**Лабораторная работа №4**

**ПОСЛЕДОВАТЕЛЬНОСТИ И КОЛЛЕКЦИИ**

**ФУНКЦИИ И ЗАМЫКАНИЯ**

Swift предлагает три фундаментальных типа данных, обеспечивающих функциональность коллекций*: массив (array), набор (set) и словарь (dictionary).* Все три типа коллекций по аналогии с кортежами представляют собой отдельные типы данных.

**Массивы**

Полная форма указания типа массива в языке Swift:

Array<SomeType>

, где SomeType является типом значений, которые могут содержаться в массиве. Вы также можете написать тип массива в сокращенном виде как [SomeType]. Хотя обе формы функционально идентичны, но в сокращенном виде запись, является предпочтительной.

Типом массива является тип-значение (value type), а не тип-ссылка (reference type). Это означает, что при попытке скопировать или передать значение массива создается его копия, с которой и происходит вся дальнейшая работа.

**Создание и инициализация массива**

Массивы хранятся в переменных и константах, поэтому для их объявления используются операторы var и let.

Вы можете создать пустой массив определенного типа, используя синтаксис инициализации:

var someInts = [ Int ]()

print ( "someInts имеет тип [Int] и содержит \( someInts.count ) элементов." )

// prints "someInts элементов [Int] и содержит 0 элементов.»

Если контекст уже обеспечивает информацию о типе, как аргумент функции или типизированная переменная или константа, то вы можете создать пустой массив с пустым литералом массива, который пишется как [] (пустая пара квадратных скобок).

someInts.append ( 3 )

// someInts теперь содержит 1 значение типа Int

someInts = []

// someInts теперь пустой массив, но по-прежнему остается типом [Int]

/\* объявляем массив с пустым значением

с помощью переданного значения \*/

var emptyArray: [String] = []

/\* объявляем массив с пустым значением

с помощью специальной функции \*/

var anotherEmptyArray = [String]()

В результате создаются два пустых массива, emptyArray и anotherEmptyArray, уже инициализированные значениями (хотя и не содержащими элементов)

Тип Array в языке Swift так же имеет свой инициализатор для создания массива определенного размера с присвоением значения по умолчанию всем исходным элементам. Вы передаете инициализатору число элементов, которое должно быть добавлено массиву (count) и значение по умолчанию с подходящим типом (repeatedValue):

var threeDoubles = [ Double ]( count : 3 , repeatedValue : 0.0 )

// threeDoubles имеет тип [Double], и равен [0.0, 0.0, 0.0]

let repeatArray = Array(repeating: "Swift", count: 5)

Вы можете создать новый массив путем сложения двух существующих массивов с совместимыми типами при помощи оператора сложения ( + ). Тип нового массива выводится из типа двух складываемых массивов:

var anotherThreeDoubles = [Double](repeating: 2.5, count: 3)

// anotherThreeDoubles выведен тип [Double], и равен [2.5, 2.5, 2.5]

var sixDoubles = anotherThreeDoubles + anotherThreeDoubles

// sixDoubles выведен тип [Double], и равен [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]

**Литералы массива**

Вы можете инициализировать массив литералом массива, что является сокращенной формой для записи одной или нескольких величин в коллекцию массива. Литерал массива записывается в виде списка значений, разделенных запятыми, в окружении пары квадратных скобок:

[Значение 1, значение 2, значение 3]

Пример ниже создает массив с именем shoppingList для хранения String значений:

var shoppingList: [String] = ["Eggs", "Milk"]

// shoppingList был инициализирован двумя исходными элементами

shoppingList переменная объявлена как «массив строковых значений», записанная, как [String]. Поскольку для этого массива указан тип хранимых значений как String, то только String значения ему и разрешено хранить.. Здесь shoppingList массив инициализируется с двумя Sting значениями ( «Eggs» и «Milk» ), записанных в  литерал массива .

Благодаря выведению типов Swift, вы не обязаны писать типа массива, если вы инициализируете его литералом, содержащим элементы одного типа. Значит инициализация shoppingList может быть записано вот так:

var shoppingList = [ "Eggs" , "Milk" ]

Поскольку все значения в литерале массива имеют один и тот же тип, Swift может сделать вывод, что [String] правильный тип для переменной shoppingList.

Вы можете создать массив с помощью функции Array(arrayLiteral:)

let alphabetArray = Array(arrayLiteral: "a", "b ", "c")

При необходимости создания массива, состоящего из нескольких последовательных целочисленных значений, можно использовать оператор диапазона.

*СИНТАКСИС*

Array(значение\_1...значение\_N)

Array(значение\_1..<значение\_N)

let lineArray = Array(0…9)

**Доступ и изменение массива**

Вы можете получать значения и изменять массив через его методы и свойства, или с помощью синтаксиса index.

Чтобы получить количество элементов в массиве воспользуемся read-only свойством count:

println ( "Список покупок содержит \( shoppingList . count ) элемента." )

// Выведет "Список покупок содержит 2 элемента."

Используйте свойство isEmpty типа Bool для быстрой проверки равенства свойства count нулю (0):

if shoppingList . isEmpty {

println ( "Список покупок пустой." )

} else {

println ( "Список покупок не пустой." )

}

// выведет "Список покупок не пустой."

Вы можете добавить новый элемент в конец массива путем вызова метода append(\_:):

shoppingList . append ( "Flour" )

// shoppingList теперь сдержит 3 элемента

Как альтернатива методу append, вы можете использовать оператор (+=) для добавления нового элемента в конец массива:

shoppingList += [ "Cake" ]

// shoppingList теперь содержит 4 элемента

shoppingList += [ "Chocolate" , "Cheese" , "Butter" ]

// shoppingList теперь содержит 7 элементов

Значения элементов подлежащих слиянию массивов должны иметь один и тот же тип данных.

Чтобы получить значение из массива, используйте синтаксис index, передавая значения индекса в квадратных скобках сразу после имени массива:

var firstItem = shoppingList [ 0 ]

// firstItem имеет значение "Eggs"

Массивы в Swift всегда нумеруются с нуля (0).

Вы можете использовать синтаксис index для изменения существующего значения в данном индексе:

shoppingList [ 0 ] = "Six eggs"

// теперь первое значение списка имеет значение "Six eggs" в место "Eggs"

Вы также можете использовать синтаксис index для изменения сразу диапазона значений, даже если набор значений имеет разную длину, чем в указанном диапазоне. Следующий пример заменяет «Chocolate», «Cheese», и «Butter» на значение «Bananas» и «Apples» :

shoppingList [ 4 ... 6 ] = [ "Bananas" , "Apples" ]

// shoppingList содержит 6 элементов

*Вы не можете использовать синтаксис Index, чтобы добавить новый элемент в конец массива.*

Для того чтобы вставить элемент на определенную позицию в массиве, нужно использовать метод insert(atIndex:):

shoppingList.insert("Syrup", atIndex: 0)

// shoppingList теперь содержит 7 элементов

// " Syrup" теперь первый элемент списка

Вызов метода insert вставляет новый элемент со значением «Syrup» в самое начало массива.

