот метод на конкретном примере. Пусть нам требуется удалить из текста все знаки, кроме заданных нами:

```
func f(x: String, s: Array<Character>) -> String {
        return String(x.filter { s.contains($0) })
}
```

```
let t = "Swift 4.0 Come Out"
let z = Array("""
abcdefghijklmnopqrstuvwxyzABCDEFGHIJ
KLKMNOPQRSTUVWXYZ
""")
let y = f(x: t, s: z)
print(y) → SwiftComeOut
```

Всю работу выполняет функция ***f***, которая принимает два аргумента: заданный текст ***x*** и массив символов ***s***, которые мы хотим оставить в полученном тексте. Выражение ***Array<Character>*** показывает, что тип элементов массива ***Character*** должен располагаться в угловых скобках. Методу ***filter*** должен быть передан блок, в данном случае ***{ s.contains($0) }***, где ***contains*** служебное слово, а ***$0*** зарезервированный идентификатор переменной, в которую будут передаваться символы из строки ***x***. Блок будет возвращать ***true*** для тех символов, которые содержаться в ***x*** и в ***s*** одновременно. И только эти символы метод ***filter*** накапливает в результате. При инициализации массива ***z*** использована строковая константа в две строки. Swift требует, чтобы тройные кавычки располагались на отдельных строках. Строка автоматически преобразуется в массив символов.

Массив ***z*** содержит все буквы латинского алфавита и значит из исходной строки будут удалены все символы, кроме букв латиницы, в частности все цифры, как в примере. Если мы захотим сохранить пробелы, достаточно внести пробел в массив ***z***.

По общепринятой терминологии метод ***filter*** можно называть словом итератор. Он организует цикл по символам строки или, как увидим далее, по элементам массива или других коллекций. Итератор ***filter*** в результате возвращает такие же коллекции.

В документации блоки называются closure (замыкание). Обычно так называют анонимные функции или вообще функции, передаваемые другим функциям в качестве аргументов. В случае итераторов действительно можно блок рассматривать, как анонимную функцию. В дальнейшем будем считать, что термины блок и closure означают одно и то же.

Ещё один пример иллюстрирует применение метода ***filter*** более эффектно и наглядно. Пусть нам надо подсчитать число вхождений заданного символа в строку:

let x = "Hello World"
let y: Character = "o"
let z = x.filter { $0 == y }.count
print(z) → 2

Здесь блок метода ***filter*** последовательно сравнивает символы строки с заданным символом и выбираются только те, для которых блок возвращает ***true***. Создаётся строка из выбранных символов а метод ***count*** вычисляет её длину.

Пусть теперь требуется выбрать из текста только символы в нижнем регистре:

let x = "Hello World"
let z = x.filter { String($0) == String($0).lowercased() }
print(z) → ello orld

Здесь потребовалось преобразовать переменную ***$0*** к типу ***String*** потому, что метод ***lowercased()*** не работает с переменными типа ***Character***.

Иногда требуется удалить из строки начальные пробелы и управляющий символ перевода строки:

import Foundation
let x = " Swift Language \n"
let y = x.trimmingCharacters(in: .whitespacesAndNewlines)
print(y) → Swift Language

Здесь метод ***trimmingCharacters*** выполняет удаление, а ***.whitespacesAndNewlines*** представляет множество, содержащее то, что требуется удалить (о синтаксисе с точкой поговорим позже). Слово ***in:*** служебное.

Или можно так:

let y = x.trimmingCharacters(in: .newlines)
let y = x.trimmingCharacters(in: .whitespaces)

(Метод ***trimmingCharacters*** находится в модуле ***Foundation***)

5. Ветвления

Переменные типа ***Bool*** могут иметь только два значения: ***true*** и ***false***. Условный оператор ***if else*** имеет обычный синтаксис:

```
func f(_ x: Bool) {
    if x { print("it's true!") }
    else { print("it's false!") }
```

}
f(true) → it's true!

Здесь использован «синтаксический сахар»: если перед идентификатором аргумента поставить знак подчёркивания (_), то при вызове функции можно опускать этот идентификатор.

Swift имеет тернарный условный оператор вида условие ? выражение1 : выражение2. Если условие даёт ***true***, выполняется выражение1, иначе - выражение2. Покажем его применение:

```
func f(_ x: Bool) {
        let y = x ? "green" : "red"
        print("The animal is \(y)")
}
```

f(true) → The animal is green
f(false) → The animal is red

Здесь подразумевается: ***let y = (x ? "green" : "red")***, и эти скобки можно поставить явно. Можно не вводить новую переменную, а написать:

```
func f(_ x: Bool) {
        let x = x ? "green" : "red"
        print("The animal is \(x)")
}
```

На месте выражение1 и выражение2 может быть блок, или, например, вызов функции:

```
func g1() { print("Hello, Anna") }
func g2() { print("Hello, Marta") }
func f(_ x: Bool) { x ? g1() : g2() }
```

f(true) → Hello, Anna
f(false) → Hello, Marta

Можно применять обычные логические операции: ***OR (||)***, ***AND (&&)*** и ***XOR (^)***:

```
if (10 < 20) && (20 < 10) { print("Expression is true")}
        else {print("Expression is false") }
```
→ Expression is false

Знак отрицания (***!***) меняет значение логической переменной на обратное:

print(!true) → false

Оператор ***switch-case*** рассмотрим отдельно несколько позже.

6. Циклы

Swift использует традиционный цикл ***for***. Наиболее просто применение цикла ***for*** для работы с коллекциями разных видов. В частности, такой цикл применяется с рангами:

for i in 0...3 { print(i) } → 0 1 2 3

Или можно использовать ранг, не включающий верхнюю границу:

for i in 0..<3 { print(i) } → 0 1 2

Подобным же образом цикл работает с такими коллекциями, как массивы и множества (set):

```
let m = ["James", "Emily", "Miles"]
for name in m { print(name) } →
James
Emily
Miles
```

Если требуется извлекать также и индексы, надо воспользоваться методом ***enumerated***:

```
let m = ["James", "Emily", "Miles"]
for (i, n) in m.enumerated() { print("The index of \(n) is \(i).") } →
The index of James is 0.
The index of Emily is 1.
The index of Miles is 2.
```

С другими сортами коллекций метод ***enumerated*** не работает.

Метод ***reversed*** меняет цикл на обратный:

for i in (0...3).reversed() { print(i) } → 3 2 1 0

Встроенная функция ***stride***, принимающая три аргумента, позволяет изменять параметр цикла на заданный шаг:

for i in stride(from: 10, to: 0, by: -3) { print(i) } → 10 7 4 1

Здесь слова ***from***, ***to*** и ***by*** служебные. Слово ***to*** можно заменить на ***through***, если нравится.

Имеется также традиционный цикл ***repeat - while***:

```
var i: Int = 2
repeat { print(i);  i += 1 } while i < 7 → 2 3 4 5 6
```

Или просто ***while***:

```
var x = 1
while x < 6 { print(x); x += 1 } → 1,2,3,4,5
```

Ключевое слово ***where*** позволяет ввести в цикл ***for*** какое-нибудь условие:

for i in 4...10 where i % 2 == 0 { print(i) } → 4 6 8 10

Условия могут самые разные, например, выделим из массива слова, содержащие букву ***s***:

let m = ["James", "Emily", "Miles"]
for t in m where t.contains("s") { print(t) } → James Miles

Метод ***contains*** возвращает ***true***, если в ***t*** присутствует символ ***"s".***

Есть также вариант цикла ***for case***. Например, вычислим количество тех пар в массиве, которые на втором месте имеют значение ***0***:

let m = [(5, 0), (5, 31), (31, 0)]
var i = 0
for case (_, 0) in m { i += 1 }
print(i) → 2

Ненужный нам первый элемент в паре мы заменили на placeholder.

Если массив имеет тип ***Optional***, то-есть может иметь элементы со значением ***nil***, то цикл ***for-case*** должен иметь такой синтаксис:

let m = [31, 5, nil]
for case let x? in m { print(x) } → 31 5

Выражение ***let x?*** возвращает ***true*** для всех элементов массива, кроме ***nil***.

Наиболее просто программировать циклы с применением итераторов. Swift имеет много разных итераторов, например ***forEach***:

```
let m = [1,2,3,4]
m.forEach {
    if $0 == 3 { return }
    print($0 * $0) → 1 4 16
}
```

Здесь с помощью оператора ***return*** мы пропустили элемент со значением ***3***.

Итераторы принимают блок (closure), с помощью которого можно выполнять различные операции над элементами массива или других коллекций. Мы раньше уже применяли итератор ***filter***, дальше мы познакомимся со многими итераторами.

Приведём ещё пример выхода из цикла с помощью оператора ***break***:

```
var m = [ "Nicole", "Richard", "Brian", "Novak", "Vick"]
var i = 0
for x in m {
```

```
    if x == "Novak" { break }
    i += 1
}
```
print("Novak is located on position [\(i)].") → Novak is located on position [3].

7. Массивы (array)

Пустой массив ***m*** можно объявить тремя способами:
var m: [String] = []
var m = [String]()
var m = Array<String>()

Во всех этих трёх случаях ***m*** представляет изменяемый (mutable) массив с элементами типа ***String***. Применение модификатора ***let*** позволяет создать неизменяемый массив. Можно совмещать объявление массива с его инициализацией:
let m = [2, 4, 7]

Здесь тип элементов выводится компилятором. Также тип можно задавать:
let m: [Double] = [2, 4, 7]; print(m) → [2.0, 4.0, 7.0]
Или так:
let m = [2, 4, 7] as [Double]; print(m) → [2.0, 4.0, 7.0]
Достаточно задать тип одного элемента и тогда тип остальных будет выведен:
let m = [2 as Double, 4, 7]; print(m) → [2.0, 4.0, 7.0]
Массивы могут содержать элементы разных типов, но тогда сам массив будет иметь тип ***Any*** и компилятор требует указывать этот тип явно:
var m: Any = [2, 4.08, "Marta", true];

С помощью инструкций ***repeating - count*** можно инициировать массивы с повторяющимися элементами:
let m = Array(repeating: "Anna", count: 3)
print(m) → ["Anna", "Anna", "Anna"]

В массивы можно трансформировать другие коллекции:
let h = ["foo" : 4, "bar" : 6]
let m = Array(h);
print(m) → [(key: "foo", value: 4), (key: "bar", value: 6)]

Здесь хеш ***h*** трансформирован в массив ***m***. Этот массив имеет сложный тип ***[(String, Int)]***, применение таких массивов и хешей мы ещё рассмотрим далее.

Многомерные массивы в Swift создаются с помощью вложенных массивов:

let m = [[1, 2, 3], [4, 5, 6]]
print(m[1][2]) → 6

Этот массив имеет тип ***[[Int]]*** или ***Array<Array<Int>>***.

Можно создавать многомерные массивы с повторяющимися элементами:

var m = Array(repeating: Array(repeating: Array(repeating: "a",count: 3),count: 3),count: 2)
print(m) → [[["a", "a", "a"], ["a", "a", "a"], ["a", "a", "a"]], [["a", "a", "a"], ["a", "a", "a"], ["a", "a", "a"]]]

Метод (итератор) ***compactMap*** позволяет извлечь из массива элементы по заданному типу:

let m: [Any] = [1, "Hello", 2, true, false, "World", 3]
let x = m.compactMap { $0 as? Int }
print(x) → [1, 2, 3]
let y = m.compactMap { $0 as? String }
print(y) → ["Hello", "World"]

Метод (итератор) ***reduce*** принимает блок, в котором можно выполнить над элементами массива нужные вычисления, например вычислить их сумму или произведение:

let m = [1,2,3,4,5]
let s = m.reduce(0) { a, e in return a + e }; print(s) → 15
let p = m.reduce(1) { a, e in return a * e }; print(p) → 120

Здесь ***a*** – это аккумулятор, а аргумент метода ***reduce*** задаёт его начальное значение. Слово ***in*** служебное. Переменная ***e*** принимает значения элементов массива. Идентификаторы ***a*** и ***e*** произвольные.

Метод (итератор) ***flatMap*** позволяет элементы массива типа ***String*** распаковать посимвольно:

let m = ["Anna", "Marta"]
let x = m.flatMap { $0 };
print(x) → ["A", "n", "n", "a", "M", "a", "r", "t", "a"]

Этот метод распаковывает также и вложенные массивы:

let m = [[1, 3], [4], [6, 8, 10], [11]]
let x = m.flatMap { $0 }

print(x) → [1, 3, 4, 6, 8, 10, 11]

Уже знакомый нам метод ***filter*** может работать и с массивами. Например, запрограммируем отбор из массива чётных чисел:

```
import Foundation
func f(x: Array<Int>) -> Array<Int> {
      return (x.filter { $0 % 2 == 0 })
}
```

let m = f(x: [1,2,3,4,5,6]); print(m) → [2, 4, 6]

Оператор ***%*** возвращает остаток от деления, а итератор ***filter*** отбирает те элементы, для которых блок возвращает ***true***.

