GeekBrains

Основы языка Swift

Методичка к уроку 3
Коллекции и замыкания

Оглавление

Введение

На предыдущем уроке мы познакомились с функциями, перечислениями, а также узнали, что такое опционалы и научились их распаковывать.

На этой лекции вы узнаете:

- Типы коллекций
- Замыкания
- Функции высшего порядка

Термины, используемые в лекции

Массив - коллекция упорядоченных элементов

Множество - коллекция неупорядоченных элементов

Словарь - коллекция неупорядоченных элементов, типа “ключ”: “значение”

Замыкания(или closure) - самодостаточные блоки функциональности, которые можно использовать и передавать в коде.

Коллекции

Swift предоставляет три вида коллекций: массивы(Array), множества(Set) и словари(Dictionary), именно их мы рассмотрим дальше.

Массив

Массив - это коллекция упорядоченных значений, то есть все элементы идут по строгому порядку и имеют свои индексы(места). Индексы элементов начинаются с 0.
Например, последовательность [0, 1, 5, 6] можно назвать массивом Intов, а последовательность {“Cat”, “Dog”} - массивом строк.
Массивы полезны, когда необходимо обработать некую последовательность данных, например, несколько чисел или несколько строк. Также, когда с бэка приходит json сразу с несколькими моделями.- удобно их скомпоновать мв массив и работать с ними в таком виде.

Указать переменной, что она имеет тип массив, можно двумя разными способами, например, нужен массив Int:

```
var a: Array<Int> // 1 способ
var b: [Int] // 2 способ
```

2 способ называется сокращенной формой и он является более предпочтительным вариантом.

Создать пустой массив можно также несколькими способами:

```
var a = [Int]()
var b: [Int] = []
```

Также массив может быть не просто последовательность Int или String, а содержать в себе опционалы

```
var a: [Int] = [5, 10, 12] // Массив 5, 10, 12
var b: [Int?] = [5, nil, 10, 12] // Массив 5, nil, 10, 12
```

Таким образом, в массив b можно будет добавить не просто число, а еще и опционал

💡 Благодаря выводу типов, компилятор может сам вычислить тип массива

```
var d = [7.5, 6.5, 4]
```

```
print(type(of: d)) // Array<Double>
```

Создание массива с дефолтным значением

Swift предоставляет возможность создать массив со значениями по умолчанию, в этом помогает инициализатор для массива с параметрами repeating (какое значение нужно повторить) и count(сколько раз необходимо повторить)

```
var c = Array(repeating: "A", count: 3) // ["A", "A", "A"] // Будет создан массив из A, в количестве 3 штук
```

Добавление элементов в массив

Добавить элемент в массив можно при помощи функции append, элемент будет добавлен в конец массива

```
a.append(5) // В массив будет добавлено 5
```

Добавить несколько элементов можно также при помощи функции append, но с другим параметром, это уже не newElement, а contentsOf newElements.

```
a.append(contentsOf: [5, 6, 7]) // В массив будет добавлено 5, 6, 7
```

🔥 Если заранее было объявлено, что переменная это массив, например, строк - мы не сможем добавить туда число

```
var c: [String] = []
c.append(5)   Cannot convert value of type 'Int' to expected argument type 'String'
```

append добавляет элемент в конец массива, чтобы вставить элемент по конкретному индексу необходимо воспользоваться функцией insert

```
var a: [Int] = [5, 12, 10, -15, 27]
a.insert(4, at: 3) // [5, 12, 10, 4, -15, 27]
```

🔥 Использовать функцию insert можно только в индексы от 0 до количество элементов (то есть для массива из 4 элементов это индексы 0, 1, 2, 3, 4), в противном случае будет ошибка

```
var a: [Int] = [5, 12, 10, -15, 27]
a.insert(4, at: 6)   error: Execution was interrupted, reason: EXC_BAD_INSTRUCTIO...
```