Тоже самое происходит и при удалении элементов из массива с помощью метода removeAtIndex.

let syrup = shoppingList.removeAtIndex(0)

// элемент с индексом 0 был удален

// shoppingList теперь содержит 6 элементов, Syrup удален

// константа syrup теперь равна удаленной строке "Syrup"

*Если вы попробуете использовать синтаксис index, чтобы получить или назначить новое значение индексу, который находиться за пределами существующего массива, то сработает runtime ошибка.*

Все пробелы внутри массива закрываются существующими элементами, то есть происходит сдвиг элементов по индексу. Теперь новый элемент с индексом 0 будет «Six eggs»:

firstItem = shoppingList [ 0 ]

// firstItem теперь равно "Six eggs"

 Если нужно удалить последний элемент массива, то используйте метод removeLast() вместо removeAtIndex( для избежание дополнительного вызова свойства count. Как и removeAtIndex(:), метод removeLast() так же возвращает значение удаленного элемента:

let apples = shoppingList.removeLast()

// последний элемент массива был удален

// shoppingList теперь содержит 5 элементов, и нет "Apples"

// константа apples теперь равна удаленной строке "Apples"

**Перебор массива**

Вы можете перебрать весь набор значений в массиве циклом (for — in):

for item in shoppingList {

println ( item )

}

// Six eggs

// Milk

// Flour

// Cake

// Bananas

Если вам нужно значение каждого элемента массива и его индекс, то необходимо использовать глобальную функцию enumerate для итерации по массиву. Функция enumerate возвращает кортеж, состоящий из индекса и значения элемента массива. Вы можете разложить кортеж во временные константы или переменные, как часть итерации:

for ( index , value ) in enumerate ( shoppingList ) {

println ( "Item \( index + 1 ) : \( value ) " )

}

// Item 1: Six eggs

// Item 2: Milk

// Item 3: Flour

// Item 4: Cake

// Item 5: Bananas

**Сравнение массивов**

Массивы, так же как и значения фундаментальных типов данных, можно сравнивать друг с другом . Два массива являются эквивалентными:

1) если количество элементов в сравниваемых массивах одинаково;

2) каждая соответствующая пара элементов эквивалентна .

let a1 = 1

let a2 = 2

let a3 = 3

if [1, 2, 3] == [a1, a2, a3] {

print("Массивы эквивалентны")

} else {

print("Массивы не эквивалентны")

}

Несмотря на то что в массиве [a1, a2, a3] указаны не значения, а константы, содержащие эти значения, условия эквивалентности массивов все равно выполняются .

**Многомерные массивы**

Элементами массива могут быть не только значения фундаментальных типов, но и другие массивы. Массивы, содержащие в себе другие массивы, называются многомерными. Необходимо обеспечить единство типа всех вложенных массивов.

var arrayOfArrays = [[1,2,3], [4,5,6], [7,8,9]]

Типом основного массива arrayOfArrays является [[Int]] (с удвоенными квадратными скобками с каждой стороны) .

Для доступа к элементу многомерного массива необходимо указывать несколько индексов

arrayOfArrays[2][1]

**Базовые свойства и методы массивов**

Свойство count возвращает количество элементов в массиве

var someArray = [1, 2, 3, 4, 5]

// количество элементов в массиве

someArray.count //5

Если значение свойства count равно нулю, то и свойство isEmpty возвращает true

var someArray: [Int] = [] //[]

someArray.count//0

someArray.isEmpty//true

Метод suffix(\_:) В качестве входного параметра ему передается количество элементов, которые необходимо получить . Элементы отсчитываются начиная с последнего элемента массива

var someArray = [1, 2, 3, 4, 5]

// получаем три последних элемента массива

let subArray = someArray.suffix(3) //[3,4,5]

Свойства first и last возвращают первый и последний элементы массива

someArray.first //1

someArray.last //5

С помощью метода append(\_:) можно добавить новый элемент в конец массива

Если массив хранится в переменной (то есть является изменяемым), то метод insert(\_:atIndex:) вставляет в массив новый одиночный элемент с указанным индексом

var someArray = [1, 2, 3, 4, 5]

// вставляем новый элемент в середину массива

someArray.insert(100, at: 2)

someArray

Так же как в случае изменения массива, методы remove(at:), removeFirst() и removeLast() позволяют удалять требуемые элементы . При этом они возвращают значение удаляемого элемента. После удаления индексы оставшихся элементов массива перестраиваются.

Если массив является неизменяемым (хранится в константе), то можно использовать методы dropFirst(\_:) и dropLast(\_:), возвращающие новый массив, в котором отсутствует несколько первых или последних элементов .

let someArray = [1, 2, 3, 4, 5]

// удаляем последний элемент

someArray.dropLast()

// удаляем три первых элемента

var newArray = someArray.dropFirst(3)

newArray //4, 5

При использовании данных методов основной массив someArray, с которым выполняются операции, не меняется (у нас let, а не var) . Они лишь возвращают получившееся значение, которое при необходимости может быть записано в новый параметр.

Метод contains(\_:) определяет факт наличия некоторого элемента в массиве и возвращает Bool в зависимости от результата

let intArray = [1, 2, 3, 4, 5, 6]

let resultTrue = intArray.contains(4)

Метод index(of:) сообщает индекс первого вхождения искомого элемента в рассматриваемом массиве. Так как искомый элемент может отсутствовать, метод возвращает опциональное значение. Если элемент отсутствует, то возвращается nil.

let intArray = [1, 2, 3, 4, 5, 6, 4, 5, 6]

// поиск первого вхождения элемента

let result = intArray.index(of:4) //3

Для поиска минимального или максимального элемента в массиве применяются методы min () и max (). *Сопоставлять* можно значения тех типов данных, которые являются хешируемыми, то есть в них должен присутствовать функционал вычисления хеша для значения. Большинство фундаментальных типов данных поддерживают хеширование.

Для доступа к хешу параметра достаточно вызвать указанное свойство:

var a: Float = 3.5

a.hashValue // 1 080 033 280

Чтобы изменить порядок следования всех элементов массива на противоположный, используйте метод reverse().

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 4.1**

1. *Определите массив целых. Проинициализируйте его последовательными числами от 5 до 125. Удалите 5 первых элементов. Замените 5 последних элементов на два 0-ых. Выведите значения массива.*
2. *Выведите массив с использованием enumerate (кортеж – индекс и значение).*

**Наборы (множества)**

В отличие от массивов, у элементов набора нет какого-либо определенного порядка следования, важен лишь факт наличия некоторого значения в наборе. Определенное значение элемента может существовать в нем лишь единожды.