Массив, содержащий целые числа и значения ***nil*** будет иметь тип ***[Int?]***, например:

```
let m : [Int?] = [nil, 1, nil, 2, nil, 3];
```

Если вывести такой массив на терминал, то получим:

print(m) → [nil, Optional(1), nil, Optional(2), nil, Optional(3)]

Итератор ***compactMap*** позволяет удалить элементы ***nil*** и получить массив типа ***[Int]***:

```
let x = m.compactMap { $0 }
print(x) → [1,2,3]
```

Напоминаю, что ***$0*** – зарезервированная переменная, принимающая значения элементов массива.

Swift позволяет использовать ранги, например:

```
let m = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let r = 2...4
let s = m[r]; print(s) → ["Bonjour", "Welcome", "Hi"]
```

Здесь ***r*** – ранг, а массивы, подобные ***s*** обычно называют срезами. Срезы в Swift не самостоятельные массивы, в этом можно убедиться, если извлечь из среза элемент:

print(s[2]) → Bonjour

Значит, индексы элементов в срезе те же, что и в исходном массиве. Чтобы получить из среза массив, надо применить директиву Array:

let x = Array(s); print(x[2]) → Hi

Можно, конечно, без промежуточных шагов:

```
let x = Array(m[2...4])
```

Удалить элемент массива по индексу можно с помощью встроенной функции ***remove(at: i)***. Здесь ***at*** служебное слово, а ***i*** индекс:

```
var m = [1,2,3,4,5]
```

m.remove(at: 2); print(m) → [1, 2, 4, 5]

Swift не имеет встроенной функции, удаляющей элементы массива по значению, но такую функцию можно создать, расширив возможности ***remove(at: i)***. Это можно сделать, например, так:

```
extension Array where Element: Equatable {
    mutating func remove(_ e: Element) {
        _ = index(of: e).flatMap {self.remove(at: $0)}
    }
}
var m = ["abc", "lmn", "pqr", "stu", "xyz"]
m.remove("lmn"); print(m) → ["abc", "pqr", "stu", "xyz"]
m.remove(at: 2); print(m) → ["abc", "pqr", "xyz"]
```

Здесь использованы несколько инструментов Swift, пока нам не знакомых, но само удаление выполняет итератор ***flatMap*** с блоком, включающем функцию ***remove(at: i)***. Обратите внимание, что эта функция применяется к массиву под именем ***self***. Позже мы разберёмся со всем этим.

При попытке удалить несуществующий элемент ошибка не генерируется, а массив остаётся неизменным. Отметим ещё, что после модификации сама функция ***remove(at:)*** по-прежнему работает.

Для сортировки массивов Swift имеет встроенную функцию ***sorted()***:

```
let m = ["Hello", "Bonjour", "Salute", "Ahola"]
let s = m.sorted(); print(s) → ["Ahola", "Bonjour", "Hello", "Salute"]
```

Кроме того, есть ещё функция ***sort()***, сортирующая массив «на месте». Ясно, что массив при этом должен быть mutable, т. е. объявлен с модификатором ***var***:

```
var m = ["Hello", "Bonjour", "Salute", "Ahola"]
m.sort(); print(m) → ["Ahola", "Bonjour", "Hello", "Salute"]
```

Обе функции ***sorted()*** и ***sort()*** могут принимать в качестве аргумента closure с помощью которого можно, например, выполнить сортировку «по убыванию»:

```
let m = ["Hello", "Bonjour", "Salute", "Ahola"]
let s = m.sorted() { $0 > $1 }
print(s) → ["Salute", "Hello", "Bonjour", "Ahola"]
```

Очевидно, что здесь стандартные переменные ***$0*** и ***$1*** принимают попарно элементы массива для сравнения по величине. Заменив знак > на < получим сортировку «по возрастанию».

Применение closure позволяет, например, отсортировать числа разных знаков по их абсолютному значению:

let m = [-77, 52, -25, 86, -34]
let s = m.sorted() { abs($0) < abs($1) }
print(s) → [-25, -34, 52, -77, 86]

Забегая вперёд, приведу ещё пример сортировки структуры, с которыми подробнее познакомимся далее:

```
struct L {
        let name : String
        let meters : Int
}
var x = [L(name: "Empire", meters: 443),
L(name: "Eifell", meters: 300),
L(name: "The Shard", meters: 310)]
let y = x.sorted {$0.name < $1.name}; print(y) → [L(name: "Eifell",
meters: 300), L(name: "Empire", meters: 443), L(name: "The Shard",
meters: 310)]
x.sort {$0.meters < $1.meters}; print(x) → [L(name: "Eifell", meters:
300), L(name: "The Shard", meters: 310), L(name: "Empire", meters:
443)]
```

Здесь объявлена структура ***L*** с двумя элементами: ***name : String*** и ***meters : Int***. В конкретном экземпляре структуры ***x*** приведены данные о высоте в метрах для трёх башен. Функции ***sorted*** и ***sort*** позволяют отсортировать структуру по именам башен или по их высоте.

Методы ***min*** и ***max*** позволяют найти минимальный и максимальный элемент в массиве. Эти методы применимы и к структурам:

let h = x.min { $0.meters < $1.meters }
print(h!) → L(name: "Eifell", meters: 300)

Знак ! добавлен здесь для удаления слова optional, см. выше.

Итератор ***map***, пожалуй, один из самых популярных. Он позволяет с помощью closure выполнять над элементами массива любые действия, например:

let m = [1.0, 2.0, 3.0, 4.0, 5.0]
let s = m.map { pow($0,2.0) }
print(s) → [1.0, 4.0, 9.0, 16.0, 25.0]

Для изменения типа элементов есть также такая конструкция:

var m = [-77, 52, -25, 86, -34]

let s = m.map(String.init)
print(s) → ["-77", "52", "-25", "86", "-34"]

Функция ***reverse()*** выполняет реверс массива «на месте»:
m.reverse(); print(m) → [-34, 86, -25, 52, -77]

Методы ***isEmpty*** и ***count*** работают с массивами точно также, как и со строками:
var m = [1,2,3,4,5]
print(m.isEmpty) → false
print(m.count) → 5

Swift имеет несколько функций, позволяющих добавлять и удалять элементы массива:
v***ar x = [1,2,3,4,5]***
x.append(6); print(x) → [1, 2, 3, 4, 5, 6]
var y = [7,8,9,10]
x += y; print(x) → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x.remove(at: 3); print(x) → [1, 2, 3, 5, 6, 7, 8, 9, 10]
x.removeLast(); print(x) → [1, 2, 3, 5, 6, 7, 8, 9]
x.removeFirst(); print(x) → [2, 3, 5, 6, 7, 8, 9]

Ещё четыре полезные функции:
var x = [6,2,3,8,5]
print(x.first) → Optional(6)
print(x.last) → Optional(5)
print(x.max()) → Optional(8)
print(x.min()) → Optional(2)

Тип ***Optional*** потому, что результат может быть ***nil***. Напоминаю, что для удаления Optional используется восклицательный знак на конце.

Имеется очень полезная функция ***zip()***, которая принимает два массива и группирует их элементы с одинаковыми индексами в пары. Над элементами пар можно затем выполнять какие-нибудь действия. Например, с помощью итератора ***map*** можно сформировать массив из этих пар:
let x = [1, 2, 4, 7]
let y = ["Anna", "Marta", "Masha"]
let s = zip(x, y).map { ($0,$1)}
print(s) → [(1, "Anna"), (2, "Marta"), (4, "Masha")]

Как видим, размеры массивов не обязательно должны быть одинаковыми, тогда число пар будет равно размеру более короткого массива.

В следующем примере с помощью итератора ***filter*** выберем только те пары, между элементами которых есть указанная зависимость:

let x = [1, 2, 4]
let y = [2, 5, 8]
let s = zip(x, y).filter { $1 == (2 * $0)}
print(s) → [(1, 2), (4, 8)]

8. Кортежи (tuples)

Кортежи это элементы через запятую в круглых скобках. Типы элементов могут быть любыми:

let x = ("one", 2, "three", false)

Как и у массивов, элементы кортежей индексируются, а для извлечения по индексу применяется такой простой синтаксис:

print(x.2) → three

Элементы кортежей могут быть именованными и по имени элемент можно извлечь также, как по индексу:

let y = (Vasja: 18, Kolja: 15, Petja: 23)
print(y.Petja) → 23

При этом сохраняется возможность извлечения и по индексу тоже:

print(y.1) → 15

В таком виде кортежи очень похожи на хеши, о которых будем говорить далее.

Тип элементов кортежа можно задать предварительно:

var x: (a :Int?, b: String, c: Double)
x = (25, "Hello", 3.89)
print(x) → (a: Optional(25), b: "Hello", c: 3.89)

Здесь тип ***Int?*** Означает целое с возможным значением ***nil*** (тип Optional).

Элементами кортежей может быть что угодно, например это могут быть функции:

f***unc f(x: String) -> () { print(x) }***
let k: (a: Int, b: Int, c: (String) -> (), d: Int)
k = (1, 2, f, 3)
k.c("Hello") → Hello

В кортеже ***k*** элемент ***c*** представлен функцией с сигнатурой ***(String) -> ()***. То-есть, эта фкнкция принимает аргумент типа ***String*** и ничего не возвращает. Строка ***k.c("Hello")*** вызывает эту функцию с передачей ей аргумента.

Кортежи можно применять для инициализации одиночных переменных:

l***et x = ("Peter", 26, "London")***
let (a, b, c) = x
print(a); print(b); print(c) → Peter 26 London

То же самое будет и в случае именованных элементов:

let x = (name: "Peter", age: 26, adress: "London")
let (a, b, c) = x
print(a); print(b); print(c) → Peter 26 London

Ненужные элементы можно заменять на знакозаменитель _:

let x = ("Peter", 26, "London")
let (_, b, _) = x
print(b) → 26

С помощью кортежей легко выполнять взаимный обмен данными между переменными:

var a = 1, b = 2, c = 3, d = 4
(c, a, d, b) = (a, b, c, d)
print(a, b, c, d) → 2 4 1 3

Кстати, для обмена значениями двух переменных любых типов есть встроенная функция ***swap***:

var a: String = "Hello"
var b: String = "World"
swap(&a, &b)
print("a = \(a), b = \(b)") → a = World, b = Hello

(Функция ***swap*** требует ставить перед аргументами знак ***&***, дальше мы узнаем, зачем этот знак нужен).

Если функция должна возвращать несколько значений, то их обычно группируют в кортеже:

```
import Foundation
func f(a: Int) -> (x: Int, y: Double) {
      let x = 2 * a; let y  =  sin(0.9); return(x, y)
}
let t = f(a: 3); print(t)
```
→ (x: 6, y: 0.78332690962748341)

Иногда аргументы функции и возвращаемые ею результаты представлены кортежами сложной формы, например они могут содержать вложенные кортежи:

```
func f(circle: (c: (x: Float, y: Float), r: Float)) -> (c:(x: Float, y: Float), r: Float) {
    return (circle.c, circle.r * 2.0)
}
print(f(circle: (c: (x: 2, y: 3), r: 7)))
```
→ (c: (x: 2.0, y: 3.0), r: 14.0)

Из примера видно, что в таких случаях структуру кортежа приходится повторять не один раз и это делает код программы громоздким. Чтобы упростить код, можно ввести новый тип, представляющий кортеж. Для этого применяется служебное слово ***typealias***. Например:

```
typealias C = (c: (x: Float, y: Float), r: Float)
func f(circle: C) -> C {
    return (circle.c, circle.r * 2.0)
}
print(f(circle: (c: (x: 2, y: 3), r: 7)))
```
→ (c: (x: 2.0, y: 3.0), r: 14.0)

Здесь мы ввели новый тип ***C*** и это избавило нас от необходимости повторять структуру кортежа при описании типа аргумента функции ***f*** и типа возвращаемого результата. Здесь есть ещё одна интересная деталь:

```
typealias T = (name: String, age: Int, address: String)
func f() -> T { return ("Marta", 45, "Paris") }
print(f())
```
→ (name: "Marta", age: 45, address: "Paris")

В качестве типа возвращаемого функцией ***f*** результата указан псевдоним для кортежа ***T***. В операторе ***return*** перечислены только значения, а возвращается кортеж с именами элементов.

9. Структуры (Structs)

Ранее мы уже использовали структуру и немного познакомились с синтаксисом. Рассмотрим структуры теперь подробнее.