Удалить элемент из массива

Для того, чтобы удалить все элементы из массива существует функция removeAll

```
var a = [5, 12, 10, -15, 27]
a.removeAll() // []
```

Можно удалить первый или последний элемент массива при помощи removeFirst и removeLast соответственно

```
var a = [5, 12, 10, -15, 27]
a.removeFirst() // [12, 10, -15, 27]
a.removeLast() // [12, 10, -15]
```

Также эти функцию возвращают удаляемый элемент, его можно получить и записать в переменную, при этом из массива элементы тоже удалятся

```
var a = [5, 12, 10, -15, 27]
let b = a.removeFirst() // b = 5
let c = a.removeLast() // c = 27
```

Еще Swift предоставляет возможность удалить несколько первых или последних элементов, необходимо вызвать функцию removeFirst или removeLast и указать количество элементов для удаления

```
var a = [5, 12, 10, -15, 27, 16, 24]
a.removeFirst(2) // [10, -15, 27, 16, 24]
a.removeLast(3) // [10, -15]
```

🔥 Количество элементов для удаления не должно превышать количество элементов в массиве

```
var a = [5, 12, 10]
a.removeFirst(4)   error: Execution was interrupted, reason: EXC_BAD_INSTRUCTION (c...
```

Получить элемент массива

Для получения первого и последнего элемента массива можно использовать first(вернет первый элемент массива) и last(вернет последний элемент массива)

```
var a: [Int] = [5, 10, 12]
print(a.first) // будет выведено Optional(5)
print(a.last) // будет выведено Optional(12)
```

Итак, у нас массив Int, а получаем опциональное значение. Почему так? Если посмотреть на first и last мы увидим, что это опциональные значение.

```
first: Int?
The first element of the collection.
```

Это спасает, например, когда хочется получить первый элемент массива, а его не существует

```
var b: [Int] = []
print(b.first) // Выведет nil, ведь массив пустой и первого элемента не существует
```

Чтобы получить элемент по конкретному индексу необходимо указать индекс через сабскрипт, допустим, необходимо получить элемент под индексом 1

```
var a = [5, 12, 10]
let b = a[1] // b = 12
```

🔥 Индекс должен существовать, иначе будет ошибка

```
var a = [5, 12, 10]
let b = a[3]  error: Execution was interrupted, reason: EXC_BAD_INSTRUCTION (code=...
```

Для получения нескольких подряд идущих элементов можно использовать диапазон

```
var a = [5, 12, 10, -14, 20]
let b = a[1...3] // [12, 10, -14]
```

Получение минимального и максимального значения

Для получения минимального и максимального значения в массиве можно воспользоваться функциями min() и max()

```
var a: [Int] = [5, 12, 10, -15, 27]
print(a.min()) // будет выведено Optional(-15)
print(a.max()) // будет выведено Optional(27)
```

Объединить два массива

Чтобы объединить два массива можно воспользоваться оператором сложения.

```
var a: [Int] = [5, 12, 10, -15, 27]
var b: [Int] = [5, 4, 3]
var c = a + b // c станет массивом [5, 12, 10, -15, 27, 5, 4, 3]
```

🔥 Для объединения массивы должны быть обязательно одинаковых типов, иначе будет ошибка

```
var c = a + b
```

Binary operator '+' cannot be applied to operands of type '[Int]' and '[String]'

Заменить элемент в массиве

Чтобы заменить элемент, находящийся на конкретном месте можно при помощи =, слева от которого находится элемент на конкретном месте, а справа то, на что необходимо изменить этот элемент.

```
var a = [5, 12, 10]
a[1] = -4
print(a) // [5, -4, 10]
```

Количество элементов в массиве

Чтобы получить количество элементов в массиве можно воспользоваться свойством count

```
var a = [5, 4, 6, 8, 10]
print(a.count) // 5
```