Набор создается с помощью литерала набора. В плане синтаксиса он идентичен литералу массива. Тип данных элементов набора должен быть хешируемым, то есть поддерживать протокол Hashable. При создании набора необходимо явно указать, что создается именно набор. В связи с этим необходимо:

* либо явно указать тип данных набора с использованием конструкции Set<ТипДанных>, где ТипДанных — тип элементов создаваемого набора;
* либо использовать функцию Set<ТипДанных>, которой в качестве входного передается параметр arrayLiteral, содержащий перечень элементов набора.

Если нет необходимости явно указывать тип данных значений создаваемого набора, можно использовать ключевое слово Set без конструкции <ТипДанных>.

/\* набор, созданный путем

явного указания типа \*/

var dishes: Set<String> = ["ice", "fruit"]

var dishesTwo: Set = ["ice", "fruit"]

var members = Set<String>(arrayLiteral: "Nikita", "Olga")

var membersTwo = Set(arrayLiteral: "Nikita", "Olga")

Пустой набор создается с помощью пустого литерала набора [] .

var dishes: Set<String> = ["bread", "milk"]

// создание пустого набора

var members = Set<String>()

// удаление всех элементов набора

dishes = []

**Доступ к набору и модификация набора**

Так как набор — это неупорядоченная коллекция элементов, не имеющая каких-либо индексов или ключей, обеспечивающих доступ к значениям, то использование синтаксиса сабскриптов невозможно.

Для создания нового значения в наборе применяется метод insert(\_:)

var musicStyleSet: Set<String> = []

musicStyleSet.insert("Jazz")

musicStyleSet //Jazz

В результате выполнения метода insert(\_:) возвращается массив, первый элемент которого содержит значение типа Bool, характеризующее успешность проведенной операции, и значение добавляемого элемента . В качестве первого элемента массива может быть возвращено false в том случае, когда производилась попытка добавить уже существующий в наборе элемент

Для удаления элемента из набора используется метод remove(\_:), который удаляет элемент с указанным значением и возвращает удаленное значение или nil, если удаляемого элемента не существует . Также вы можете задействовать метод removeAll() для удаления всех элементов набора

var musicStyleSet: Set<String> = ["Jazz", "Hip-Hop", "Rock"]

musicStyleSet.remove("Hip-Hop")

// удаляем несуществующий элемент

musicStyleSet.remove("Classic") //nil

musicStyleSet.removeAll() ///Set([])

Проверка факта наличия значения в наборе осуществляется методом contains(\_:) . Данный метод возвращает значение типа Bool в зависимости от результата проверки

**Базовые свойства и методы наборов**

Для двух любых наборов можно произвести следующие операции :

* получить все общие для обоих наборов элементы (intersection) - создается новый набор, содержащий значения, общие для двух наборов;

var oddDigits:Set<Int> = [2,4,6]

var differentDigits:Set<Int> = [1,2,3,4,6]

var inter = differentDigits.intersection(oddDigits).sorted()

* получить все непересекающиеся (не общие) для обоих наборов элементы (symmetricDifference);

var exclusive = differentDigits.symmetricDifference(oddDigits).sorted()

* получить все элементы обоих наборов (union);

var union = differentDigits.union(oddDigits).sorted()

* получить разницу элементов, то есть элементы, которые входят в первый набор, но не входят во второй (subtracting) .

var subtract = differentDigits.subtracting(oddDigits)

Два набора считаются эквивалентными, если у них один и тот же набор элементов. Эквивалентность наборов проверяется с помощью оператора эквивалентности (==)

var bSet: Set = [1, 3]

var copyOfBSet = bSet

if bSet == copyOfBSet {

print("Наборы эквивалентны")

}

Метод isSubset(of:) определяет, является ли один набор субнабором другого.

var aSet: Set = [1, 2, 3, 4, 5]

var bSet: Set = [1, 3]

if bSet.isSubset(of: aSet) {

print("bSet — это субнабор для aSet")

}

Метод isSuperset(of:) вычисляет, является ли набор супернабором для другого набора.

Метод isDisjoint(with:) определяет, существуют ли в двух наборах общие элементы, и в случае их отсутствия возвращает true

Методы isStrictSubset(of:) и isStrictSuperset(of:) определяют, является набор субнабором или супернабором, не равным указанному множеству

var aSet: Set = [1, 2, 3, 4, 5]

var bSet: Set = [1, 3]

if bSet.isStrictSubset(of: aSet) {

print("bSet — субнабор для aSet")

}

if aSet.isStrictSuperset(of: bSet) {

print("aSet — супернабор для bSet")

}

*Самое важное, чтобы при создании набора вы не забывали использовать ключевое слово Set, иначе в результате будет создан массив.*

**Словари**

Словарь — это неупорядоченная коллекция элементов одного и того же типа, для доступа к значениям которых используются ключи. Каждый элемент словаря состоит из уникального ключа, который указывает на данный элемент, и значения. В качестве ключа выступает не автоматически устанавливаемый индекс (как в массивах), а уникальный для словаря литерал произвольного типа, устанавливаемый разработчиком. Чаще всего ключи — это строковые литералы.

Значение словаря устанавливается с помощью литерала словаря.

var dictionary = ["one":"один", "two": "два", "three":"три"]

При попытке создания словаря с двумя одинаковыми ключами Xcode сообщит об ошибке.

Другим способом является инициализация словаря на основе двух произвольных коллекций

// базовая коллекция кортежей (пар значений)

let baseCollection = [(2, 5), (3, 6), (1, 4)]

// создание словаря на основе базовой коллекции

let newDictionary = Dictionary(uniqueKeysWithValues:baseCollection)

// [2: 5, 3: 6, 1: 4]

Функция zip(\_:\_:) - возвращает последовательность пар, основанную на двух базовых последовательностях. После этого сформированная последовательность передается свойству uniqueKeysWithValues инициализатора типа Dictionary. В качестве ключей результирующий словарь будет содержать значения первой базовой коллекции, а в качестве значения — элементы второй базовой коллекции.

let nearestStarNames = ["Proxima Centauri", "Alpha Centauri A", "Alpha Centauri B", "Barnard's Star", "Wolf 359"]

let nearestStarDistances = [4.24, 4.37, 4.37, 5.96, 7.78]

let starDistanceDict = Dictionary(uniqueKeysWithValues: zip(nearestStarNames, nearestStarDistances))

// ["Wolf 359": 7.78, "Alpha Centauri B": 4.37, "Proxima Centauri":

//4.24;, "Alpha Centauri A": 4.37, "Barnard's Star": 5.96]

**Взаимодействие с элементами словаря**

Ключи предназначены не только для получения значений элементов словаря, но и для их изменения

var countryDict = ["RUS":"Россия", "BEL": "Беларусь", "UKR":"Украина"]

countryDict["RUS"] = "Russian"

Для обновления значения элемента словаря можно также использовать метод updateValue(\_:forKey:) . Данный метод возвращает опциональное старое значение (nil, если значения по изменяемому ключу не существует).