Те переменные, что объявляются в теле структуры, называют то членами (members), то свойствами (properties***)***. Приведём пример структуры, у которой есть член, представляющий вложенную структуру:

```
struct Location {
```

```
    var latitude: Double
    var longitude: Double
}
struct Delivery {
    var range: Double
    let center: Location
}
let a = Location(latitude: 44.9871, longitude: -93.2758)
var b = Delivery(range: 200, center: a)
```

Здесь переменная ***a*** представляет конкретный экземпляр структуры ***Location***, а переменная ***b*** – экземпляр структуры ***Delivery***. Как видим, название структуры ***Location*** представляет тип переменной ***center***. Структура ***Delivery*** содержит вложенную структуру ***Location***. В экземпляре ***b*** член ***center*** инициализирован экземпляром ***a***. Для извлечения значений членов структуры применяется точечная нотация:

print(b.range) → 200.0
print(b.center.latitude) → 44.9871

Такой же синтаксис применяется и для присваивания членам новых значений:

```
b.range = 250
```

Структуры могут иметь методы, для создания которых применяется слово ***mutating***. Такие методы могут изменять члены самой (itself) структуры:

```
struct C {
    var x = 0
    mutating func f(a: Int) { x += a }
}
var y = C(x: 5)
y.f(a: 8)
print(y.x)
```

→ 13

Отметим, что экземпляр структуры ***C y*** в этом случае не может быть константой (let).

Методы, не изменяющие членов структуры, объявляются, как обычные функции и не требуют слова ***mutating***:

```
struct C {
    var x: Int
    func f(a: Int) -> Int {return a + 3 + x}
```

```
}
let y = C(x: 5)
let r = y.f(a: 2); print(r) → 10
```

В этом случае ***y*** может быть константой (let).

В отличие от классов структуры не могут наследовать, но они могут иметь дочерние протоколы (protocols), с которыми познакомимся далее. А в общем-то структуры похожи на классы. В частности они, как и классы, могут иметь конструкторы ***init***, позволяют создавать экземпляры структур, производят новые типы данных и так далее.

Приведу ещё один характерный пример сохранения и извлечения данных в структуре:

```
struct Person {
        let name: String
        let birthYear: Int?
}
let persons = [
        Person(name: "Walter White", birthYear: 1959),
        Person(name: "Jesse Pinkman", birthYear: 1984),
        Person(name: "Skyler White", birthYear: 1970),
        Person(name: "Saul Goodman", birthYear: nil)
]
let names = persons.map { $0.name }
print(names) → ["Walter White", "Jesse Pinkman", "Skyler White", "Saul
                Goodman"]
```

10. Множества (Sets)

Множество это неупорядоченная коллекция без повторяющихся элементов. Для объявления множества применяется такой синтаксис:

```
var colors = Set<String>()
```

Объявление может быть с одновременной инициализацией:

```
var colors: Set<String> = ["Red", "Blue", "Green", "Blue"]
print(colors) → ["Red", "Blue", "Green"]
```

Значит, повторяющиеся элементы автоматически удаляются.

Метод ***intersection*** позволяет выбрать из двух множеств одинаковые элементы:

let c1: Set = ["Red", "Blue", "Green"]
let c2: Set = ["Purple", "Orange", "Green"]
let x = c1.intersection(c2)
print(x) → ["Green"]

Метод ***subtract*** выбирает те элементы из множества ***c1***, которые не встречаются в множестве ***c2***:

v***ar c1: Set = ["Red", "Blue", "Green"]***
let c2: Set = ["Purple", "Orange", "Green"]
c1.subtract(c2)
print(c1) → ["Red", "Blue"]

Результат располагается на месте множества ***c1***, поэтому это множество должно объявляться как mutable (var).

Метод ***union*** объединяет два множества в одно:

let x = c1.union(c2)
print(x) → ["Red", "Orange", "Blue", "Purple", "Green"]

Метод insert позволяет добавлять в множество новые члены:

var x: Set = ["Red", "Blue", "Green"]
x.insert("Orange")
print(x) → ["Red", "Orange", "Blue", "Green"]

А метод ***remove*** удаляет члены из множества:

x.remove("Red")
print(x) → ["Orange", "Blue", "Green"]

Метод ***contains*** позволяет узнать, имеется ли в множестве заданный элемент:

v***ar s: Set = ["Red", "Blue", "Green"]***
let b = s.contains("Blue")
print(b) → true

Мы можем создавать множества с элементами нашего собственного типа, но для этого надо согласовать наш тип со специальным типом ***Hashable***. Например, создадим множество с элементами типа структуры:

s***truct S: Hashable {***
    let name: String
    let age: Int
}
let t : Set<S> = [S(name:"Masha", age: 17), S(name:"Kolja", age: 20), S(name:"Nina", age: 19)]

print(t) → [S(name: "Masha", age: 17), S(name: "Kolja", age: 20), S(name: "Nina", age: 19)]

Мы создали множество ***t***, содержащее экземпляры структуры ***S***.

11. Хеши (hash)

Коллекции этого вида называют ещё ассоциативными списками, словарями (dictionary), отображениями (map). В Swift применяется термин ***Dictionary***, но в тексте будем применять название хеш (вообще-то это переводится, как мусор), как наиболее краткое. Конечно, map тоже краткое, но этот термин часто применяется и в других обстоятельствах.

В общем случае хеш объявляется так:

var h : Dictionary<Int, String> = Dictionary<Int, String>()

Хеш ***h*** имеет ключи типа ***Int*** и значения типа ***String*** (не забывайте ставить круглые скобки в конце). Теперь хеш можно инициировать:

h = [1: "Petja", 2: "Vasja", 3: "Kolja"]

print(h) → [2: "Vasja", 3: "Kolja", 1: "Petja"]

Как видим, порядок элементов в хеше не фиксируется.

Можно применять и более краткую форму объявления хеша, есть даже ещё два варианта:

var h = [Int: String]()

Или:

var h: [Int: String] = [:]

Объявление можно совместить с инициализацией:

var h: [Int: String] = [1: "Petja", 2: "Vasja", 3: "Kolja"]

Для извлечения значений по ключу принят тот же синтаксис, что и для массивов:

print(h[2]) → Optional("Vasja")

Однако, возвращается тип Optional, и для его удаления (unwrapping) надо применить, как обычно, восклицательный знак:

print(h[2]!) → Vasja

Изменять элементы хеша также просто:

var h = ["name": "John", "surname": "Doe"]
h["name"] = "Jane"
print(h) → ["surname": "Doe", "name": "Jane"]

Метод ***keys*** извлекает все ключи хеша в формате массива:

let h = ["name" : "Kirit" , "surname" : "Modi"]

let k = h.keys
print(k) → ["surname", "name"]

А метод values извлекает все значения:

let v = h.values
print(v) → ["Modi", "Kirit"]

Значение можно изменить и с помощью функции ***updateValue***:

var h = ["name" : "Kirit" , "surname" : "Modi"]
h.updateValue("Peter", forKey: "name")

Слово ***forKey*** здесь служебное, смысл его понятен.

print(h) → ["surname": "Modi", "name": "Peter"]

По ключам можно выполнять цикл ***for***:

var h: [Int: String] = [1: "Petja", 2: "Vasja", 3: "Kolja"]
for i in h.keys { print(i) } → 3 2 1 (порядок не фиксирован)

Точно также можно выполнять цикл ***for*** по значениям (***values***):

for t in h.values { print(t) } → Vasja Kolja Petja

Или по ключам и значениям одновременно:

for (x, y) in h { print(x, y) } →
2 Vasja
3 Kolja
1 Petja

Хеши могут быть вложенными, например значения могут сами быть хешами:

var h: [String:[Int:String]] = ["Toys":[1:"Car",2:"Truck"],"Interests":
[1:"Science",2:"Math"]]
print(h["Toys"]![2]!) → Truck
print(h["Interests"]![1]!) → Science

Восклицательные знаки здесь всё из-за того же Optional.

Рассмотрим теперь пример, в котором используется ещё один полезный механизм языка Swift:

```
extension Dictionary {
    func f(a: Dictionary) -> Dictionary {
        var b = self
        for (key, value) in a { b[key] = value }
        return b
    }
}
var m1 = [1: "aa", 2: "bb", 3: "cc"]
var m2 = [4: "dd", 2: "ee"]
```

let m = m1.f(a: m2)
print(m) → [2: "ee", 3: "cc", 1: "aa", 4: "dd"]

Здесь мы создали функцию ***f***, способную работать с двумя хешами. Если во втором хеше встречается ключ, такой же, как в первом хеше, то значение в первом хеше заменяется на значение из второго хеша. У нас изменилось значение с ключом ***2***. При этом функция ***f*** вызывается для первого хеша, в точечной нотации, как метод: ***m1.f(a: m2)***. Swift позволяет любую пользовательскую функцию наделить такой возможностью. Для этого надо применить директиву ***extension*** к тому типу, для которого функция вызывается, как метод. В примере это тип ***Dictionary***. Переменная, для которой функция вызывается в точечной нотации, доступна в теле функции под именем ***self***. Рассмотрим этот приём для простейшего случая:

```
extension Int {
    func f(x: Int) -> Int { return self * x }
}
print(2.f(x: 3)) → 6
```

Теперь функция ***f*** работает с целыми числами. Значит, директиву ***extension*** надо применить к типу ***Int***. Тип аргументов функции в круглых скобках и тип результата может быть любым:

```
extension Int {
    func f(x: Double) -> Double { return Double(self) * x }
}
print(2.f(x: 7.82)) → 15.64
```

Напоминаю, что ***Double(self)*** трансформирует целое число ***self*** к типу ***Double***.

12. Переключатель switch - case

Swift имеет традиционный переключатель ***switch — case***, обычно эту операцию называют «сопоставление с образцом» (pattern matched):

```
let x = 3
switch x {
    case 1: print("One!")
    case 2: print("Two!")
    case 3: print("Three!") → Three!
    default: print("Not One, Two or Three")
}
```

Как обычно, если какой-то из ***case*** возвращает ***true***, выполняется выражение после двоеточия и остальные условия не проверяются. Вариант с ***default*** выполняется всегда, если ни один из предшествующих ***case*** не вернул ***true***.

Посмотрим теперь на пример работы с типом:

```
var x: Any? = 77
switch x {
    case nil: print("x is nothing")
    case is String: print("x is a String")
    case _ as Double: print("x is not nil, any value that is a Double")
    case let y as Int where y > 0:
        print("\(y) value is not nil but an int and greater than 0") →
            77 value is not nil but an int and greater than 0
    default: print("something another thing")
}
```

Поскольку тип переменной ***x*** здесь может быть любым, мы объявили тип, как ***Any?***, а знак вопроса указывает, что этот тип ещё и ***Optional***, то-есть, ***x*** может принимать значение ***nil***. В третьем ***case***, продемонстрирована возможность замены сопоставляемой переменной (***x***) на placeholder. В четвертом ***case*** вместо ***x*** использована вновь объявленная переменная ***y***, которая автоматически принимает то же значение, что и ***x***. Кроме того, здесь мы можем видеть, что при сопоставлении можно вводить дополнительное условие (в примере ***y > 0***).

В одном ***case*** можно сопоставлять с образцом сразу несколько данных:

```
let x = "Cat"
switch x {
    case "Cat", "Dog": print("Animal is a house pet.")
    default: print("Animal is not a house pet.")
}
```

Будет выведено: Animal is a house pet.

Можно проверять попадание числа в диапазон:

```
let x = 20
switch x {
    case 0: print("Zero")
    case 1..<10: print("Between 1 and 10")
    case 10..<20: print("Between 10 and 20")
```

```
        case 20..<30: print("Between 20 and 30")
        default: print("Greater than 30 or less than Zero")
}
```

Получим: Between 20 and 30

Применяя placeholder можно выполнять частичное сопоставление. Например, можно сопоставлять кортежи не по всем, а только по выбранным элементам:

```
let p: (x: Int, y: Int, z: Int) = (3, 2, 5)
switch p {
        case (0, 0, 0): print("Origin")
        case (_, 0, 0): print("On the x-axis.")
        case (0, _, 0): print("On the y-axis.")
        case (0, 0, _): print("On the z-axis.")
        default: print("Somewhere in space")
}
```

Получим: Somewhere in space

Подставив вместо координаты placeholder, мы указываем, что значение данной координаты нас не интересует.

Можно игнорировать значение данной координаты при этом сохраняя его для использования. Для этого применяется такой синтаксис:

```
let p: (x: Int, y: Int, z: Int) = (0, 2, 0)
switch p {
        case (0, 0, 0): print("Origin")
        case (let x, 0, 0): print("On the x-axis at x = \(x)")
        case (0, let y, 0): print("On the y-axis at y = \(y)")
        case (0, 0, let z): print("On the z-axis at z = \(z)")
        case (let x, let y, let z): print("Somewhere in space at x = \(x), y = \
(y), z = \(z)")
}
```

Получим в результате: On the y-axis at y = 2

Здесь не нужна строчка ***default:***, так как предыдущие ***case*** исчерпывают все возможные варианты.

При сопоставлениях можно задавать дополнительные условия с использованием ключевого слова ***where***:

```
let t = 20
switch t {
        case 0...9 where t % 2 == 0: print("Cold and even")
```

```
        case 10...29 where t % 2 == 0: print("Warm and even")
        case 30...50 where t % 2 == 0: print("Hot and even")
        default: print("Temperature out of range or odd")
}
```

Получим: Warm and even

Здесь заданная температура проверяется на попадание в данный интервал и одновременно на чётность числа (пример конечно несколько искусственный).

