Лабораторная работа №2

Тема занятия: «Использование языка программирования Swift: функции, замыкания, перечисления»

Цель: Выполнить разработку приложения с использованием языка программирования Swift: функции, замыкания, перечисления.

Время выполнения: 8 часа.

2.1 ФУНКЦИИ

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

Язык *Swift* предлагает достаточно гибкий единый синтаксис функций – от простых *C*-подобных функций без параметров до сложных методов в стиле *Objective-C* с локальными и внешними параметрами. Параметры могут служить как для простой инициализации значений внутри функции, так и для изменения внешних переменных после выполнения функции.

Каждая функция в *Swift* имеет тип, описывающий тип параметров функции и тип возвращаемого значения. Тип функции можно использовать аналогично любым другим типам в *Swift*, т. е. одна функция может быть параметром другой функции либо ее результирующим значением. Функции также могут вкладываться друг в друга, что позволяет инкапсулировать определенный алгоритм внутри локального контекста.

2.1.1 Объявление и вызов функций

При объявлении функции можно задать одно или несколько именованных типизированных значений, которые будут ее входными данными (или *параметрами*), а также тип значения, которое функция будет возвращать в качестве результата (или *возвращаемый тип*).

У каждой функции должно быть *имя*, которое отражает решаемую задачу. Чтобы воспользоваться функцией, ее нужно «вызвать», указав имя и входные значения (*аргументы*), соответствующие типам параметров этой функции. Аргументы функции всегда должны идти в том же порядке, в каком они были указаны при объявлении функции.

В приведенном ниже примере функция называется greet (person:), потому что это отражает ее задачу – получить имя пользователя и вежливо поздороваться. Для этого задается один входной параметр типа String под названием person, а возвращается тоже значение типа String, но уже содержащее приветствие:

```
func greet(person: String) -> String {
    let greeting = "Привет, " + person + "!"
    return greeting
}
```

Вся эта информация указана в *объявлении функции*, перед которым стоит ключевое слово func. Тип возвращаемого значения функции ставится после *результирующей стрелки* -> (это дефис и правая угловая скобка).

Из объявления функции можно узнать, что она делает, какие у нее входные данные и какой результат она возвращает. Объявленную функцию можно однозначно вызывать из любого участка кода:

```
print(greet(person: "Anna"))
// Выведет "Привет, Anna!"
print(greet(person: "Brian"))
// Выведет "Привет, Brian!"
```

Функция `greet(person:)` вызывается, принимая значение типа `String`, которое стоит после имени person, например вот так – `greet(person: "Anna")`. Поскольку функция возвращает значение типа String, вызов функции `greet(person:)` может быть завернут в вызов для функции `print(_:separator:terminator:)`, чтобы напечатать полученную строку и увидеть возвращаемое значение.

У `print(_:separator:terminator:)` нет ярлыка для первого аргумента, и его другие аргументы являются опциональными, поскольку имеют дефолтное значение (по умолчанию).

Тело функции `greet(person:)` начинается с объявления новой константы типа `String` под названием greeting, и устанавливается простое сообщение-приветствие. Затем это приветствие возвращается в точку вызова функции с помощью ключевого слова `return`. После выполнения оператора `return greeting` функция завершает свою работу и возвращает текущее значение `greeting`.

Функцию `greet(person:)` можно вызывать многократно и с разными входными значениями. В примере выше показано, что будет, если функцию вызвать с аргументом `"Anna"` и со значением `"Brian"`. В каждом случае функция возвратит персональное приветствие.

Чтобы упростить код этой функции, можно записать создание сообщения и его возврат в одну строку:

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
// Выведет "Hello again, Anna!"
```

2.2 ПАРАМЕТРЫ ФУНКЦИИ И ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

В языке *Swift* параметры функций и возвращаемые значения реализованы очень гибко. Разработчик может объявлять любые функции – от простейших, с одним безымянным параметром, до сложных, со множеством параметров и составными именами.

2.2.1 Функции без параметров

В некоторых случаях функции могут не иметь входных параметров. Вот пример функции без входных параметров, которая при вызове всегда возвращает одно и то же значение типа `String`:

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// Выведет "hello, world"
```

Обратите внимание, что несмотря на отсутствие параметров, в объявлении функции все равно нужно ставить скобки после имени. При вызове после имени функции также указываются пустые скобки.

2.2.2 Функции с несколькими входными параметрами

У функции может быть несколько параметров, которые указываются через запятую в скобках.