Для удаления некоторого элемента (пары «ключ-значение») достаточно присвоить удаляемому элементу nil или использовать метод removeValue(forKey:), указав ключ элемента - возвращается значение удаляемого элемента. Если вы попытаетесь получить доступ к несуществующему элементу словаря, это не приведет к ошибке — Swift просто вернет nil. Это говорит о том, что любое возвращаемое значение элемента словаря — опционал.

countryDict["UKR"] = nil

countryDict.removeValue(forKey: "BEL")

let myCountry: String? = countryDict["BEL"] //nil

Для преобразования возвращаемого значения элемента из типа String? в тип String выполняется принудительное извлечение значения.

Тип данных элементов словаря содержит в себе два типа данных: тип ключа и тип значения .

*СИНТАКСИС*

var имя\_словаря: Dictionary<ТипКлюча:ТипЗначения>

var имя\_словаря: [ТипКлюча:ТипЗначения]

Для того чтобы создать пустой словарь, он должен быть инициализирован значением без элементов . Это делается с помощью конструкции [:], которая как раз и является литералом словаря, не имеющего элементов

var emptyDictionary: [String:Int] = [:]

var AnotherEmptyDictionary: Dictionary<String,Int> = [:]

С помощью конструкции [:] можно также уничтожить все элементы словаря, если присвоить ее словарю в качестве значения

**Базовые свойства и методы словарей**

Свойство count возвращает количество элементов в словаре

Если свойство count равно нулю, то свойство isEmpty возвращает true

var someDictionary: [String:Int] = [:]

someDictionary.count

someDictionary.isEmpty

При необходимости вы можете получить все ключи или все значения словаря с помощью свойств keys и values

// все ключи словаря countryDict

var keys = [String](countryDict.keys) //["BEL", "UKR", "RUS"]

// все значения словаря countryDict

var values = [String](countryvaDict.values) //["Беларусь", "Украина", "Россия"]

При вызове свойства keys или values Swift возвращает не массив, набор или словарь, а значение некоего типа LazyMapCollection, которое представляет собой ленивую коллекцию.

С помощью цикла for-in удобно перебирать значения коллекций . Возможна ситуация, когда требуется получить не пару «ключ-значение» из словаря, а только ключ или только значение . Для этого в кортеже на месте того элемента, который загружать не следует, необходимо вставить символ нижнего подчеркивания

var countriesAndBlocks = ["Россия": "ЕАЭС", "США": "НАТО", "Франция":"ЕС"]

for (countryName, \_) in countriesAndBlocks {

print("страна — \(countryName)")

}

Можно сделать обход только по ключам словаря:

for country in countryDict.keys {

print("Counrty code: \(country)")

}

//Counrty code: BEL

//Counrty code: UKR

//Counrty code: RUS

let somePoint = (1, 1)

switch somePoint {

case (0, 0):

print("\(somePoint) is at the origin")

case (\_, 0):

print("\(somePoint) is on the x-axis")

case (0, \_):

print("\(somePoint) is on the y-axis")

case (-2...2, -2...2):

print("\(somePoint) is inside the box")

default:

print("\(somePoint) is outside of the box")

}

// Prints "(1, 1) is inside the box»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 4.2**

1. *Создайте словарь типа [String: (alpha:Character,num:Int)?], который описывает шахматную фигуру на игровом поле. В ключе словаря должно храниться имя фигуры, например «Белый конь», а в значении — кортеж, указывающий на координаты фигуры на игровом поле. Если вместо кортежа находится nil, это означает, что фигура убита (не имеет координат на игровом поле).*
2. *Добавьте в него три произвольные фигуры, одна из которых не должна иметь координат.*

*3) Создайте конструкцию if-else, которая проверяет, убита ли переданная ей фигура (элемент словаря), и выводит на консоль информацию либо о координатах фигуры, либо о ее отсутствии на игровом поле. Для получения координат переданной фигуры используйте опциональное связывание.*

**Функции и замыкания**

Используйте func для объявления функции. Функция вызывается по ее имени и списку параметров в скобках. Используйте -> для отделения имен параметров и типов от возвращаемого значения функции

СИНТАКСИС

func имяФункции (входные\_параметры) ->

ТипВозвращаемогоЗначения {

тело\_функции

}

func greet(name: String, day: String) -> String {

return "Hello \(name), today is \(day)."

}

greet(name: "Bob", day: "Tuesday")

Если функция не возвращает никакого значения, то в качестве типа данных возвращаемого значения необходимо указать тип Void. Существует альтернативная ключевому слову Void форма записи — пустые скобки, которые также сообщают о том, что функция не возвращает значение.

func printMessage() -> () {

print("Сообщение принято")

}

Кроме того, если функция не возвращает никакого значения, написание какой бы то ни было конструкции можно вовсе опустить.

func printMessage() {

print("Сообщение принято")

}

Существует способ сообщить Swift, что для определенных параметров нет необходимости указывать имена при функции. Для этого перед именем таких параметров необходимо поставить символ нижнего подчеркивания, отделив его от имени пробелом

func sum(a:Int, \_ b: Int, c:Int) -> () {

print("Сумма — \(a+b+c)")

}

sum(a: 15, 12, c: 9)

Все входные параметры функции — константы. При попытке изменения их значения внутри тела функции происходит ошибка. При необходимости изменения переданного входного значения внутри функции потребуется создать новую переменную и присвоить переданное значение ей.

Используйте кортеж (tuple) для возврата нескольких значений из функции. Элементы набора могут быть отнесены либо по имени или по номеру.

func minMax(array: [Int]) -> (min: Int, max: Int) {

var currentMin = array[0]

var currentMax = array[0]

for value in array[1..<array.count] {

if value < currentMin {

currentMin = value

} else if value > currentMax {

currentMax = value

}

}

return (currentMin, currentMax)

}

let stat = minMax(array: [4,65,7,8]) //(min 4, max 65)

stat.min // 4

stat.1 // 65

**Входной параметр с переменным числом аргументов**

Функции могут использовать разное число аргументов. Вариативный параметр обозначается в списке входящих параметров указанием оператора закрытого диапазона (...) сразу после типа входного параметра. Значения для этого параметра при вызове функции задаются через запятую.

func sumOf(numbers: Int...) -> Int {

var sum = 0

for number in numbers {

sum += number

}

return sum

}

sumOf() //0

sumOf(numbers: 42, 597, 12) //

У одной функции может быть только один вариативный параметр. Он должен находиться в самом конце списка входных параметров.