Можно сопоставлять с несколькими образцами одновременно в одном ***case***:

```
let n = 5
switch n {
        case 1, 2: print("One or Two!")
        case 3: print("Three!")
        case 4, 5, 6: print("Four, Five or Six!")
        default: print("Not One, Two, Three, Four, Five or Six")
}
```

Получим: Four, Five or Six!

Посмотрим ещё, как переключатель работает с переменными типа Optional:

```
var s: String? = "hi"
var x: Int? = 5
switch (s, x) {
        case (.some, .some): print("Both have values")
        case (.some, nil): print("String has a value")
        case (nil, .some): print("Int has a value")
        case (nil, nil): print("Neither have values")
}
```

Получим: Both have values

Если, например, переменную ***x*** не инициализировать, то-есть написать:

```
var x: Int?
```

тогда ***x*** будет иметь значение ***nil*** и в результате получим:
String has a value

Значит, в Swift ***.some*** означает некоторое реальное значение (не ***nil)***.

13. Enum

(слово переводится, как перечисление)

Как и во многих современных языках программирования, в Swift есть эта разновидность коллекций, называемая перечисление. Будем дальше для краткости пользоваться английским словом enum. Enum позволяет объединять в некую группу несколько родственных переменных. Посмотрим на примере:

```
enum D {
    case up
    case down
    case left
    case right
}
```

Здесь мы создали enum под названием ***D***, содержащее четыре переменных, которым должно предшествовать слово ***case***. Допускается использовать более краткую форму:

enum D { case up, down, left, right }

Создавая enum, мы получаем новый тип и можем, например, объявить переменную этого типа (тип указывается впереди и отделяется от переменной точкой):

let x = D.up

При этом ***x*** получает значение ***up***:

print(x) → up

То-есть, переменная ***x*** представляет только идентификатор члена enum и в этом, кажется, мало толку. Но, как увидим далее, к переменным в enum можно присоединять ассоциированные переменные, имеющие значения.

В Swift имеется оператор печати ***debugPrint***, применяемый при отладке, который выводит в данном случае полное (квалифицированное) имя переменной:

debugPrint(x) → prog.D.up

Здесь ***prog*** – имя программы.

Можно указать тип как обычно через двоеточие:

let y: D = .up

При этом точка обязательна.

print(x == y) → true

Такой же синтаксис применяется при использовании переменных этого типа в функциях:

```
func f(x: D) {print(x)}
f(x: D.up) → up
```

Если тип может быть выведен автоматически, его можно не указывать, а ставить только точку:

f(x: .up) → up

Переменные этого вида могут быть членами коллекций всех видов:

let m: Array<D> = [.right, .left]
print(m) → [D.right, D.left]

Чаще всего enum используется в переключателях ***switch-case***:

```
let x = D.left
switch x {
        case .up: print(x)
        case .down: print(x)
        case .left: print(x)
        case .right: print(x)
}
```

Получим результат: left.

К элементам enum можно присоединить ассоциированные с ним переменные, их ещё называют payloads — полезная нагрузка. Например:

```
enum A { case x, y, z(d: Float) }
let a = A.z(d: 0.5); print(a) → z(0.5)
switch a {
        case .x: print(A.x)
        case .y: print(A.y)
        case .z(let t): print("d =  \(t)") →d =  0.5
}
```

Здесь у переменной ***A.z*** имеется ассоциированная переменная ***d***. В последнем switch-case показано, как значение ассоциированной переменной можно извлечь. Тут использована локальная переменная ***t***, и, как обычно, для неё можно применить тот же идентификатор (***d***):

case .z(let d): print("d = \(d)")

Тогда локальная ***d*** временно маскирует глобальную ***d***. Допустим тут и такой синтаксис:

case let .z(t): print("d = \(t)")

Для извлечения одиночного значения можно применить вариант ***if-case***:

if case .z(let t) = a { print("d: \(t)") } → d: 0.5

Так же можно использовать и оператор ***guard***:

guard case .z(let t) = a else { print("A is not z"); return }

При удачном сопоставлении эта строка просто пропускается.

Переменные с ассоциированными переменными (payloads) по умолчанию не сопоставимы (не Equatable). Поэтому при необходимости надо создать для них оператор сравнения (==) самостоятельно. Например, можно создать новую функцию (==):

```
func == (p: A, r: A) -> Bool {
    switch p {
        case .x: if case .x = r { return true }
        case .y: if case .y = r { return true }
        case .z(let t): if case .z (let s) = r { return t == s }
    }
return false
}
```

print(A.x == A.y) → false
print(A.z(d: 0.5) == A.z(d: 0.5)) → true
print(A.z(d: 0.5) == A.z(d: 0.7)) → false

Enum очень удобны при работе с деревьями. Не вдаваясь пока в детали, приведём простой пример, чтобы указать на необходимость применения директивы ***indirect*** при рекурсивном обращении к enum:

```
enum D<T> {
    case leaf(T)
    indirect case b(D<T>, D<T>)
}
let t = D.b(.leaf(1), .b(.leaf(2), .leaf(3)))
print(t) → b(D<Int>.leaf(1),D<Int>.b(D<Int>.leaf(2), D<Int>.leaf(3)))
```

Здесь создано дерево ***D***, имеющее один лист ***leaf*** и одну ветвь ***b***, использующую enum ***D*** рекурсивно. В этом случае при создании ветки надо использовать оператор ***indirect***, иначе компилятор зафиксирует ошибку, связанную с бесконечным расходом памяти. Переменная ***t*** представляет конкретный экземпляр дерева.

Оператор ***indirect*** можно указать при объявлении enum, тогда он будет использоваться там, где потребуется:

```
indirect enum D<T> {…}
```

В этом примере буквой ***T*** обозначен произвольный тип (генерик), но о них будем говорить позже.

Enum без payloads может содержать так называемые ***raw values*** — сырые (необработанные) значения при таком синтаксисе:

```
enum R: Int {
    case a = 0
```

```
        case b = 90
        case c = 180
        case d = 270
}
let x = R.c
print(x.rawValue) → 180
```

Значит, сырое значение можно извлечь с помощью встроенного метода ***rawValue***. При объявлении enum надо указывать тип этих raw values через двоеточие. Если объявить тип raw values как ***Int***, то при отсутствии их явного указания они принимают значение ***0*** и дальше увеличиваются на ***1***, и после явного указания тоже:

```
enum E: Int {
        case a
        case b
        case c = 7
        case d
}
let  m: Array<Int> = [E.a.rawValue,  E.b.rawValue, E.c.rawValue,
E.d.rawValue]
print(m) → [0, 1, 7, 8]
```

Если указать тип ***String***, то метод ***rawValue*** сами идентификаторы элементов преобразует в строковые значения:

```
enum Mars: String {
        case phobos
        case deimos
}
let x =  Mars.phobos
let y =  Mars.deimos
let m: Array<String> = [x.rawValue, y.rawValue]
print(m) → ["phobos", "deimos"]
```

Ещё пример применения метода ***rawValue*** в теле функции:

```
enum R: String {
        case up
        case right
        case down
        case left
}
func f(x: R) -> String {
```

```
return x.rawValue
}
print(f(x: R.right) + f(x: R.left))
```
→ rightleft

Есть возможность наоборот извлекать элементы по заданным «сырым» (***rawValue)*** значениям:

```
enum E: Int {
        case a
        case b
        case c = 7
        case d
}
let x = E(rawValue: 1)
let y = E(rawValue: 7)
let z = E(rawValue: 25)
print([x,y,z])
```
→ [Optional(E.b), Optional(E.c), nil]

Если указанное ***rawValue*** не существует, возвращается ***nil.***

В следующем примере показано использование ***rawValue*** в так называемой ***if-let*** конструкции, подробнее о ней в следующем разделе.

```
enum Mars: String {
        case phobos
        case deimos
}
let x = "deimos"
if let y = Mars(rawValue: x) { print("Mars has a moon named \(y)") }
        else { print("Mars doesn't have a moon named \(x)") }
```

Ответ будет: Mars has a moon named deimos, а если принять:

```
let x = "aaaaaa"
```

то получим: Mars doesn't have a moon named aaaaa.

Здесь выражение ***let y = Mars(rawValue: x)***, расположенное после ***if***, даёт ***true***, если ***Mars(rawValue: x)*** не равно ***nil***, и даёт ***false*** в противном случае. К тому же во втором случае переменная ***y*** будет считаться не существующей.

Имеется ещё метод ***hashValue***, который позволяет узнать индекс (начиная с 0) для заданного элемента enum:

```
enum E: Int {
        case a
        case b
        case c = 7
```

```
        case d
}
let z = E(rawValue: 8)
print(z!) → d
print((z?.hashValue)!) → 3
```

Здесь надо не забывать ставить знак вопроса после ***z***.

В enum может быть использована функция ***init?*** для инициализации элементов. Посмотрим на пример её использования и на довольно замысловатый синтаксис:

```
enum C { case north(Int), south(Int), east(Int), west(Int)
        init?(d: Int) {
                switch d {
                        case 0...45: self = .north(d)
                        case 46...135: self = .east(d)
                        case 136...225: self = .south(d)
                        case 226...315: self = .west(d)
                        case 316...360: self = .north(d)
                        default: return nil
                }
        }
}
var x = C(d: 0); print(x as Any) → Optional(C.north(0))
x = C(d: 90); print(x as Any) → Optional(C.east(90))
x = C(d: 500); print(x as Any) → nil
```

Это похоже на вызов функции ***C*** с передачей ей аргумента ***d***.

Enum позволяет применять встроенные enum для более ясной структуризации данных. Например:

```
enum Orchestra {
        enum Strings {case violin, viola, cello, doubleBasse}
        enum Keyboards {case piano, celesta, harp}
        enum Woodwinds {case flute, oboe, clarinet, bassoon}
}
let instrment1 = Orchestra.Strings.viola; print(instrment1) → viola
let instrment2 = Orchestra.Keyboards.piano; print(instrment2) → piano
```

14. Тип Optional

Рассмотрим ранее уже неоднократно используемый нами тип Optional теперь более подробно. Мы уже знаем, что любой тип, базовый или созданный пользователем, может дополнительно быть ещё и Optional и в этом случае переменная этого типа может принимать значение ***nil***. При объявлении такого типа достаточно поставить после указателя типа знак вопроса:

var x: Int?

При этом переменная ***x*** получит значение ***nil*** по умолчанию. Но можно инициировать значением ***nil*** явно:

var x: Int? = nil

Если инициировать ***x*** другим значением, например целым числом:

var x: Int? = 5

то при выводе получим:

print(x) → Optional(5)

Это работает нормально, только при компиляции выдаётся предупреждение, для удаления которого требуется добавлять ***as Any***:

print(x as Any) → Optional(5)

Для удаления Optional надо поставить восклицательный знак:

print(x!) → 5

Эта процедура, обозначенная восклицательным знаком, называется «вскрытие» (unwrapping).

Но что же всё это нам даёт? Один важный фактор уже можно отметить. Для получения числа из строкового представления мы поступаем так:

let x :Int? = Int("42")

print(x as Any) → Optional(42)

Здесь всё нормально, для выхода из Optional достаточно выполнить unwrapping: ***print(x!)*** → 42. Теперь посмотрим, что будет, если текстовая величина не представляет числа:

let x :Int? = Int("hello")

print(x as Any) → nil

Итак, при попытке получения числа из строки, не представляющей число, получаем не исключение, а значение ***nil***. То же самое будет и при работе с другими типами.

При работе с типом Optional надо иметь ввиду, что попытка выполнить unwrapping для значения ***nil*** вызывает исключение:

var x: String? = nil

var y: String = x!

Компиляция пройдёт нормально, но на этапе runtime получим: Unexpectedly found nil while unwrapping an Optional value. Для безопасного unwrapping рекомендуется применять так называемое ***if-let*** выражение:

```
var x: Int?
if let y = x { print("x: \(y)") }
        else { print("x was not assigned a value") }
```

Результат: x was not assigned a value. В таком контексте выражение ***let y = x*** является условным и даёт значение ***false***, если ***x*** равно ***nil***, и ***true*** - в противном случае. Если написать: ***var x: Int? = 5*** то получим x: 5.

Тот же самый результат можно получить с помощью имеющейся у Swift директивы ***guard***:

```
func f(x: Int?) {
        guard let y = x else { print("x was not assigned a value"); return }
        print("x: \(y)")
}
f(x: 5) → x: 5
```

Выражение ***if-let*** позволяет проверять сразу несколько переменных:

```
var x:String? = "Hello"
var y:String? = "World"
if let x1 = x, let y1 = y { print("\(x1) + \(y1)") }
        else { print("x or y was not assigned a value") }
```

Получим: Hello + World

Здесь условное выражение даёт ***true*** только в том случае, когда обе переменные не равны ***nil***. Можно выстраивать целую цепочку проверок:

```
let Name:String? = "Bob"
let x:Bool? = false
if let y = Name, y == "Bob", let z = x, !z {
        print("Name is Bob and x was false!")
}
```

Ответ будет Name is Bob and x was false! Так как все четыре условия: ***y = Name***, ***y == "Bob"***, ***let z = x***, ***!z*** дают true.