Также можно проверить является ли массив пустой при помощи свойства isEmpty, если массив пустой, то будет получено true, в противном случае false

```
var a = [5, 4, 6, 8, 10]
print(a.isEmpty) // false
var b: [Int] = []
print(b.isEmpty) // true
```

Итерации по массиву

Для того, чтобы пройти по массиву можно использовать for-in, который будет иметь следующую конструкцию:

```
for название для текущего элемента массива in массив {
    // тело цикла
```

```
}
```

Для следующего кода:

```
var a = [5, -4, 10]
for element in a {
    print(element)
}
```

В консоль будет выведено

```
5
-4
10
```

, то есть в первую итерацию element равен 5, затем -4, затем 10

Для того чтобы в цикле получать не только элемент массива, но и индекс необходимо использовать enumerated, в этом случае цикл будет иметь следующую конструкцию:

```
for (название для индекса, название для элемента) in
массив.enumerated() {
    // тело цикла
}
```

Для следующего кода:

```
var a = ["A", "B", "C"]

for (index, element) in a.enumerated() {
    print(element + " " + String(index))
}
```

В консоль будет выведено:

```
A 0
B 1
C 2
```

, то есть в первой итерации element равно A, index равен 0, для второй element равен B, index равен 1, а для третьей element равен C, а index 2.

Вариативность параметров в функции

Вариативный параметр - параметр, который может принимать сразу несколько значений или не иметь ни одного. С помощью такого параметра можно передать в функцию сразу несколько произвольных значений. Вариативные значения доступны в функции в виде массива.
Для того, чтобы указать, что параметр является вариативным необходимо рядом с типом указать “...”, например,

```
func test(num: Int...) {
    for number in num {
        print( number)
    }
}

test(num: 1, -4, 10)
```

В консоль будет последовательно выведено 1, затем -4, затем 10.

Множества

Множества - коллекция с неупорядоченными элементами, то есть нет гарантии, что элементы будут находится в том порядке, в котором были созданы.
На рисунке ниже можно увидеть, что элементы были заданы в одном порядке, а выведены уже совершенно в другом.

```
4  var a: Set<Int> = [6, 10, 14, 28, 5, -12]
5  print(a)
```

```
[28, 14, 5, 10, -12, 6]
```

🔥 В отличие от массива, во множестве элементы не могут повторяться

Создание множества

Создать пустое множество можно несколькими способами

```
var a: Set<Int> = []
var b = Set<Int>()
```

Также можно создать множество, которое уже будет хранить элементы

```
var a: Set<Int> = [5, 6, 10]
```

При помощи вывода типов компилятор может сам вычислить тип множества, тем не менее, указать, что переменная является именно множеством (Set) нужно обязательно, в противном случае переменная будет массивом

```
var a: Set = [5, 4, 6, 8, 10]
print(type(of: a)) // Set<Int>
```

Количество элементов во множестве

Узнать сколько элементов во множестве можно при помощи свойства count

```
var a: Set = [5, 4, 6, 8, 10]
print(a.count) // 5
```

Чтобы узнать, является ли множество пустым необходимо воспользоваться свойством isEmpty

```
var a: Set = [5, 4, 6, 8, 10]
print(a.isEmpty) // false
var b: Set<Int> = []
print(b.isEmpty) // true
```

Добавить новый элемент во множество

Для того, чтобы добавить новый элемент во множество необходимо использовать функцию insert

```
var a: Set = [5, 4, 6, 8, 10]
a.insert(7) // 7 будет добавлено в a
```

Удалить элемент из множества

Чтобы удалить все элементы из множества можно воспользоваться функцией removeAll, после ее вызова множество станет пустым.

```
var a: Set = [5, 4, 6, 8, 10]
a.removeAll() // a станет пустым
```

Для удаления конкретного элемента из множества существует функция remove(_ member), которая к тому же возвращает удаленный элемент

```
var a: Set = [5, 4, 6, 8, 10]
let b = a.remove(10) // b будет равно 10, мнодество a будет
хранить в себе значения 5, 4, 6, 8
```