**Вложенные функции**

Функции могут быть вложенными. Вложенные функции имеют доступ к переменным, объявленным во внешнем функции. Используйте вложенные функции для организации кода длинной и сложной функции.

func returnFifteen() -> Int {

var y = 10

func add() {

y += 5

}

add()

return y

}

returnFifteen() //15

Функции по своей сути являются *типом* — то есть вы можете возвращать функцию из функции:

func makeIncrementer() -> ((Int) -> Int) {

func addOne(number: Int) -> Int {

return 1 + number

}

return addOne

}

var increment = makeIncrementer()

increment(7)

**Сквозные параметры**

Для того чтобы входные аргументы сохранили свои значения даже после завершения вызова функции, необходимо использовать *сквозные параметры*. Чтобы преобразовать входной параметр в сквозной, перед описанием его типа необходимо указать модификатор inout. Сквозной параметр передается в функцию, изменяется в ней и сохраняет свое значение при завершении работы функции, заменяя собой исходное значение. При вызове функции перед передаваемым значением аргумента необходимо ставить символ амперсанда (&), указывающий на то, что параметр передается по ссылке.

func changeValues(\_ a: inout Int, \_ b: inout Int) -> () {

let tmp = a

a = b

b = tmp

}

var a = 150, b = 45

changeValues(&a, &b)

a //45

b //150

Аргументом сквозного параметра может быть только *переменная*. Константы или литералы нельзя передавать, так как они являются неизменяемыми.

**Функция в качестве значения аргумента**

Функция может принимать другую функцию, как ее аргумент:

func hasAnyMatches(list: [Int], condition:(Int)-> Bool)->Bool{

for item in list {

if condition(item) {

return true

}

}

return false

}

func lessThanTen(number: Int) -> Bool {

return number < 10

}

var numbers = [20, 19, 7, 12]

hasAnyMatches(list: numbers, condition: lessThanTen)//true

**Внешние имена аргументов**

Для любого входного параметра можно задать его внешнее имя, то есть имя, которое указывается при вызове функции. Оно пишется перед внутренним именем аргумента и отделяется от него пробелом

func sumWallet(banknotsArray wallet: [Int]? = nil ) -> Int? {

var sum = 0

if wallet == nil {

return nil

}

for oneBanknote in wallet! {

sum += oneBanknote

}

return sum

}

// сосчитаем сумму всех купюр

sumWallet(banknotsArray: [50, 100, 100, 500, 50, 1000, 5000, 50, 100]) //6950

Входной параметр wallet теперь имеет внешнее имя banknotsArray, поэтому при вызове функции sumWallet(banknotsArray:) необходимо указать не только значение параметра, но и его внешнее имя.

Внешние имена входных параметров служат для того, чтобы скрывать их внутренние имена. Например, в качестве внешних вы можете использовать логически понятные для разработчика длинные имена, а в качестве внутренних — сокращенные.

**Функциональный тип**

Если функция ничего не принимает и не возвращает, то ее тип указывается двумя парами круглых скобок, разделенных стрелкой:

() -> ()

В первых скобках всегда описываются типы данных входных параметров, а вместо вторых указывается тип данных выходного значения, если, конечно, оно существует.

**Безымянные функции или замыкания**

Безымянные функции также называются замыкающими выражениями, или замыканиями. Функции — это на самом деле специальный случай замыканий.

Замыкающие выражения могут не просто выполнять какие-либо действия, но и принимать входные параметры, а также возвращать произвольные значения.

Вы можете написать замыкание без имени, окружив код фигурными скобками. Используйте in для разделения аргументов и типа возвращаемого значения от тела замыкания:

var numbers = [20, 19, 7, 12]

numbers.map({

(number: Int) -> Int in

let result = 3 \* number

return result

}) //[60, 57, 21, 36]

Замыкающие выражения позволяют в значительной мере оптимизировать программы.

У вас есть несколько опций для написания замыканий более кратко. Когда тип замыкания уже известен, например, обратный вызов делегата (callback), вы можете пропустить тип его параметров, тип возвращаемого значения или и то и другое. Однострочное замыкание в примере ниже возвращает значение своего единственного выражения:

var numbers = [20, 19, 7, 12]

let mappedNumbers = numbers.map({ number in 3 \* number })

mappedNumbers

В результате мы добились того, что замыкающее выражение записывается всего в одну короткую строку и при этом код становится понятнее.

Вы можете ссылаться на параметры по номеру (сокращенные имена), вместо его имени — этот подход особенно удобен в очень коротких замыканиях.

**Каррирование функций**

Каррирование — это процесс, при котором функция от нескольких аргументов преобразуется в функцию (или набор функций) от одного аргумента. Это становится возможным благодаря тому, что в качестве выходного значения функции может выступать другая функция.

Рассмотрим пример. Существует функция с типом (Int, Int)->Int, которая получает на вход два целочисленных значения, производит некоторую операцию и возвращает ответ в виде целого числа:

func sum(x: Int, y: Int) -> Int {

return x + y

}

sum(x: 1,y: 4) // вернет 5

С целью каррирования напишем новую функцию, которая принимает на вход всего один целочисленный параметр, а возвращает замыкание типа (Int)->Int

func sum2(\_ x: Int) -> (Int) -> Int {

return { return $0+x }

}

var closure = sum2(1) //(Int)->Int

closure(12) //13

Переменная closure получает в качестве значения замыкание, которому мы можем передать входной параметр. Прелесть каррирования в том, что мы можем объединить вызов функции sum2(x:) и передачу значения в возвращаемое ею замыкание

sum2(5)(12) // вернет 17

Если говорить о пользе, то каррирование хорошо именно тем, что устраняет зависимость функции от нескольких параметров. Мы можем вызвать функцию, как только получим первое нужное значение, и далее при необходимости многократно вызывать возвращенное ею замыкание при поступлении второго требуемого параметра

var closure = sum2( 1) //(Int)->Int

closure(12) // вернет 13

closure(19) // вернет 20

**Захват переменных**

Swift позволяет зафиксировать значения внешних по отношению к замыканию параметров, которые они имели на момент его определения.

var a = 1

var b = 2

let closureSum : () -> Int = {

return a+b

}

closureSum() // 3

a = 3

b = 4

closureSum() // 7

Замыкание, хранящееся в константе closureSum, складывает значения перемененных a и b. При изменении их значений возвращаемое замыканием значение меняется. Существует способ «захватить» значения параметров, то есть зафиксировать те значения, которые имеют эти параметры на момент написания замыкающего выражения. Для этого в начале замыкания в квадратных скобках необходимо перечислить захватываемые переменные, разделив их запятой, после чего указать ключевое слово in.

var a = 1

var b = 2

let closureSum : () -> Int = {

[a,b] in

return a+b

}

closureSum() // 3

a = 3

b = 4

closureSum() // 3

Замыкание, хранящееся в константе closureSum, складывает значения перемененных a и b. При изменении этих значений возвращаемое замыканием значение не меняется.

Другим способом захвата значения внешнего параметра является вложенная функция, написанная в теле другой функции. Вложенная функция может захватывать произвольные переменные, константы и даже входные аргументы, объявленные в родительской функции

func makeIncrement(forIncrement amount: Int) -> () -> Int {

var runningTotal = 0

func increment() -> Int {

runningTotal += amount

return runningTotal

}

return increment

}

Функция makeIncrement(forIncrement:) возвращает значение с функциональным типом ()->Int. Это значит, что вернется замыкание, не имеющее входных аргументов и возвращающее целочисленное значение.