Имеется также оператор, обозначаемый двумя знаками вопроса (??), который в документации называется ***nil-coalescing*** (не придумаю, как лучше это перевести). Посмотрим, как он работает на примере:

func f(s: String?) -> String { return s ?? "Good bye!" }
print(f(s: "Hi")) → Hi
print(f(s: nil)) → Good bye!

Таким образом, выражение ***s ?? "Good bye!"*** возвращает значение переменной ***s***, если оно не равно ***nil***, иначе возвращает то, что находится справа от ***??***.

Есть также оператор, обозначаемый одним вопросительным знаком (***?***), в документации назван ***Optional Chaining***. Посмотрим на примере:

```
struct S {
        var t: Int
        func f() { print("Hello World!") }
}
var x: S? = S(t: 77)
x?.f() → Hello World!
var y:S? = nil
y?.f() → метод f не вызван
print(x?.t as Any) → Optional(77)
```

Этот оператор обеспечивает доступ к методам и свойствам (к любым членам) структуры через заданное значение переменной. В примере задана структура ***S***, имеющая свойство ***t*** и метод ***f***. Объявляем и инициируем переменную ***x*** типа ***S*** (обязательно Optional). Теперь можно вызвать метод ***f*** с помощью оператора (***?***):

```
x?.f()
```

Вызов реализуется только в том случае, если переменная ***x*** не равна ***nil***. В примере через переменную ***y***, равную ***nil*** вызвать метод ***f*** нельзя, строка ***y?.f()*** просто пропускается.

В строке ***print(x?.t as Any)*** выводим значение свойства ***t*** с помощью оператора (***?***). Результат всегда будет иметь тип Optional независимо от того, было ли объявлено свойство в структуре как Optional. В примере свойство ***t*** имеет тип ***Int*** (не Optional).

Метод ***f*** в нашем примере возвращает ***Void?***, то-есть при отсутствии результата (только вывод на печать), мы в этой ситуации тоже имеем тип Optional и это можно использовать:

```
let x : S? = S(t:7)
if x?.f() != nil { print("x is non-nil, and f() was called") }
        else { print("x is nil, therefore f() wasn't called") }
```

Результат: Hello World и x is non-nil, and f() was called, так как метод ***f*** вернул ***Void*** (не ***nil***). Но если заменить на ***let x : S? = nil***, то получим: x is nil, therefore f() wasn't called поскольку метод ***f*** теперь не был вызван и выражение ***x?.f()*** вернуло нам значение ***nil.***

Рассмотрим ещё несколько примеров использования полезной конструкции ***if-let***. Выражение ***if let y = x*** выполняет unwrapping и переменную ***y*** можно использовать, например, в арифметических операциях:

```
let x: Int? = 10
if let y = x { print("x was not nil: \(y + 1)") }
    else { print("x was nil") }
```

Получим: x was not nil: 11. При этом можно не применять новый идентификатор, а использовать тот же, то-есть ***x***:

```
let x: Int? = 10
if let x = x { print("x was not nil: \(x + 1)") }
    else { print("x was nil") }
```

Можно выстраивать цепочки проверок, например, так:

```
if let y = x, let t = s {
    Здесь можно что-то делать с использованием y и  t
    } else if let y = x {
    Здесь можно  что-то делать с y
    } else {
    Здесь мы имеем x = nil
}
```

При использовании ***if-let*** можно добавлять через запятую проверку дополнительных условий:

```
let x: Int? = 8
if let y = x, y % 2 == 0 { print("x is non-nil, and it's an even number") }
```

Допустимо применять самые разнообразные комбинации проверок, например:

```
let x: Int? = 5, s: String? = "Hello"
if let x = x, x % 2 == 1,  let s = s,  let fc = s.first,  fc != "x"  {
 print("allbindings & conditions succeeded!") }
```

Получим: allbindings & conditions succeeded!, значит число ***x*** не равно ***nil***, оно нечётное, строка ***s*** не равна ***nil***, и первая буква в этой строке не равна ***"x"***.

Оператор ***guard*** можно использовать для выполнения блока ***else*** с выходом из него с помощью ***return, break,*** или ***continue***. Можно, например, программировать так:

```
func f(x: Int) {
    guard x == 10 else {
        print("x is not 10")
        return
    }
    print("x is 10")
}
```

f(x: 8) → x is not 10

Здесь если условие даёт ***false***, выполняется блок ветви ***else*** с выходом из него с помощью оператора ***return***.

Можно также применять комбинацию ***guard-let-else*** например так:

```
func f(x: Int?) {
    guard let y = x else { print("x does not exist")
        return
    }
    print(y)
}
```

f(x: nil) → x does not exist

f(x: 7) → 7

Здесь ветвь ***else*** будет выполнена, если переменная ***x*** равна ***nil*** (не имеет значения). При этом выполняется ещё и unwrapping, если ***x*** не равна ***nil***.

Как и при использовании ***if-let***, можно через запятую добавлять дополнительные условия:

```
func f(x: Int?) {
    guard let y = x, y == 10 else {
        print("x does not exist or is not 10")
        return
    }
print(y)
}
```

f(x: 8) → x does not exist or is not 10

f(x: 10) → 10

Как видим, тип Optional позволяет создавать и использовать целый ряд замысловатых трюков, наверное они в некоторых случаях могут быть полезными.

15. Функции

Ранее мы уже часто создавали и использовали функции и успели хорошо с ними познакомиться. Теперь рассмотрим ещё некоторые особенности функций в Swift.

При вызове функций требуется указывать идентификаторы аргументов, например:

```
f(x: 5, y: 10)
```

Однако, такие аргументы не считаются именованными, поскольку при вызове функции нельзя изменить порядок их расположения. Но Swift позволяет именованные аргументы, при этом их имена указываются впереди идентификаторов:

```
func f(one x: Int, two y: Int)  {
        print("\(x + y) is x + y")
}
let ten: Int = 10
let five: Int = 5
f(one: ten, two: five)
```
→ 15 is x + y

Тем не менее, и в этом случае должен соблюдаться порядок расположения аргументов, то-есть, нельзя написать такой вызов:

```
f(two: five, one: ten)
```

И тогда не очень ясно, какая от именованных аргументов польза. Для не именованных аргументов можно использовать знак (_), чтобы при вызове не указывать идентификаторы аргументов:

```
func f(_  x: Int, _  y: Int)  {
        print("\(x + y) is x + y")
}
let ten: Int = 10
let five: Int = 5
f(ten, five)
```
→ 15 is x + y

При именованных аргументах так поступать нельзя.

Допускается применять значения аргументов по умолчанию:

```
func f(x: Int = 10, y: Int = 5)  {
        print("\(x + y) is x + y")
```

}
f() → 15 is x + y

Задавать значения таких аргументов при вызове функции не требуется, но значения по умолчанию можно изменить, если надо:
f(y: 7) → 17 is x + y

Для того, чтобы функция могла принимать произвольное число аргументов достаточно после идентификатора аргумента поставить три точки (в документации такой аргумент называется variadic). Тогда этот аргумент в теле функции будет рассматриваться, как массив:

```
func f(x: Int...) -> Int {
        let s = x.reduce(0){a, e in return a + e}
        return s
}
```

let a = f(x: 1, 2); print(a) → 3
let b = f(x: 1,2,3,4,5); print(b) → 15

Для вычисления суммы мы использовали знакомый уже итератор ***reduce***, работающий с массивами. Здесь локальные переменные ***a*** и ***e*** не требуется объявлять заранее, их тип выводится автоматически. Можно, конечно, применить и такой вариант:

```
func f(_ x: Int...) -> Int {
        let s = x.reduce(0){a, e in return a + e}
        return s
}
```

let b = f(1,2,3,4,5); print(b) → 15

В списке аргументов функции перед идентификатором для произвольного числа аргументов могут присутствовать другие аргументы:

```
func f(a: String, b: Float, x: Int...)  {
        print("\(a), \(b), \(x[2])")
}
```

f(a: "Anna", b: 1.034, x: 1,2,3,4,5) → Anna, 1.034, 3

Кстати, если мы используем индекс за пределами списка, например, укажем ***print("\(a), \(b), \(x[7])")***, будет ошибка на этапе runtime.

В Swift функции могут принимать в качестве аргументов другие функции и возвращать функции, как результат:

```
func f() -> ((String, Int) -> String) {
```

```
    func g(name: String, age: Int) -> (String) {
        return "\(name) has been \(age) years"
    }
return g
}
let fi = f()
print(fi("Joe Biden", 80)) → Joe Biden has been 80 years
```

Здесь функция ***f*** имеет такую сигнатуру:

```
() -> ((String, Int) -> String)
```

Значит, функция ***f*** не имеет аргументов и возвращает функцию ***g***, у которой два аргумента типов ***String***, и ***In***t и которая возвращает результат типа ***String***. В строке

```
let fi = f()
```

мы присваиваем полученной функции новое имя ***fi***. Но можно было бы написать и так:

```
print(f()("Joe Biden", 80))
```

Swift позволяет создавать функции, способные модифицировать переменную, передаваемую функции в качестве аргумента. Для этого надо перед указателем типа аргумента поставить слово ***inout***:

```
func f(x: inout Int) { x -= 5 }
var y = 30
print("y = \(y) ") → y = 30
f(x: &y)
print("now y = \(y)") → now y = 25
```

Кроме того, при вызове функции перед передаваемым параметром следует поставить знак ***&***.

Если функция должна возвращать несколько значений, их надо поместить в кортеж:

```
func f(x: Int, y: String) -> (Int, String) {
    let a = x * 2
    let b = String(y.reversed())
    return (a, b)
}
let z = f(x: 5, y: "Hello")
print("a = \(z.0), b = \(z.1)") → a = 10, b = olleH
```

Если аргументом функции является другая функция, применяется следующий синтаксис:

```
import Foundation
```

```
func p(a: Float) -> Float { return sin(a) }
func f(x: Float, g: ((Float) -> Float)) -> Float {
        return g(x)
}
let y = f(x: 0.9, g: p)
print(y)
```
→ 0.783327

Значит, для функции в списке аргументов вместо типа надо указывать сигнатуру функции. Как обычно, перед аргументами можно поставить знак (_):

```
func f(_ x: Float, _ g: ((Float) -> Float))
```

и тогда при вызове не надо указывать идентификаторы аргументов:

```
let y = f(0.9, p)
```

Вместо функции аргументом может быть closure (здесь анонимная функция) и тогда надо программировать, например, так:

```
import Foundation
func f(x: Float, g: ((Float) -> Float)) -> Float {
        return g(x)
}
let y = f(x: 0.9) { t in sin(t) }
print(y)
```
→ 0.783327

Значит, closure выносится за скобки списка передаваемых значений, слово ***in*** служебное, а для параметра можно применить любой идентификатор. Подробнее closure обсудим позже.

Конечно Swift позволяет рекурсию и, значит, есть возможность программирования в функциональном стиле. Приведу наверное самой популярный пример на эту тему — вычисление факториала числа:

```
func f(x: Int) -> Int {
        var fac: Int
        switch x {
                case 1:  fac = 1
                default: fac = x * f(x: x - 1)
        }
        return fac
}
print("factorial 5 = \(f(x: 5))")
```
→ factorial 5 = 120

Здесь мы использовали переключатель ***switch – case***, хотя могут быть и другие варианты.

В следующем примере в качестве аргумента ***g*** метода ***f*** структуры ***S*** использована функция ***fi***:

```
struct S {
	func f(mr: [Int], g: (Int)-> Int)-> [Int] {
		var a: [Int] = []
		for i in mr { a.append(g(i)) }
		return a
	}
}
let m = [1,3,5,7,9,11,8,6,4,2,100]
let p = S()
func fi(x: Int)-> Int { return (x + 2) }
print(p.f(mr: m, g: fi))
```

→ [3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]

Можно, конечно, методу ***f*** прямо передать closure:

```
struct S {
	func f(mr: [Int], g: (Int)-> Int)-> [Int] {
		var a: [Int] = []
		for i in mr { a.append(g(i)) }
		return a
	}

}
let m = [1,3,5,7,9,11,8,6,4,2,100]
let p = S()
print(p.f(mr: m, g: {(i: Int)-> Int in return i + 2}))
```

16. **Extension** (переводится как расширение)

Extension применяются для различных целей. Например они могут создавать методы, вызываемые в точечной нотации. Наверное чаще всего extension применяется для изменения (расширения) функциональности существующих методов. Снова рассмотрим задачу вычисления факториала. С помощью extension это можно сделать, например, так:

```
extension Int {
	func factorial() -> Int {
		return (1...self).reduce(1){ a, e in return a * e }
```

```
    }
}
```
print(5.factorial()) → 120

Значит надо начать со служебного слова ***extension*** и указать тип аргумента, для которого вызывается метод. В теле extension можно создать функцию, а параметр, для которого она вызывается, обозначить зарезервированным словом ***self***. Для вычисления факториала здесь использован ранг и итератор ***reduc***e.