Также для множества есть функции removeFirst и remove(at:), но, так как Set - коллекция неупорядоченных элементов. нет никакой гарантии, что будет удален тот, элемент, который ожидается, например, существует множество элементов:

```
var a: Set = [5, 4, 6, 8, 10]
```

И после использования removeFirst будет удалено не 5, как ожидается, а 4, поэтому использование функций removeFirst и remove(at:) небезопасно, так как может привести к неожиданному результату

```
4 var a: Set = [5, 4, 6, 8, 10]
5 let b = a.removeFirst()
6 print(a)
```

```
[10, 8, 6, 5]
```

Узнать, существует ли элемент во множестве

При помощи функции contains можно узнать, содержит ли множество определенный элемент. Функция возвращает Bool, то есть true, если множество содержит элемент и false, если такого элемента во множестве нет.

```
var a: Set = [5, 4, 6, 8, 10]
print(a.contains(8)) // true
print(a.contains(20)) // false
```

Итерация по множеству

Для итерации по множеству используется for-in, который имеет следующий синтаксис

```
for название для текущего элемента массива in массив {
    // тело цикла
}
```

Например,

```
4 var a: Set = [5, 4, 6, 8, 10]
5 for element in a {
6     print(element)
7 }
```

```
8
6
5
10
4
```

Базовые операции множеств

Существует несколько базовых операций для множеств:

1. subtracting - создание множества, которое содержит в себе только значения, которые не содержит указанное множество.

```
var a: Set = [5, 4, 6, 8, 10]
var b: Set = [1, 6, -5, 8, 10, 12]
let c = a.subtracting(b)
print(c) // [5, 4]
```

   Множество с будет содержать элементы, которые есть в a и их нет в b
2. intersection - создание множество, которое содержит в себе только общие элементы двух множеств

```
var a: Set = [5, 4, 6, 8, 10]
var b: Set = [1, 6, -5, 8, 10, 12]
let c = a.intersection(b)
print(c) // [8, 6, 10]
```

   Множество с будет содержать элементы, которые есть и в a, и в b
3. union - создание множества, которое состоит из всех элементов двух других множеств

```
var a: Set = [5, 4, 6, 8, 10]
var b: Set = [1, 6, -5, 8, 10, 12]
let c = a.union(b)
print(c) // [8, -5, 5, 6, 4, 10, 12, 1]
```

   Множество с будет содержать все элементы множества a и множества b
4. symmetricDifference - создание множества, которое не содержит элементы, которые повторяются в двух других множествах

```
var a: Set = [5, 4, 6, 8, 10]
```

```
var b: Set = [1, 6, -5, 8, 10, 12]
let c = a.symmetricDifference(b)
print(c) // [12, 5, -5, 1, 4]
```

Множество c будет содержать элементы, которые не повторяются для множеств a и b

Сравнение множеств

1. == - позволяет определить, полностью ли совпадают элементы двух множеств

```
var a: Set = [5, 4, 6, 8, 10]
var b: Set = [1, 6, -5, 8, 10, 12]
var c: Set = [6, 8, 5, 4, 10]
print(a == b) // false
print(a == c) // true
```

2. isSuperset - позволяет определить, содержит ли множество все значения указанного множества

```
var a: Set = [5, 4, 6, 8, 10, 14, 15]
var b: Set = [1, 6, -5, 8, 10, 12]
var c: Set = [6, 8, 5, 4, 10]
print(a.isSuperset(of: c)) // true
print(c.isSuperset(of: a)) // false
```

3. isSubset - позволяет определить, все ли значения множества есть в указанном множестве

```
var a: Set = [5, 4, 6, 8, 10, 14, 15]
var b: Set = [1, 6, -5, 8, 10, 12]
var c: Set = [6, 8, 5, 4, 10]
print(a.isSubset(of: c)) // false
print(c.isSubset(of: a)) // true
```