Вложенная функция increment() не имеет входных или объявляемых параметров, но при этом обращается к runningTotal и amount внутри своей реализации. Она делает это в автоматическом режиме путем захвата значений обоих параметров по ссылке. Захват значений по ссылке гарантирует, что измененные значения параметров не исчезнут после окончания работы функции makeIncrement(forIncrement:) и будут доступны при повторном вызове функции increment() .

var incrementByTen = makeIncrement(forIncrement: 10)

var incrementBySeven = makeIncrement(forIncrement: 7)

incrementByTen() // вернется 10

incrementByTen() // вернется 20

incrementByTen() // вернется 30

incrementBySeven() // вернется 7

incrementBySeven() // вернется 14

incrementBySeven() // вернется 21

В переменных incrementByTen и incrementBySeven хранятся возвращаемые функцией makeIncrement(forIncrement:) замыкания. В первом случае значение runningTotal увеличивается на 10, а во втором — на 7. Каждая из переменных хранит свою копию захваченного значения runningTotal, именно поэтому при их использовании увеличиваемые значения не пересекаются и увеличиваются независимо друг от друга.

Функциональный тип данных — это тип-ссылка (reference type). Это значит, что замыкания передаются не копированием, а с помощью ссылки на область памяти, где хранится это замыкание.

var incrementByFive = makeIncrement(forIncrement: 5)//()->Int

var copyIncrementByFive = incrementByFive

В данном примере используется функция makeIncrement(forIncrement:), объявленная ранее. Напомню, она возвращает замыкание типа ()->Int, которое в данном случае предназначено для увеличения значения на 5 . Возвращаемое замыкание записывается в переменную incrementByFive, после чего копируется в переменную copyIncrementByFive. В результате можно обратиться к одному и тому же замыканию, используя как copyIncrementByFive, так и incrementByFive

incrementByFive() // вернет 5

copyIncrementByFive() // вернет 10

incrementByFive() // вернет 15

Как вы можете видеть, какую бы функцию мы ни использовали, происходит модификация одного и того же значения (каждое последующее значение больше предыдущего на 5). Это обусловлено тем, что замыкания — ссылочный тип данных, или тип-ссылка, или reference type.

**Автозамыкания**

Автозамыкания — это замыкания, которые автоматически создаются из переданного выражения. Иными словами, может существовать функция, имеющая один или несколько входных параметров, которые при ее вызове передаются как значения, но во внутренней реализации функции используются как самостоятельные замыкания.

var arrayOfNames = ["Helena", "Baz", "Alex"]

func printName(nextName: String ) {

// какой-либо код

print(nextName)

}

printName(nextName: arrayOfNames.remove(at: 0)) //Helena

При каждом вызове функции printName(nextName:) в качестве входного значения ей передается результат вызова метода remove(at:) массива arrayOfNames. Независимо от того, в какой части функции будет использоваться переданный параметр (или не будет использоваться вовсе), значение возвращаемое методом remove(at:), будет вычислено в момент вызова функции printName(nextName:). Получается, что передаваемое значение вычисляется независимо от того, нужно ли оно в ходе выполнения функции. Отличным решением данной проблемы станет использование ленивых вычислений, то есть таких вычислений, которые будут выполняться лишь в тот момент, когда это понадобится. Для того чтобы реализовать этот подход, можно передавать в функцию printName(nextName:) замыкание, которое будет вычисляться в тот момент, когда к нему обратятся

var arrayOfNames = ["Helena", "Baz", "Alex"]

func printName(nextName: ()->String) {

// какой-либо код

print(nextName())

}

printName(nextName: {arrayOfNames.remove(at: 0)}) //Helena

Для решения этой задачи потребовалось изменить тип входного параметра nextName на ()->String и заключить передаваемый метод remove(at:) в фигурные скобки. Теперь внутри реализации функции printName(nextName:) к входному аргументу nextName необходимо обращаться как к самостоятельной функции (с использованием круглых скобок после имени параметра). Таким образом, значение метода remove(at:) будет вычислено именно в тот момент, когда оно *понадобится*, а не в тот момент, когда оно будет *передано*. Единственным недостатком данного подхода является то, что входной параметр должен быть заключен в фигурные скобки, а это несколько усложняет использование функции и чтение кода. С помощью автозамыканий можно использовать положительные функции обоих рассмотренных примеров: отложить вычисление переданного значения и передавать значение в виде значения (без фигурных скобок). Для того чтобы реализовать автозамыкание, требуется, чтобы выполнялись следующие требования:

* Входной аргумент должен иметь функциональный тип. В примере, приведенном ранее, аргумент nextName уже имеет функциональный тип ()->String .
* Функциональный тип не должен определять типы входных параметров. В примере типы входных параметров не определены (пустые скобки) .
* Функциональный тип должен определять тип возвращаемого значения. В примере тип возвращаемого значения определен как String.
* Переданное выражение должно возвращать значение того же типа, которое определено в функциональном типе замыкания. В примере передаваемая в качестве входного аргумента функция возвращает значение типа String точно так же, как определено функциональным типом входного аргумента.
* Перед функциональным типом необходимо использовать атрибут @autoclosure.
* Передаваемое значение должно указываться без фигурных скобок . Перепишем код из предыдущего листинга в соответствии с указанными требованиями

var arrayOfNames = ["Helena", "Baz", "Alex"]

func printName(nextName: @autoclosure ()->String) {

// какой-либо код

print(nextName())

}

printName(nextName:arrayOfNames.remove(at: 0)) //Helena

Теперь метод remove(at:) передается в функцию printName(nextName:) как обычный аргумент, без использования фигурных скобок, но внутри тела используется как самостоятельная функция. Ярким примером глобальной функции, входящей в стандартные возможности Swift и использующей механизм автозамыканий, является функция assert(condition:message:file:line:). Аргументы condition и message — это автозамыкания, первое из которых вычисляется только в случае активного debug-режима, а второе — только в случае, когда condition соответствует false.

По умолчанию все переданные в функцию замыкания имеют ограниченную этой функцией область видимости, то есть если вы решите сохранить замыкание для дальнейшего использования, то получите сообщение об ошибке. Другими словами, все переданные в функцию замыкания называются не выходящими за пределы ее тела. Если Swift видит, что область, где замыкание доступно, ограничена, он при первой же возможности удалит его, чтобы освободить и не расходовать оперативную память. Для того чтобы позволить замыканию выйти за пределы области видимости функции, необходимо указать атрибут @escaping перед функциональным типом при описании входных параметров функции.