У функции не указываются аргумент, а ставятся пустые скобки, поскольку использовано слово ***self***. В данном варианте и при вызове метода надо ставить пустые скобки.

Ранее указывалось, что Swift не имеет возможности извлекать знаки из текста по индексу. Эту возможность можно получить с помощью extension:

```
extension String {
    subscript(n: Int) -> Character {
        let i = self.index(self.startIndex, offsetBy: n)
        return self[i]
    }
}
var s = "StackOverFlow"
```
print(s[2]) → a
print(s[3]) → c

Здесь используется ***subscript***, о котором будем говорить далее, и встроенная функция ***index***, работу которой можно понять из самих названий её аргументов. Таким образом, здесь с помощью ***extension*** мы расширили способности типа ***String***.

Отметим ещё, что переменную ***self*** в теле ***extension*** можно изменять, используя функцию с модификатором ***mutating***:

```
extension Bool {
    mutating func f() -> Bool {
        self = !self
        return self
    }
}
var x: Bool = true
```
print(x.f()) → false

17. Классы

Swift относится к объектно-ориентированным языкам программирования и позволяет создавать классы. При этом техника работы с классами и объектами предельно проста и не содержит ничего оригинального.

```
import Foundation
class A {
        var a: Int = 77
        func f(x:  Float, y: Float) -> Float {
                return(pow(x, 2.0) + pow(y, 2.0))
        }
}
let p = A()
print(p.a) → 77
p.a = 32
let r =  p.f(x: 3.0, y: 4.0)
print(r) → 25.0
```

Наш класс ***A*** имеет одну переменную экземпляра класса ***a*** (или свойство, по стандартной терминологии) и один метод экземпляра класса ***f***. Переменная, объявленная со словом ***var*** доступна по чтению и по записи. Методы создаются, как обыкновенные функции. В примере создан экземпляр класса ***p***. Для обращения к переменным и методам применяется точечная нотация. Класс может содержать только переменные (свойства) и методы, и ничего более.

Классы представляют ссылочный тип, то-есть, несколько переменных могут ссылаться на один и тот же экземпляр класса:

```
class A {
        var x = ""
}
let p1 = A()
let p2 = p1
p1.x = "Anna"
p2.x = "Marta"
print(p1.x) →  Marta
```

Как видим, изменение ***x*** в экземпляре ***p2*** изменило значение ***x*** в экземпляре ***p1***. Но если создать:

```
let p1 = A()
```

let p2 = A()
то это будут разные экземпляры класса, ничем не связанные друг с другом:
p1.x = "Anna"
p2.x = "Marta"
print(p1.x) → Anna

Функция ***init*** служит конструктором класса, она позволяет передавать экземпляру класса параметры:

```
class Dog {
        var name: String
        let age: Int
        init(name: String, age: Int) {
                self.name = name
                self.age = age
        }
}
let a = Dog(name: "Rover", age: 5)
var b = Dog(name: "Spot", age: 7)
a.name = "Fido"
//a.age = 6 — здесь будет ошибка, так как свойство age - константа
//a = Dog(name: "Fido", age: 6) — ошибка, экземпляр a - константа
b.name = "Ace"
//b.age = 8 — ошибка, свойство age - константа
b = Dog(name: "Ace", age: 8)
print("\(a.name), \(a.age)") → Fido, 5
print("\(b.name), \(b.age)") → Ace, 8
```

Функция ***init*** ссылается на создаваемый экземпляр класса посредством служебного слова ***self***.

Классы в Swift могут иметь и деструкторы, роль которых выполняет встроенная функция ***deinit***:

```
class C {
        var x: Double
        init(x: Double) { self.x = x }
        func f() { print(x) }
        deinit { print("Goodbye") }
}
let p = C(x: 2.55)
p.f() → 2.55
```

do {
let q = C(x: 3.77)
q.f() → 3.77
} → Goodbye

Функция deinit должна иметь хотя бы пустой блок, или в этом блоке можно вывести какой-нибудь текст, который будет сигнализировать о том, что деструктор выполнился. Деструктор работает только для тех экземпляров класса, которые создаются внутри исполняемых блоков или в теле функций. Вообще-то Swift сам автоматически ликвидирует ненужные более объекты и применять ***deinit*** имеет смысл только в том случае, когда программист желает сам контролировать эту работу.

Для проверки экземпляров класса на равенство применяется операция (===), а для проверки на эквивалентность модифицируем операцию (==):

class A {
let x: String
init(x: String) { self.x = x} }
}
func ==(a: A, b: A) -> Bool { return a.x == b.x }
let p1 = A(x: "Hello")
let p2 = A(x: "Hello")
print(p1 === p2) → ***false*** - (p1 и p2 разные экземпляры класса A)
print(p1 == p2) → true - (p1 и p2 эквивалентны)

Наследование классов синтаксически обозначается двоеточием:

class A {…}
class B: A {..}

Здесь класс ***A*** является родительским (superclass), а класс ***B*** – дочерним (subclass). Множественное наследование в Swift не поддерживается. Частично это компенсируется применением так называемых протоколов, которые рассмотрим отдельно.

Если в классе отсутствует конструктор ***init***, то он всё-равно имеется неявно. Его всегда можно сделать явным, написав пустой конструктор без параметров:

init() {}

Практически это ничего не изменит. Но такой конструктор будет иметь значение, если его применить с модификатором ***private***:

private init() {}

В этом случае функция ***init*** становится недоступной за пределами класса и это не позволяет создавать экземпляры класса. Такие классы иногда бывают полезны и мы ещё поговорим о них далее.

Дополнительно к конструктору ***init*** Swift позволяет использовать эту функцию с модификатором ***convenience***. Посмотрим на примере:

```
class Foo {
    var x: String, y: Int, z: Bool
    init(x: String, y: Int, z: Bool) { self.x = x; self.y = y; self.z = z }
    convenience init() { self.init(s: "Hi") }
    convenience init(s: String) { self.init(x: s, y: 0, z: false) }
}
class Baz: Foo {
    var f: Float
    init(f: Float) { self.f = f; super.init(x: "Hi", y: 0, z: false) }
    convenience init() { self.init(f: 7) }
}
let c = Foo(x: "Hello", y: 10, z: true)
let a = Foo(); print(a.x, a.y, a.z) → Hi 0 false
let b = Foo(s: "Hello"); print(b.x, b.y, b.z) → Hello 0 false
let d = Baz(f: 3); print(d.x, d.y, d.z, d.f) → Hi 0 false 3.0
let e = Baz(); print(e.x, e.y, e.z, e.f) → Hi 0 false 7.0
```

При этом экземпляры ***a*** и ***b*** класса ***Foo*** не имеют переменной экземпляра ***s***.

Для полноты картины в примере дочерний класс ***Baz*** наследует родительскому классу ***Foo***. Я думаю, тут не нужны какие-то дополнительные комментарии, поскольку разобраться с тем, что тут происходит нетрудно.

В классе или в структуре может быть несколько конструкторов ***init***. При создании экземпляра нужный конструктор выбирается по именам параметров:

```
struct S {
    var x: Double
    init(a: Double) { x = a / 100 }
    init(b: Double) { x = b * 1000 }
}
let s1 = S(a: 42)
print(s1.x) → 0.42
let s2 = S(b: 42)
```

print(s2.x) → 42000.0

Здесь по имени параметра ***a*** был выбран первый ***init***, а по имени параметра ***b*** – второй. При нескольких конструкторах нельзя опускать имя параметра при создании экземпляра, то-есть нельзя написать:

let s1 = S(42)

Конструктор позволяет использовать локальную переменную, в частности, предыдущий пример можно изменить так:

```
struct S {
        var x: Double
        init(a y: Double) { x = y / 100 }
        init(b z: Double) { x = z * 1000 }
}
let s1 = S(a: 42)
print(s1.x) → 0.42
let s2 = S(b: 42)
print(s2.x) → 42000.0
```

Переменные ***y*** и ***z*** локальные, они записываются после параметра через пробел и принимают значение, передаваемое параметру при создании экземпляра. При единственном конструкторе с локальной переменной можно название параметра заменить на (_) и тогда не надо указывать имя параметра при создании экземпляра:

```
struct S {
        var x: Double
        init(_ y: Double) { x = y / 100 }
}
let s1 = S(42)
print(s1.x) → 0.42
```

Swift имеет встроенный класс ***Mirror*** с параметром ***reflecting***, позволяющий создавать так называемые отображения (Mirror):

```
class A { var x: String = ""; var y: Int = 0; var z: String? }
let p = A()
p.x = "Hello"; p.y = 199; p.z = "test"
let pm = Mirror(reflecting: p)
print(pm) → Mirror for A
let t = pm.children
print(t.count) → 3
print(t.first?.label as Any) → Optional("x")
print("\(t.first!.value)\n") → Hello
```

for i in t { print("\(i.label!):\(i.value)") } → x:Hello y:199 z:Optional("test")

Здесь мы создали экземпляр ***p*** класса ***A*** и затем создали экземпляр ***pm*** класса ***Mirror***, передав ему ***p*** в качестве параметра. При выводе ***pm*** на печать получаем текст Mirror for A. С помощью встроенного метода ***children*** превращаем ***pm*** в кортеж ***t***, содержащий все переменные экземпляра класса ***A*** (свойства) в виде именованных элементов.

18. Генерики (*Generics)*

Иногда генерики называют параметрами типа, что более точно и понятно, поскольку генерики позволяют работать с переменными произвольного типа.

```
var x: Character?
func f<T>( a:T,  b:T) -> T { return x == "h" ? a : b }
x = "w"
print(f(a: 3,  b: 5)) → 5
x = "h"
print(f(a: false, b: true)) → false
```

Здесь обозначение ***f<T>*** означает, что объявляемая функция ***f*** может принимать аргументы типа ***T*** и буквой ***T*** обозначен произвольный тип. Затем функции ***f*** передаются сначала аргументы типа ***Int***, а затем аргументы типа ***Bool***. По традиции для параметров типа обычно используют букву ***T***, но это необязательно и всё будет работать, если, например, написать:

```
func f<R>( a:R,  b:R) -> R { return x == "h" ? a : b }
```

Можно также применять и набор символов, например ***f<Elem>***. Подобным же образом генерики могут использоваться с классами, структурами или enums:

```
class A<T> {
        var x : T
        init(x:T) { self.x = x }
}
let p = A(x: 2.75); print(p.x) → 2.75
let r = A(x: "Hello"); print(r.x) → Hello
```

Здесь конструктор класса принимает параметры любого типа и выводит их тип автоматически. Можно также задать тип аргументов класса явно:

let q = A<Bool>(x: true); print(q.x) → true

При создании экземпляра класса тип аргументов фиксируется и в данном экземпляре класса уже не может быть изменён. Это справедливо и для экземпляра класса, передаваемого функции, как аргумент:

```
func f(p: A<Double>) {
        print(p.x)
}
```

f(p: A(x: 3.12)) → 3.12

Если мы попытаемся вызвать:

f(p: A(x: "Hello")) - будет ошибка.

В следующем примере покажем применение генерика при использовании протокола ***Equatable*** (о протоколах позже):

```
class A<T: Equatable>{
        var x: T
        init(x: T) { self.x = x }
        func f() -> T { return self.x }
        func g(y: T) -> Bool { return self.x == y }
}
let a = A<Double>(x: 2.71828); print(a.f())  →  2.71828
let b = A<String>(x: "Hello"); print(b.f())  → Hello
//let c = A<[Int]>(x: [2])  — ошибка, [2] не согласуется с Equatable
let d = A<Int>(x: 72)
print(d.g(y: 72))  →  true
print(d.g(y: -274))  →  false
//print(d.g(y: "My String"))  — тип String не конвертируется к Int
```

Здесь ***class A<T: Equatable>*** означает, что объявлен класс ***A***, а параметр типа ***T*** использован в протоколе ***Equatable***, обеспечивающем возможность сопоставления переменных разных типов. Протокол ***Equatable*** встроен в дистрибутив и нам не надо его создавать. Метод ***f*** возвращает значение свойства ***x***, а метод ***g*** выполняет проверку на равенство своего аргумента с ***x***.