4. isDisjoint - позволяет определить, отсутствуют ли одинаковые значения в двух множествах

```
var a: Set = [5, 4, 6, 8, 10, 14, 15]
var b: Set = [1, -5, 23]
var c: Set = [6, 8, 5, 4, 10]
print(a.isDisjoint(with: b)) // true
print(a.isDisjoint(with: c)) // false
```

5. isStrictSubset - позволяет определить, является ли множество подмножеством к указанному множеству

```
var a: Set = [5, 4, 6, 8, 10, 14, 15]
```

```
var b: Set = [1, -5, 23]
var c: Set = [6, 8, 5, 4, 10]
print(a.isStrictSubset(of: c)) // false
print(c.isStrictSubset(of: a)) // true
```

6. isStrictSuperset - позволяет определить, является ли множество надмножеством к указанному множеству

```
var a: Set = [5, 4, 6, 8, 10, 14, 15]
var b: Set = [1, -5, 23]
var c: Set = [6, 8, 5, 4, 10]
print(a.isStrictSuperset(of: c)) // true
print(c.isStrictSuperset(of: a)) // false
```

Словарь

Словарь - коллекция неупорядоченных значений, типа “ключ”: “значение”, то есть каждое значение связано со своим уникальным идентификатором - ключом. Словарь указывается как Dictionary<Key, Value>, где на месте Key указывается тип ключа (например, String или Int), а на месте Value тип значения, которое будет хранится под своим уникальным ключом.

Создание словаря

Указать тип “словарь” для переменной можно двумя способами

```
var a: Dictionary<String, Int> // 1 способ
var b: [String: Int] // 2 способ
```

2 способ является сокращенным способом создания словаря.
Создать пустой массив можно также несколькими способами:

```
var a = Dictionary<String, Int>()
var b: [String: Int] = [:]
```

Для того, чтобы создать словарь уже со значениями, необходимо их указать в паре “ключ”: “значение”

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50] // будет создан словарь ["iceCream": 50, "coffee": 100]
```

В данном примере “coffee” - это ключ, а 100 - это значение.

Количество элементов в словаре

Чтобы узнать количество элементов в словаре можно воспользоваться свойством count

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50] // будет создан словарь ["iceCream": 50, "coffee": 100]
print(b.count) // 2
```

Также есть свойство isEmpty, которое поможет узнать пустой словарь или нет. Если словарь пуст, то оно будет true, в противном случае false

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50] // будет создан словарь ["iceCream": 50, "coffee": 100]
var c: [Int: String] = [:]
print(b.isEmpty) // false
print(c.isEmpty) // true
```

Добавление элемента в словарь

Первый способ добавление элемента в словарь, это через синтаксис индекса.

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50] // будет создан словарь ["iceCream": 50, "coffee": 100]
b["cake"] = 150
print(b) // ["cake": 150, "coffee": 100, "iceCream": 50]
```

Таким образом, в словарь b будет добавлен новый элемент “cake”: 150. Если же в квадратных скобках будет указан ключ, который уже существует, тогда новый элемент не добавится, а изменится значение текущего

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50]
b["cake"] = 150
print(b) // ["cake": 150, "coffee": 100, "iceCream": 50]
b["coffee"] = 120
print(b) // ["coffee": 120, "iceCream": 50, "cake": 150]
```

Таким образом, еще одного элемента с ключом “coffee” не появится, а старое значение станет равно 120

Еще один способ добавить элемент в массив - использование функции updateValue, данная функция добавит элемент в словарь, если такого ключа не существует и обновит старое значение, если указанный ключ уже есть в словаре.

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50]
b.updateValue(150, forKey: "cake")
print(b) // ["cake": 150, "coffee": 100, "iceCream": 50]
b.updateValue(120, forKey: "coffee")
print(b) // ["cake": 150, "coffee": 120, "iceCream": 50]
```