Рассмотрим пример. Предположим, что в программе есть специальная переменная, предназначенная для хранения замыканий, то есть являющаяся коллекцией замыканий

var arrayOfClosures: [()->Int] = [] //[]

Пока еще пустой массив arrayOfClosures может хранить в себе замыкания с функциональным типом ()->Int. Реализуем функцию, добавляющую в этот массив переданные ей в качестве аргументов замыкания.

func addNewClosureInArray(\_ newClosure: ()->Int){

arrayOfClosures.append(newClosure) // ошибка

}

Получаем:

swiftBase.playground:9:29: note: parameter 'newClosure' is implicitly non-escaping

func addNewClosureInArray(\_ newClosure: ()->Int){

^

@escaping

Xcode сообщит вам об ошибке по двум причинам:

* Замыкание — это тип-ссылка, то есть оно передается по ссылке, но не копированием.
* Замыкание, которое будет храниться в параметре newClosure, будет иметь ограниченную телом функции область видимости, а значит, не может быть добавлено в глобальную (по отношению к телу функции) переменную arrayOfClosures.

Для решения этой проблемы необходимо указать, что замыкание, хранящееся в переменной newClosure, является выходящим. Для этого перед описанием функционального типа данного аргумента укажите атрибут @escaping, после чего вы сможете передать в функцию addNewClosureInArray(\_:) произвольное замыкание

func addNewClosureInArray(\_ newClosure: @escaping ()->Int){

arrayOfClosures.append(newClosure)

}

addNewClosureInArray({return 100})

addNewClosureInArray{return 1000}

arrayOfClosures[0]() // 100

arrayOfClosures[1]() // 1000

Обратите внимание на то, что в одном случае замыкание передается с круглыми скобками, а в другом — без них. Так как функция addNewClosureInArray(\_:) имеет один входной аргумент, то допускаются оба варианта.

**Перегрузка функций**

Swift позволяет перегружать функции (overloading).

func say(what: String){}

func say(what: Int){}

Если вы имеете один и тот же список входных параметров (их имена и типы идентичны), то для перегрузки необходимо, чтобы функции имели различные типы выходных значений.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 4.3**

1. *Напишите функцию sum с переменным числом аргументов типа Double, которая суммирует все переданные значения.*
2. *Напишите функцию average с переменным числом аргументов типа Double, которая вызывает функцию sum и считает среднее переданных чисел.*
3. *Напишите функцию с inout параметром, которая меняет знак аргумента на противоположный.*
4. *Напишите функцию doWithTwo с тремя аргументами: первый функциональный тип (Double...) -> Double, два других типа Double, возвращаемое значение Double.*

func doWithTwo (doFunc: (Double...) -> Double, arg1: Double, arg2: Double) -> Double

*Функция возвращает результат выполнения переданной в качестве параметра функции над вторым и третьим параметрами.*

1. *Напишите функцию operation, которая принимает в качестве параметра строку, а возвращает функциональный тип:*

func operation (which: String) -> (Double...) -> Double

*В функции проверяется строка с названием операции (сложение или вычисление среднего (см. имена функций п.1. и п.2)) и возвращается соответствующая функция.*

1. *Напишите функцию doOperation (или сделайте перегрузку предыдущей функции) со следующим заголовком*

func doOperation(operation: String) -> ((Double, Double) -> Double)?

*Так же как в предыдущей функции она принимает строку содержащую операцию: “+” , “-“ , “/” , “\*” и возвращает одну из четырех локальных (вложенных) функций от двух аргументов. Предусмотрите случай, когда введена не верная операция – в этом случае должно возвращаться nil.*

**Вспомогательные функциональные элементы**

**Метод map(\_:)**

Данный метод позволяет применить переданное в него замыкание для каждого элемента коллекции. В результате его выполнения возвращается новая последовательность, тип элементов которой может отличаться от типа исходных элементов

var myArray = [2, 4, 5, 7]

var newArray = myArray.map{$0}

newArray // [2, 4, 5, 7]

Метод map(\_:) принимает замыкание и применяет его к каждому элементу массива myArray. Переданное замыкание {$0} не производит каких-либо действий над элементами, поэтому результат, содержащийся в переменной newArray, не отличается от исходного.

Если не использовать сокращенный синтаксис, то вызов метода будет выглядеть следующим образом:

var array = [2, 4, 5, 7]

let newArray = array.map({

(value: Int) -> Int in

return value

})

Оптимизируем замыкание. Сократим код перед ключевым словом in, так как передаваемое замыкание имеет всего один входной аргумент уберем круглые скобки, так как метод map(\_:) имеет один входной аргумент уберем оператор return, так как тело замыкания помещается в одно выражение. В результате получим следующий код:

var array = [2, 4, 5, 7]

let newArray = array.map{value in value}

Теперь можно убрать ключевое слово in и заменить value на сокращенное имя $0:

var array = [2, 4, 5, 7]

let newArray = array.map{$0}

Как говорилось ранее, тип значений результирующей последовательности может отличаться от типа элементов исходной последовательности

var array = [1, 2, 3, 4]

var boolArray = array.map{$0 > 2}

boolArray // [false, false, true, true]

Каждый элемент последовательности сравнивается с двойкой, в результате чего возвращается логическое значение. Другой особенностью map(\_:) является возможность создания многомерных массивов

let numbers = [1, 2, 3, 4]

let mapped = numbers.map { Array(repeating: $0, count: $0) }

mapped // [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]

Помимо массивов, метод map(\_:) может быть применен к словарям.

**Метод mapValues(\_:)**

Он поочередно применяет переданное ему замыкание к значению каждого элемента словаря

var mappedCloseStars = ["Centauri": 4.24, "Alpha A": 4.37, "Alpha B": 4.37]

let newCollection = mappedCloseStars.mapValues{ $0+1 }

newCollection // ["Centauri": "5.24", "Alpha A": "5.37", "Alpha B": "5.37"]

В результате вы получаете все тот же словарь, но с обработанными в соответствии с замыканием значениями.

**Метод filter(\_:)**

Метод filter(\_:) используется, когда требуется отфильтровать элементы коллекции по определенному правилу:

let numbers = [1, 4, 10, 15]

let even = numbers.filter{ $0 % 2 == 0 }

even // [4, 10]

Помимо массивов, вы можете производить фильтрацию и других типов коллекций.

**Метод reduce(\_:\_:)**

Метод reduce(\_:\_:) позволяет объединить все элементы коллекции в одно значение в соответствии с переданным замыканием. Помимо самих элементов, метод принимает первоначальное значение, которое служит для выполнения операции с первым элементом коллекции. Предположим, необходимо определить общее количество имеющихся у вас денег. На вашей карте 210 рублей, а в кошельке 4 купюры разного достоинства. Эта задача легко решается с помощью метода reduce(\_:\_:)

let cash = [10, 50, 100, 500]

let total = cash.reduce(210, +) // 870

Первый аргумент — это начальное значение, второй — замыкание, обрабатывающее каждую пару элементов. Первая операция сложения производится между начальным значением и первым элементом массива cash. Результат этой операции складывается со вторым элементом массива и т.д.