Часто бывает полезно создать в классе специальные методы доступа к переменным экземпляра (свойствам). В примере метод доступа по чтению назван ***getValue***, а по записи - ***setValue***:

```
class A<T>{
        var x: T
        init(x: T) { self.x = x }
        func getValue() -> T { return self.x }
        func setValue(x: T) { self.x = x }
}
let a = A<String>(x: "My String Value")
let b = A<Int>(x: 42)
print(a.getValue()) → "My String Value"
print(b.getValue()) → 42
a.setValue(x: "Another String")
b.setValue(x: 1024)
print(a.getValue()) → "Another String"
print(b.getValue()) → 1024
```

В одном генерике можно применять сразу несколько параметров типа, например, так:

```
class A<T1, T2, T3>{
        var v1: T1, v2: T2, v3: T3
        init(v1: T1, v2: T2, v3: T3) { self.v1 = v1; self.v2 = v2; self.v3 = v3 }
        func getV1() -> T1{return self.v1}
        func getV2() -> T2{return self.v2}
        func getV3() -> T3{return self.v3}
}
let p = A<String, Int, Double>(v1: "Value of pi", v2: 3, v3: 3.14159)
print(p.getV1() is String) → true
print(p.getV2() is Int) → true
print(p.getV3() is Double) → true
print(p.getV2() is String) → false
print(p.getV1()) → "Value of pi"
print(p.getV2()) → 3
print(p.getV3()) → 3.14159
```

Рассмотрим ещё пример, в котором с помощью extension создан метод ***remove***, позволяющий удалять любой элемент массива по индексу:

```
extension Array where Element: Equatable {
        mutating func remove(_ e: Element) {
                if let index = self.index(of: e ) { self.remove(at: index)}
```

```
                else { fatalError("Removal error, no such element:\"\
(e)\" in array.\n") }
        }
}
var m = [1,2,3]
print(m) → [1, 2, 3]
m.remove(2)
print(m) → [1, 3]
```

Здесь снова использован протокол ***Equatable*** с параметром типа ***Element***. Слова ***index***, ***of*** и ***at*** служебные.

Swift позволяет добавлять функциональность для базовых операторов. Например, можно операторы сложения (+) и (+=) наделить способностью объединять два хеша в один:

```
func +<K, V>(h1: [K : V], h2: [K : V]) -> [K : V] {
        var c = h1
        for (key, value) in h2 {
                c[key] = value
        }
        return c
}
func += <K, V>(h1: inout [K : V], h2: [K : V]) {
        for (key, value) in h2 {
                h1[key] = value
        }
}
let h1 = ["hello" : "world"]
let h2 = ["world" : "hello"]
var h3 = h1 + h2
print(h3) → ["hello": "world", "world": "hello"]
h3 += ["hello":"bar", "baz":"qux"]
print(h3) → ["hello": "bar", "baz": "qux", "world": "hello"]
```

Использование параметров типа позволяет работать с элементами хешей любых типов. Функция (+=) имеет возможность изменять свой аргумент, что достигается применением модификатора ***inout***. Интересно, что здесь для хеша не надо применять знак ***&*** при передаче аргумента функции.

19. Closures (замыкания)

Чаще closure называют анонимной функцией, в Swift это почти то же самое, что блок, поэтому дальше термины closure и блок часто будем использовать на равных правах. Блоком можно инициализировать переменную, которую затем можно использовать, как функцию:

let c = { print("Hello") }
c() → Hello

Тип этой переменной (closure) ***c*** будет ***() -> Void*** или ***() -> ()***. Применительно к функциям это обычно называют сигнатурой. Значит, чтобы заставить блок выполниться, надо инициировать этим блоком переменную, а затем эту переменную вызвать, как функцию. Есть также и второй способ — чтобы блок выполнить, достаточно поставить перед ним оператор ***do***:

```
do {
    let x = "Hello"
    print(x)
}
```

Переменная ***c*** может использоваться, как всякая другая переменная, например, она может быть аргументом функции:

func f(x: () -> Void) { x() }
f(x: c) → Hello

Но можно, конечно, и не использовать промежуточную переменную ***c***, а передать функции closure непосредственно:

func f(x: () -> Void) { x() }
f(x: { print("Hello") }) → Hello

Ранее мы уже рассматривали примеры, в которых closure передаётся итераторам, например, итератору ***map***:

let s = (1...10).map({ $0 * $0 })
print(s) → [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Closure может принимать и аргументы, как обычная функция. При этом применяется такой синтаксис:

let c = { (x: Int, y: Int) -> Int in return x + y }
let x = c(3, 5)
print(x) → 8

Здесь closure ***c*** имеет такой тип: ***(Int, Int) -> Int***

Функции могут возвращать closure, как результат:

```
import Foundation
func f() -> ((Double) -> Double) {
        let c = { (x: Double) -> Double in return sin(x) }
        return c
}
let g = f()
print(g(0.5))
```
→ 0.479425538604203

Или опуская промежуточные переменные:

```
import Foundation
func f() -> ((Double) -> Double) {
        return { (x: Double) -> Double in return sin(x) }
}
print(f()(0.5))
```
→ 0.479425538604203

Сигнатура функции ***f***: ***() -> ((Double) -> Double)***

Для аргументов closure можно также воспользоваться зарезервированными переменными ***$0***, ***$1***, и так далее, которые мы уже применяли ранее:

```
import Foundation
func f() -> ((Double, Double) -> Double) {
        return { sin($0 + $1) }
}
print(f()(0.5, 0.4))
```
→ 0.783326909627483

Чтобы не повторять сигнатуру несколько раз можно использовать псевдоним (alias), для чего существует директива ***typealias***:

```
typealias t = (Int, Int) -> Int
func f(x: Int, y: Int, g: t) -> Int { return g(x, y) }
let cl: t = { return $0 + $1 }
print(f(x: 5, y: 3, g: cl))
```
→ 8

Создав псевдоним ***t*** для сигнатуры, мы избавились от необходимости записывать её дважды, а именно, при описании функции ***f*** и при описании closure ***cl***.

20. Протоколы (protocols)

Внешне протоколы выглядят, как классы: название протокола и тело в фигурных скобках. Как и в классах, в протоколах могут быть объявлены свойства (переменные экземпляра) и методы, но свойства

не могут быть инициализированы, а методы не могут иметь тела. Всё ограничивается только тем, что свойствам присвоены типы, а методам сигнатуры. Дополнительно для свойств может быть указана доступность — слова get и set в фигурных скобках. Свойство с ***get*** доступно только по чтению, а с ***get*** и ***set*** - по чтению и записи. На протокол может ссылаться класс при том же синтаксисе, что и при наследовании классов — двоеточие перед именем протокола. Например:

```
protocol Pr {
    var x: String { get }
    var y: Int {get set}
}
class Person : Pr {
    var x: String
    var y: Int
    init(x: String, y: Int){ self.x = x; self.y = y }
}
let p = Person(x: "Anna", y: 25)
print(p.x) → Anna
print(p.y) → 25
```

В классе ***Person*** обязательно должны присутствовать свойства ***x*** и ***y*** с теми же типами, что в протоколе ***Pr***, иначе компилятор зафиксирует ошибку. В документации те свойства и методы класса, которые должны удовлетворять требованиям протокола, называют требуемыми свойствами и методами. В экземпляре класса свойству ***y*** можно присвоить новое значение, например ***p.y = 17***, а свойству ***x*** нет. В остальном класс ***Person*** ничем не отличается от обычного. Таким образом, протоколы не добавляют никакой функциональности классам, а только накладывают на них некоторые дополнительные требования. В частности, если мы в нашем примере удалим в классе ссылку на протокол, ничего не изменится, кроме только того, что свойству ***x*** можно будет присваивать новые значения.

Протоколы можно использовать, как типы переменных и функций, например:

```
protocol R { func ran() -> Double }
class L: R {
        var l = 42.0
        let m = 139968.0
```

```
        let a = 3877.0
        let c = 29573.0
        func ran() -> Double {
            l = ((l * a + c).truncatingRemainder(dividingBy: m))
            return l / m
        }
}
class D {
        let s: Double
        let g: R
        init(s: Double, g: R) {
            self.s = s
            self.g = g
        }
        func f() -> Int {
            return Int(g.ran() * s) + 1
        }
}
let p = D(s: 6.0, g: L())
var m: [Int] = []
for _ in 1...20 { m.append(p.f()) }
print(m) → [3, 5, 4, 5, 4, 1, 4, 2, 1, 4, 2, 5, 3, 2, 4, 2, 6, 3, 3, 6]
```

В классе ***L*** реализована (довольно примитивная) функция ***ran***, генерирующая псевдослучайное число. Уже знакомый нам встроенный оператор ***truncatingRemainder*** возвращает остаток от деления двух чисел с плавающей точкой.

Классу ***L*** присвоен тип протокола ***R*** и функция ***ran*** согласуется с требованием протокола. В классе ***D*** объявлена переменная ***g*** с типом ***R***, поэтому конструктор класса ***D*** может принимать в качестве параметра экземпляр ***g*** класса ***L***. Метод ***f*** класса ***D*** с помощью функции ***ran*** генерирует псевдослучайные числа из заданного диапазона. Отметим ещё, что в цикле ***for*** параметр цикла не используется, поэтому мы его заменили на (_) для того, чтобы компилятор не выдавал замечание.

Пример программы моделирует бросание игральной кости, в напечатанном массиве получены результаты двадцати бросков кости с шестью гранями.

И снова мы должны отметить, что протокол ***R*** ничего не даёт, кроме требования к функции ***ran*** иметь заданную сигнатуру. Мы можем удалить протокол и ссылку на него и получим тот же результат:

```
class L {
	var l = 42.0
	let m = 139968.0
	let a = 3877.0
	let c = 29573.0
	func ran() -> Double {
		l = ((l * a + c).truncatingRemainder(dividingBy: m))
		return l / m
	}
}
let q = L()
class D {
	let s: Double
	init(s: Double) {
		self.s = s
	}
	func f() -> Int {
		return Int(q.ran() * s) + 1
	}
}
let p = D(s: 6.0)
var m: [Int] = []
for _ in 1...20 { m.append(p.f()) }
print(m) → [3, 5, 4, 5, 4, 1, 4, 2, 1, 4, 2, 5, 3, 2, 4, 2, 6, 3, 3, 6]
```

С одним протоколом могут быть согласованы несколько классов и тогда все они позволяют создавать экземпляры одного типа, определяемого этим протоколом:

```
protocol Pr {
	var f: Double { get }
}
class C1: Pr {
	let pi = 3.1415927
	var r: Double
	var f: Double { return pi * r * r }
```

```
        init(r: Double) { self.r = r }
}
class C2: Pr {
        var f: Double
        init(f: Double) { self.f = f }
}
class C3 {
        var a: Int
        init(a: Int) { self.a = a }
}
let m: [Any] = [
        C1(r: 2.0),
        C2(f: 243_610),
        C3(a: 4)
]
for p in m {
        if let x = p as? Pr { print("S = \(x.f)") }
        else { print("There is no") }
}
```

Получим такой результат:

```
S = 12.5663708
S = 243610.0
There is no
```

Классы ***C1*** и ***C2*** имеют тип ***Tr***, при этом свойство ***f*** в классе ***C2*** является параметром экземпляра, а в классе ***C1*** — closure. Класс ***C3*** не соответствует протоколу ***Pr***. Массив ***m*** содержит три экземпляра классов ***C1***, ***C2*** и ***C3***. В цикле ***for*** с помощью директивы ***as?*** проверяем, принадлежит ли экземпляр класса типу ***Tr*** и если да, то выводим на печать содержимое свойства ***f***.

Как видим, это уже намного интереснее: протокол позволяет создать тип и классы, привязанные к протоколу, могут создавать объекты одного и того же типа, но с разными возможностями.

Методам протокола можно придать функциональность с помощью extensions:

```
protocol Pr { func f() -> Void }
class C: Pr { func f() {}}
extension Pr {
        func f(x: String) { print(x) }
```

```
}
let p = C()
p.f(x: "Hello") → Hello
```

Здесь метод ***f*** класса C получил функциональность через протокол ***Pr***.

Протоколы можно применять не только с классами, но также и со структурами и с enums. При этом порядок использования протоколов и синтаксис практически тот же. В следующем примере применим протокол для структуры используя ***extension***:

```
protocol Pr {
    func f() -> String
}
struct S {
    var name: String
    func f() -> String { return "Name: \(name)" }
}
extension S: Pr {}
let x = S(name: "Vanja")
let p: Pr = x
print(p.f()) → name: Vanja
```

Здесь структура ***S*** первоначально не связана с протоколом, хотя удовлетворяет его требованию в отношении метода ***f***. Затем с помощью ***extension*** структура ***S*** связывается с протоколом ***Pr*** и, соответственно, приобретает тип ***Pr***.