Для функции updateValue сначала указывается значение, которое необходимо установить, а затем ключ, который нужно добавить или значение для которого нужно обновить.
Также updateValue возвращает старое значение для указанного ключа, как опционал, поэтому, если элемента еще не существует - будет возвращен nil

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50]
print(b.updateValue(150, forKey: "cake")) // nill
print(b.updateValue(120, forKey: "coffee")) // Optional(100)
```

Получить значение из словаря

Для того, чтобы получить значение из словаря можно воспользоваться синтаксисом индексов, то есть в квадратных скобках указать ключ, значение которого необходимо получить. Если значение не существует, то будет получен nil, в противном случае будет получено опциональное значение.

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50]
print(b["coffee"]) // Optional(100)
print(b["tea"]) // nil
```

Удаление элемента из словаря

Первый способ удаление элемента из словаря, это присвоить по ключу nil

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50, "cake":
150]
b["coffee"] = nil
print(b) // ["iceCream": 50, "cake": 150]
```

Второй способ удаления элемента из словаря, это использования функции removeValue. Она удалит элемент, если он существует.

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50, "cake":
150]
b.removeValue(forKey: "coffee")
print(b) // ["iceCream": 50, "cake": 150]
```

Кроме того, функция removeValue возвращает удаляемое значение, если оно есть. В противном случае будет возвращен nil.

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50, "cake":
150]
print(b.removeValue(forKey: "coffee")) // Optional(100)
print(b.removeValue(forKey: "tea")) // nil
```

Итерация по словарю

Для того, чтобы поэлементно пройтись по словарю можно воспользоваться циклом for-in, который будет иметь следующий синтаксис:

```
for (имя для текущего ключа в цикле, имя для текущего значения в цикле) in словарь {
    // тело цикла
}
```

Таким образом можно будет по итерационно получить каждый элемент словаря

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50, "cake": 150]
for (key, value) in b {
    print(key)
    print(value)
}
```

В первую итерацию будет выведено сначала “coffee”, затем 100
Во вторую итерацию будет выведено “iceCream”, затем 50
В третью итерацию будет выведено “cake”, затем 150

Получить все ключи и значения

Для словаря можно получить все его ключи при помощи свойства keys

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50, "cake": 150]
print(b.keys) // ["iceCream", "cake", "coffee"]
```

Также можно получить и все значения словаря при помощи свойства values

```
var b: [String: Int] = ["coffee": 100, "iceCream": 50, "cake": 150]
print(b.values) // [50, 150, 100]
```

Замыкания

Замыкания(или closure) - самодостаточные блоки функциональности, которые можно использовать и передавать в коде. Замыкания может быть передано в функцию, а может храниться в переменной.

> 💡 Функции это частный вид замыкания

Замыкание имеет следующий синтаксис:

```
{(переменные) -> (тип возвращаемого значения) in
    // тело замыкания
}
```

Для начала рассмотрим сохранение замыкания в переменную

```
let test = { (k: Int) -> String in
    let a = String(k * 7)
    print(a)
    return a
}
```

Переменная test это closure, которая на вход принимает число с типом Int и возвращает значение с типом String. В ходе выполнения замыкания происходит следующее:

1. Создается константа a и ей присваивается результат перемножения входного значения и 7, который преобразован в строку
2. Печатается константа a
3. Возвращается значение константы a

Теперь test можно вызывать в коде

```
test(7) // Выведет 49
```

Другой вариант использования замыканий - передача их в функцию