**Метод flatMap(\_:)**

Метод flatMap(\_:) отличается от map(\_:) тем, что всегда возвращает плоский одномерный массив.

let numbers = [1, 2, 3, 4]

let flatMapped = numbers.flatMap { Array(repeating: $0, count: $0) }

flatMapped // [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]

Вся мощь flatMap(\_:) проявляется тогда, когда в многомерном массиве требуется найти все подпадающие под некоторое условие значения

let someArray = [[1, 2, 3, 4, 5], [11, 44, 1, 6], [16, 403, 321, 10]]

let filterSomeArray = someArray.flatMap{$0.filter{ $0 % 2 == 0}}

filterSomeArray // [2, 4, 44, 6, 16, 10]

**Метод zip(\_:\_:)**

Функция zip(\_:\_:) предназначена для формирования последовательности пар значений, каждая из которых составлена из элементов двух базовых последовательностей. Сначала вы берете первые элементы каждой последовательности, группируете их, потом берете вторые элементы и т.д.

let collectionOne = [1, 2, 3]

let collectionTwo = [4, 5, 6]

var zipSequence = zip(collectionOne, //Zip2Sequence<Array<Int>

collectionTwo) //Array<Int>>

// генерация массива из сформированной последовательности

Array (zipSequence)// [(.0 1, .1 4), (.0 2, .1 5), (.0 3, .1 6)]

// генерация словаря на основе последовательности пар значений

let newDictionary = Dictionary (uniqueKeysWithValues: zipSequence)

//[2: 5, 3: 6, 1: 4]

**Метод sort ()**

Функция sort используется для сортировки. Принимает параметр by, который задает принцип сортировки. В примере eсли это значение равно true, то первое значение ставится до второго, если false - то после.

var numbers: [Int] = [10, 4, 12, 1, 3]

numbers.sort(by: {$0 > $1})

print(numbers)  // [12, 10, 4, 3, 1]

var nums = numbers.sorted(by: <)

print(nums)     // [1, 3, 4, 10, 12]

**Метод** forEach()

Вместо применения цикла также можно использовать метод forEach(), который перебирает все элементы. В качестве параметра этот метод принимает функцию, которая производит действия над текущим элементом:

var numbers: [Int] = [1, 2, 3, 4, 5, 6, 7, 8]

numbers.forEach({print($0)})

**Ленивые вычисления**

Хорошо оптимизированные программы практически всегда используют lazy-вычисления. В программировании *ленивыми* называются такие элементы, вычисление значений которых откладывается до момента обращения к ним. Таким образом, пока значение не потребуется и не будет использовано, оно будет храниться в виде исходных данных. С помощью ленивых вычислений достигается экономия процессорного времени, то есть компьютер не занимается ненужными в данный момент вычислениями. Существует два типа ленивых элементов:

lazy-by-name — значение элемента вычисляется при каждом доступе к нему;

lazy-by-need — элемент вычисляется один раз при первом обращении к нему, после чего фиксируется и больше не изменяется.

Swift позволяет работать с обоими типами ленивых элементов, но в строгом соответствии с правилами.

С помощью замыканий мы можем создавать ленивые конструкции типа lazy-by-name, значение которых высчитывается при каждом обращении к ним.

var arrayOfNames = ["Helena", "Baz", "Alex"]

arrayOfNames.count //3

let nextName = { arrayOfNames.remove(at: 0) }

arrayOfNames.count //3

nextName() // "Helena"

arrayOfNames.count// 2

В константе nextName хранится замыкание, удаляющее первый элемент массива arrayOfNames . Несмотря на то что константа объявлена, а ее значение проинициализировано, количество элементов массива не уменьшается до тех пор, пока ее значение не будет вызвано непосредственно. Это связано с тем, что замыкание, хранящееся в константе, — ленивое. Таким образом, можно сказать, что любая функция или метод являются lazy-by-name, так как их значение высчитывается при каждом обращении к ним.

**Свойство lazy**

Некоторые конструкции языка Swift (например, массивы и словари) имеют свойство lazy, позволяющее преобразовать их в ленивые. Наиболее часто это происходит, когда существуют цепочки вызова свойств или методов и выделение памяти и вычисление промежуточных значений является бесполезной тратой ресурсов, так как эти значения никогда не будут использованы. Рассмотрим следующий пример: существует массив целых чисел, значения которого непосредственно не используются в работе программы. Вам требуется лишь результат его обработки методом map(\_:), и то не в данный момент, а позже

var myCollection = [1,2,3,4,5,6].lazy.map{$0 + 1}

//LazyMapRandomAccessCollection<Array<Int>, Int>

В результате выполнения возвращается ленивая коллекция. При этом память под отдельный массив целочисленных значений не выделяется, а вычисления метода map(\_:) не производятся до тех пор, пока не произойдет обращение к переменной myCollection. Вся прелесть такого подхода в том, что вы можете увеличивать цепочки вызовов, но при этом лишнего расхода ресурсов не будет

var myCollection = [1,2,3,4,5,6] // LazyFilterBidirectionalCollection

.lazy // <LazyMapRandomAccessCollection

.map{$0 + 1} // <Array<Int>, Int>>

.filter{$0 % 2 == 0}

Array(myCollection) // [2, 4, 6]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 4.4**

1. *Напишите замыкание, которое будет сортировать элементы массива по возрастанию и убыванию (используйте метод sort).*
2. *Напишите замыкание, которое будет отображать каждый элемент массива (в массиве могут быть nil) в строку, значение которой равно сумме цифр числа (если оно не nil).*

*Например: [23, 45] - > [“5”,”9”]*

1. *Напишите замыкание, которое будет преобразовывать двумерный массив целых в одномерный*

*[[1,2,3],[34,56],[1,78,5]] -> [1,2,3,34,56,1,78,5]*

1. *Используя Reduce напишите замыкание, которое находит максимальный элемент массива*

*[1,2,3] ->3*

**Вопросы**

1. Какие есть способы создания и инициализации массива?
2. Как сравнить массивы?
3. Как обратиться к диапазону массива?
4. Как создать множество?
5. Как получить доступ к элементам множества?
6. Какие есть способы определения словаря? Как изменить значения в словаре?
7. Какой синтаксис определения функции?
8. Можно ли при вызове функции опустить имя параметра?
9. Можно ли изменять значения параметров функции в теле функции? Почему?
10. Как задать вложенную функцию? Перечислите ее основные свойства.
11. Как можно вернуть функцию из функции? Приведите пример.
12. Что такое сквозные параметры функции? Как их можно использовать?
13. Приведите пример, функции которая в качестве аргумента принимает другую функцию.
14. Что такое замыкания? Приведите пример.
15. Для чего нужно каррирование функций? Как это можно выполнить?
16. Какие есть способы захвата внешних переменных в замыкании?
17. Что называется автозамыканием? Какие требования нужно выполнить, чтобы реализовать автозамыкание?
18. Для чего в замыкании используется атрибут @escaping?