21. Subscript

Swift имеет некий инструмент, называемый в документации словом ***subscript***, которое почему-то переводят, как список индексов. Этот subscript может применяться в структурах, в классах, в enums, в протоколах. Внешне subscript похож на функцию и в качестве аргументов принимает индексы, используемые при работе с различными коллекциями или последовательностями. Сначала посмотрим на применение subscript в самом простом примере:

```
struct D {
    var m = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
    subscript(i: Int) -> String {
        get { return m[i]}
        set {m[i] = newValue}
```

```
    }
}
var w = D()
print(w[1]) → Mon
print(w[0]) → Sun
w[0] = "Sunday"
print(w) → D(m: ["Sunday", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"])
print(w.m) → ["Sunday", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
```

Итак, в нашем примере ***subscript*** используется для работы с массивом ***m***, элементы которого представлены названиями дней недели. Subscript принимает индекс и возвращает строку. В своём теле subscript содержит уже знакомые нам методы ***get*** и ***set*** с приписанными к ним блоками, в которых можно запрограммировать некоторые действия. В нашем случае эти действия предельно простые: метод ***get*** просто возвращает элемент массива по индексу, а метод ***set*** устанавливает новое значение для элемента массива. Мы уже знакомы с тем, что новое значение доступно под именем ***newValue***.

Весь фокус в том, что при создании экземпляра структуры ***w*** методы ***get*** и ***set*** автоматически доступны без явного вызова этого самого subscript и, таким образом, сам экземпляр структуры ***w*** становится как бы массивом: при распечатке его выводится имя структуры ***D***, название исходного массива ***m*** и его элементы. Сам массив ***m*** доступен, как свойство экземпляра, то-есть, как ***w.m***.

В общем-то рассмотренный нами пример ничего полезного не содержит, но посмотрим теперь на более содержательный случай. Как уже отмечалось, многомерные массивы в Swift создаются, как вложенные одномерные массивы и извлечение элемента, например, из двумерного массива выглядит примерно так: ***m[2][1]***. С помощью subscript можно создать некую имитацию для применения привычного синтаксиса: ***m[2, 1]***:

```
struct M {
    let rs: Int, cs: Int; var g: [Double]
    init(rs: Int, cs: Int) {self.rs = rs; self.cs = cs; g = Array(repeating:
0.0, count: rs * cs)}
    func f(r: Int, c: Int) ->Bool {return r >= 0 && r < rs && c >= 0
&& c < cs}
    subscript(r: Int, c: Int) -> Double {
```

```
        get {assert(f(r: r, c: c), "Index is out of range"); return g[(r * cs) +
c]}
        set {assert(f(r: r, c: c), "Index is out of range"); return g[(r * cs) +
c]             = newValue}
        }
}
var m = M(rs: 2, cs: 2)
m[0, 1] = 1.5
m[1, 0] = 3.2
print(m) → M(rs: 2, cs: 2, g: [0.0, 1.5, 3.2, 0.0])
m[0,3] = 9.0 → Assertion failed: Index is out of range
```

Структура ***M*** содержит: переменные ***rs*** и ***cs***, определяющие размеры двумерного массива, одномерный массив ***g***, содержащий все элементы двумерного массива, конструктор ***init***, который ещё и заполняет массив ***g*** нулями, метод ***f***, позволяющий контролировать попадание текущих индексов в допустимые пределы и сам subscript. Subscript принимает индексы элемента: ***r*** – номер строки и ***c*** – номер столбца и содержит знакомые нам методы ***get*** и ***set*** со своими блоками в которых с помощью директивы ***assert***, принимающей метод ***f***, контролируется попадание индексов в допустимые пределы. Кроме того, метод ***get*** возвращает элемент одномерного массива ***g***, индекс которого вычисляется через ***r*** и ***c***, а метод ***set*** позволяет устанавливать новое значение элементов массива ***g*** используя зарезервированную переменную ***newValue***. Таким образом, создаваемый экземпляр ***m*** структуры ***M***, имитирует двумерный массив элементы которого можно извлекать так, как мы хотели, например, ***m[0,1]***. Легко преобразовать нашу программу и для трёхмерных массивов.

Subscript применяют обычно при работе с массивами фактически только потому, что аргументы, принимаемые subscript(ом) должны записываться в квадратных скобках, как это принято для индексов в массивах. На самом деле это могут быть не обязательно индексы, например:

```
struct T{
        let x: Int
        subscript(i: Int) -> Int { return x * i }
}
let m = T(x: 3)
print("\(m[6])") → 18
```

Формально здесь можно считать, что ***m[6]*** это шестой элемент массива, представляющего таблицу умножения на ***3***. Хотя на самом деле, если распечатать ***m***, то получим:

print(m) → T(x: 3)

То-есть, никакой это не массив.

Посмотрим теперь на пример работы subscript с хешем:

```
struct F {
    enum M { case Breakfast, Lunch, Dinner }
    var m: [M: String] = [:]
    subscript (t: M) -> String? {
        get { return m[t] }
        set { m[t] = newValue }
    }
}
var d = F()
var c = F.M.self
d[c.Breakfast] = "Scrambled Eggs"
d[.Lunch] = "Rice"
print("I had \(d[.Breakfast]!) for breakfast") →
```

I had Scrambled Eggs for breakfast

В строке ***var m: [M: String] = [:]*** создаём пустой хеш ключи в котором представлены экземплярами ***enum M***, а значения — имеют тип ***String Optional***. Вместо индексов subscript принимает теперь объекты ***enum M***. Соответственно, экземпляр структуры ***d*** имитирует хеш и в квадратных скобках теперь будут ключи этого хеша. Поскольку наш enum не имеет конструктора ***init***, то экземпляр допускается создавать с использованием ключевого слова self: ***var c = F.M.self***

Строка ***d[c.Breakfast] = "Scrambled Eggs"*** инициирует элемент хеша. В том случае, когда тип может быть выведен из контекста, Swift позволяет опускать имя экземпляра enum, оставляя только точку, поэтому явно экземпляр ***c*** можно было не создавать. В строке ***d[.Lunch] = "Rice"*** и в операторе ***print*** использован этот синтаксический сахар. При выводе выполняется unwrapping с помощью восклицательного знака.

22. Модификаторы

Модификатор ***static*** позволяет создавать переменные и методы, принадлежащие самому классу, а не его экземплярам. В более ранних языках программирования, например в Ruby, их обычно называют переменными и методами класса в отличие от переменных и методов экземпляра класса. Например:

```
class A {
    var x: Int = 1
    static var y: Int = 2
}
let p = A()
p.x = 3
A.y = 4
let q = A()
print(q.x, A.y) → 1 4
```

По традиционной терминологии здесь ***x*** – переменная экземпляра, а ***y*** – переменная класса. При создании нового экземпляра класса ***q*** переменная ***x*** восстановила своё исходное значение, равное ***1***. Переменная класса ***y*** доступна по имени самого класса — ***A.y*** и при создании нового экземпляра класса она сохраняет своё новое значение, равное ***4***. В экземплярах класса переменная класса ***y*** не доступна, то-есть, нельзя, например, вызвать:

```
print(p.y)
```

Переменную класса можно использовать, например, для подсчёта числа созданных экземпляров класса, например, так:

```
class A {
    static var x: Int = 0
}
let p1 = A(); A.x += 1
print(A.x) → 1
let p2 = A(); A.x += 1
print(A.x) → 2
let p3 = A(); A.x += 1
print(A.x) → 3
```

В классах вместо модификатора ***static*** можно использовать слово ***class*** на тех же правах, и только такие статические переменные и методы могут переопределяться в дочерних классах:

```
class A {
    class var x: String { return "Anna" }
```

```
}
class B: A {
        override class var x: String { return "Peter" }
}
print(A.x) → Anna
print(B.x) →  Peter
```

Класс ***B*** наследует классу ***A*** и в классе ***B*** с помощью директивы ***override*** переопределено значение переменной класса ***A***, при этом в классе ***B*** переменная ***x*** также статическая. В обоих классах переменная ***x*** доступна только через имя класса и недоступна через экземпляры класса.

Точно так же статическими могут быть члены структур и enums, например:

```
struct S {
        static var x: Int = 77
        var y: Int
}
let a = S( y: 3)
print(S.x) → 77
print(a.y) → 3
```

Здесь также статическая переменная ***x*** недоступна через экземпляр структуры, то-есть нельзя написать:

```
print(a.x)
```

Иногда в программах используют так называемые синглетоны (***singleton***) — классы, не позволяющие создавать экземпляры, не разрешающие наследование и переопределение переменных и методов. В Swift синглетон можно создать с помощью модификаторов ***static*** и ***private***:

```
class A {
        static let x = A()
        private init() {}
        func f() { print("Hello") }
}
A.x.f() →  Hello
```

Переменная ***x*** представляет как бы экземпляр класса в теле самого класса. Возможен такой фокус только для статических переменных. Конструктор ***init*** без параметров объявлен с модификатором ***private*** и это не позволяет создавать обычные экземпляры за пределами класса.

Тогда ***A.x*** это «внутренний» экземпляр класса, позволяющий использовать все переменные и методы класса синлетон.

Как и все современные языки программирования, Swift позволяет использовать модификаторы доступа. Всего имеется три модификатора, соответствующие трём уровням доступа:

- ***public*** – полностью открытый доступ, включая импорт модуля
- ***internal*** – доступ только внутри создаваемой программы. Этот модификатор принят «по умолчанию» и потому обычно явно не указывается
- ***private*** – доступ только внутри объекта (класс, структура или enum). Есть ещё частный вариант private - ***fileprivate*** позволяющий доступ из файла, содержащего данный объект.

Продемонстрируем все три уровня доступа на примере структуры:

```
public struct Car {
    public let make: String
    let model: String
    private let fullName: String
    fileprivate var otherName: String
    public init(_ make: String, model: String) {
        self.make = make
        self.model = model
        self.fullName = "\(make)\(model)"
        self.otherName = "\(model) - \(make)"
    }
}
let myCar = Car("Apple", model: "iCar")
```

print(myCar) → Car(make: "Apple", model: "iCar", fullName: "AppleiCar", otherName: "iCar - Apple")
print(myCar.make) → Apple
print(myCar.model) → iCar (здесь private по умолчанию)
print(myCar.otherName) → iCar - Apple
//print(myCar.fullName) — здесь будет ошибка компиляции, за пределами класса недоступно

Если внутри объекта имеется модификатор не совпадающий с модификатором для самого объекта, то будет использован модификатор более низкого уровня.

```
public class A {
    private func f() {print("Hello")}
```

```
}
internal class B: A {
        override internal func f() { super.f() }
}
let p = B()
p.f()
```

Этот код не компилируется. Хотя класс ***A*** объявлен ***public***, метод ***f*** будет ***private***, как более низкий доступ. По этой причине он не доступен в дочернем классе ***B*** и не может быть ***override***.

Приведём ещё пример использования модификатора ***private*** со знакомой уже нам функцией ***didSet***:

```
struct S {
        private (set) var s = 0
        var a: Int = 0 { didSet { s = a * a } }
}
var p = S()
p.a = 3
print(p.s) → 9
```

Здесь мы как бы скрываем детали инициализации переменной.

23. Создание новых операторов

Swift позволяет создавать новые операторы, подобные базовым. В документации их называют словом ***advanced*** — продвинутые или улучшенные. Для создания таких операторов используется директива ***operator*** с модификатором, определяющим тип оператора: ***infix***, ***prefix*** или ***postfix***. Например, Swift не имеет традиционного оператора (**) - возведение в степень, приходится использовать встроенную функцию ***pow(x,n)***. Но такой оператор можно создать. Применяется такой синтаксис:

```
import Foundation
infix operator **: BitwiseShiftPrecedence
func ** (x: Double, n: Double) -> Double {
        return pow(x, n)
}
print((2 * 3) ** 2) → 36
```

Здесь ключевое слово ***BitwiseShiftPrecedence*** определяет старшинство операции, в данном случае операция возведения в степень имеет приоритет над арифметическими операторами.

Можно даже создать краткую форму (*****=)*** этого оператора:

```
import Foundation
infix operator **=: AssignmentPrecedence
        func **= (x: inout Double, n: Double) -> Double {
        x = pow(x, n)
        return x
}
var x = 3.2
x **= 2
print(x) → 10.24
```

Что можно сказать в итоге?

Нет сомнения, что Swift современный язык программирования, по своим возможностям не уступающий многим другим языкам и универсален по назначению. В нём много чего накручено, иногда даже кажется, что кое-какие его средства избыточны. Вместе с тем вряд ли его можно назвать шедевром и ожидать, что он действительно способен вытеснить из употребления большинство других современных языков, например, даже такой перспективный язык, как Go. Главное достоинство Swift это, несомненно, возможность компиляции в исполняемый exe-файл и в этом он имеет большое преимущество над такими языками, построенными на виртуальной машине Java, как Scala, Clojure, Groovy, Fantom и другие.

В данном руководстве совсем не затронуты некоторые аспекты языка, такие, как регулярные выражения, исключения, делегаты, параллельные вычисления и некоторые другие. В документации эти разделы изложены несколько неуклюже, а для знакомства с ними требуются рассматривать довольно громоздкие примеры. Считаю, что с большими примерами надо разбираться самостоятельно. К тому же указанные темы довольно тривиальны.