```
func testClosure(needMultiply: Int, test: (Int) -> String) {
    test(needMultiply)
}

testClosure(needMultiply: 5, test: test) // Выведет 35
```

Функция testClosure принимает на вход значение, которое необходимо перемножить и замыкание, которое имеет некий параметр с типом Int и возвращает String. При вызове функции ее передается 5 и некое замыкание, которое удовлетворяет требованиям.

Рассмотрим более внимательно параметр test: (Int) -> String, он указывает, что замыкание, которое будет передано, должно обязательно на вход принимать Int, а на выход выдавать String. Если, например, замыкание, которое мы хотим вызвать не должно возвращать никаких параметров, что test выглядел бы так: test: (Int) -> Void. Бывают случаи, когда необходимо передать некий функциональный блок кода, который не принимает никаких параметров и не возвращает никаких значений, тогда test выглядел бы так: test: () -> Void, в таком случае само замыкание будет иметь следующий синтаксис:

```
{
    // тело замыкания
}
```

Например,

```
let test = {
    print(7)
}
```

Когда в функцию необходимо передать замыкание, то можно указать его не в качестве передаваемой переменной, а написать сразу в вызове функции

```
func testClosure(needMultiply: Int, test: (Int) -> Void) {
    test(needMultiply)
}

testClosure(needMultiply: 5, test: { a in print(a)})
```

В данном случае при вызове testClosure в качестве параметра test написано замыкание, в котором есть параметр a с типом Int (тип параметра будет выведен благодаря выводу типов, то есть при объявлении функции было сказано, что замыкание будет на вход принимать параметр с типом Int, а значит и когда пишется closure уже понятно, что a будет иметь тип Int). Затем указывается in, а далее само тело замыкания, в данном случае печатается переменная, которая была принята на вход.

Также передать замыкание в функцию можно указав само замыкание сразу после функции, в этом случае параметр test указывать не надо.

```
func testClosure(needMultiply: Int, test: (Int) -> Void) {
    test(needMultiply)
}

testClosure(needMultiply: 5) { a in print(a)}
```

🔥 Такой способ работает только если замыкание указано последним аргументом в функции.

Если замыкание является единственным аргументом в функции, то можно не указывать круглые скобки, а сразу написать closure

```
func testClosure(test: (Int) -> Void) {
```

```
    test(5)
}

testClosure { a in print(a)}
```

Сбегающее замыкание

Убегающее замыкание - замыкание, которое было передано в функцию в качестве аргумента и будет выполнено после того, как функция вернет значение. Для того, чтобы объявить замыкание сбегающим необходимо указать @escaping. Такое объявление функции будет иметь следующий синтаксис:

```
func <название функции>(<имя для замыкания>: @escaping
<замыкание>) {
    // тело функции
}
```

Например,

```
func testClosure(test: @escaping (Int) -> Void) {
    test(5)
}

testClosure { a in print(a)}
```

Сбегающее замыкание может пригодиться, когда, например, есть массив closure и необходимо добавить новую, в таком случае, если использовать обычное замыкание выскочит ошибка

```
7  var closures: [() -> Void] = []
8  func testClosure(closure: () -> Void) {
9    closures.append(closure)   Converting non-escaping parameter 'closure' to generi...
10 }
```

А если использовать escaping, то все пройдет хорошо

```
7  var closures: [() -> Void] = []
8  func testClosure(closure: @escaping () -> Void) {
9    closures.append(closure)
10 }
```

forEach

forEach - метод, который вызывает переданное замыкание для каждого элемента в последовательности в том же порядке, что и for-in, имеет следующий синтаксис

```
<последовательность>.forEach(замыкание)
```

Также, так как замыкание - единственный аргумент метода forEach - его можно указать после функции, опустив круглые скобки.

```
<последовательность>.forEach { <имя для текущего элемента последовательности> in
    // тело замыкания
}
```

```
var a: [Int] = [10, 4, 8]
a.forEach { element in
    print(element)
}
```

Для кода выше будет последовательно выведено 10, 4, 8

Если текущий элемент не должен использоваться внутри замыкания, то вместо переменной необходимо указать “_”

```
var a: [Int] = [10, 4, 8]
a.forEach { _ in
    print("Cat")
}
```

В ходе выполнения данного кода будет последовательно выведено “Cat”, “Cat”, “Cat”.

Сокращенные имена параметров для замыканий

В Swift есть сокращенные имена параметров для однострочных замыканий. Они могут быть использованы для обращения к параметрам внутри замыкания. Сокращенные имена: $0 - текущий элемент, $1 - следующий элемент, $2 - элемент через элемент. Такие сокращенные имена могут использоваться, например, в forEach.

```
var a: [Int] = [10, 4, 8]
a.forEach { print($0) }
```

В данном случае будет последовательно выведено 10, 4, 8.

Функции высшего порядка

map

При помощи map можно пройтись по коллекции и выполнить одно и то же действия со всеми элементами коллекции

```
var a: [Int] = [10, 4, 8]
a = a.map { $0 + 1 }
print(a)
```

В данном случае каждый элемент массива a будет увеличен на 1, поэтому будет выведено [11, 5, 9]

compactMap

compactMap - тоже, что и map, но при этом удаляются все опционалы.
Например, если выполнить код ниже, используя функцию map, то будет выведено [Optional(10), nil, nil, Optional(11), Optional(5)], ведь “Cat” и “Dog” невозможно преобразовать в число

```
var a = ["10", "Cat", "Dog", "11", "5"]
var b = a.map { Int($0) }
print(b) // [Optional(10), nil, nil, Optional(11), Optional(5)]
```

Если же использовать вместо map compactMap, то будет выведено [10, 11, 5], то есть все значения будут распакованы, а nil удалены

```
var a = ["10", "Cat", "Dog", "11", "5"]
var b = a.compactMap { Int($0) }
print(b) // [10, 11, 5]
```

sorted

sorted - сортирует коллекцию по заданным условиям.

```
var a = [10, -4, 20, 7, 0, 15]
var b = a.sorted { $0 > $1 }
print(b) // [20, 15, 10, 7, 0, -4]
```

В данном случае массив будет отсортирован по условию $0 > $1, то есть текущий элемент должен быть больше следующего
Также можно вызвать sorted и указать знак по которому нужно отсортировать, например, если указать “<” массив будет отсортирован по возрастанию

```
var a = [10, -4, 20, 7, 0, 15]
var b = a.sorted(by: <)
print(b) // [-4, 0, 7, 10, 15, 20]
```

filter

filter - позволяет отфильтровать массив по указанным условиям
Например, в коде ниже условие $0 < 6, то есть текущий элемент должен быть меньше 6, а значит массив b будет состоять из отфильтрованных элементов массива a, то есть тех элементов, что меньше 6

```
var a = [10, -4, 20, 7, 0, 15]
var b = a.filter { $0 < 6 }
print(b) // [-4, 0]
```

reduce

reduce - приводит все элементы коллекции к единому значению, например, если нужно сложить все элемента массива функция reduce пригодится как нельзя кстати.

```
var a = [10, -4, 20, 7, 0, 15]
var b = a.reduce(0, +)
print(b) // 48
```

В данном случае 0 - стартовое значение, с которого необходимо начать складывать, а + - показывает, что нужно сложить все элементы
Например, если бы вместо 0 было 5, то значение было уже не 48, а 53, так как стартовое значение увеличилось на 5

```
var a = [10, -4, 20, 7, 0, 15]
var b = a.reduce(5, +)
print(b) // 53
```

flatMap

flatMap необходим, когда есть коллекци внутри коллекции и необходимо их объединить в одну коллекцию

```
var a = [[10, -4], [20, 7], [0, 15]]
var b = a.flatMap { $0 }
print(b) // [10, -4, 20, 7, 0, 15]
```

В функции выше есть переменная a, которая является массивом из массивов и при помощи функции flatMap все эти массивы будут объединены в один массив - b

Что можно почитать еще?

1. SwiftBook. Замыкания
2. SwiftBook. Типы коллекций
3. Функции высшего порядка

Используемая литература

1. https://docs.swift.org/swift-book/LanguageGuide/Closures.html
2. https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html