Эта функция принимает два параметра: имя человека и булево значение, приветствовали ли его уже, и возвращает соответствующее приветствие для этого человека:

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Выведет "Hello again, Tim!"
```

Вы вызываете функцию `greet(person:alreadyGreeted:)`, передавая значение типа String параметру с ярлыком person и булево значение с ярлыком `alreadyGreeted`, взятое в скобки через запятую. Обратите внимание, что эта функция отличается от функции `greet(person:)`, которую вы видели в предыдущем разделе. Хотя имена обеих функций начинаются с `greet`, функция `greet(person:alreadyGreeted:)` принимает два аргумента, а `greet(person:)` принимает только один.

2.2.3 Функции, не возвращающие значения

В некоторых случаях функции могут не иметь возвращаемого типа. Вот другая реализация функции `greet(person:)`, которая выводит свое собственное значение типа `String`, но не возвращает его:

```
func greet(person: String) {
    print("Привет, \(person)!")
}
greet(person: "Dave")
// Выведет "Привет, Dave!"
```

Так как у функции нет выходного значения, в ее объявлении отсутствует результирующая стрелка (->) и возвращаемый тип.

Строго говоря, функция `greet(person:)` все же возвращает значение, хотя оно нигде и не указано. Функции, для которых не задан возвращаемый тип, получают специальный тип `Void`. По сути, это просто пустой кортеж, т. е. кортеж с нулем элементов, который записывается как ().

Выходное значение функции может быть игнорировано:

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")
// Выведет "hello, world" и возвращает значение 12
printWithoutCounting(string: "hello, world")
// Выведет "hello, world", но не возвращает значения
```

Первая функция, `printAndCount(string:)` выводит строку, а затем возвращает подсчет символов в виде целого `(Int)`. Вторая функция, `printWithoutCounting(string:)` вызывает первую, но игнорирует ее возвращаемое значение. При вызове второй функции первая функция по-прежнему печатает сообщение, но ее возвращаемое значение не используется.

Хотя возвращаемые значения можно игнорировать, функция все же должна возвратить то, что задано в ее объявлении. Функция, для которой указан возвращаемый тип, не может заканчиваться оператором, который ничего не возвращает, иначе произойдет ошибка во время компиляции.

2.2.4 Функции, возвращающие несколько значений

Вы можете использовать кортежный тип в качестве возвращаемого типа для функции для возврата нескольких значений в виде составного параметра.

В следующем примере объявлена функция `minMax(array:)`, которая ищет минимальный и максимальный элементы в массиве типа Int:

```
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)
}
```

Функция `minMax(array:)` возвращает кортеж из двух значений типа Int. Этим значениям присвоены имена min и max, чтобы к ним можно было обращаться при запросе возвращаемого типа функции.

Тело функции `minMax(array:)` начинается с инициализации двух рабочих переменных `currentMin` и `currentMax` значением первого целого элемента в массиве. Затем функция последовательно проходит по всем остальным значениям в массиве и сравнивает их со значениями `currentMin` и `currentMax` соответственно. И наконец, самое маленькое и самое большое значения возвращаются внутри кортежа типа `Int`.

Так как имена элементов кортежа указаны в возвращаемом типе функции, к ним можно обращаться через точку и считывать значения:

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Выведет "min is -6 and max is 109"
```

Обратите внимание, что элементам кортежа не нужно давать название в момент возвращения кортежа из функции, так как их имена уже указаны как часть возвращаемого типа функции.

2.2.5 Опциональный кортеж как возвращаемый тип

Если возвращаемый из функции кортеж может иметь «пустое значение», то его следует объявить как *опциональный кортеж*, т. е. кортеж, который может равняться nil. Чтобы сделать возвращаемый кортеж опциональным, нужно поставить вопросительный знак после закрывающей скобки: `(Int, Int)?` или `(String, Int, Bool)?`.

Кортеж-опционал вида `(Int, Int)?` – это не то же самое, что кортеж, содержащий опционалы: `(Int?, Int?)`. Кортеж-опционал сам является опционалом, но не обязан состоять из опциональных значений.

Функция `minMax(array:)` выше возвращает кортеж из двух значений типа `Int`, однако не проверяет корректность передаваемого массива. Если аргумент array содержит пустой массив, для которого `count` равно 0, функция `minMax` в том виде, в каком она приведена выше, выдаст ошибку выполнения, когда попытается обратиться к элементу `array[0]`.

Для устранения этого недочета перепишем функцию `minMax(array:)` так, чтобы она возвращала кортеж-опционал, который в случае пустого массива примет значение nil:

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
```

```
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Чтобы проверить, возвращает ли эта версия функции `minMax(array:)` фактическое значение кортежа или `nil`, можно использовать привязку опционала:

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
// Выведет "min is -6 and max is 109"
```

2.2.6 Функции с неявным возвращаемым значением

Если тело функции состоит из единственного выражения, то функция неявно возвращает это выражение. Например, обе функции в примере ниже имеют одно и то же поведение:

```
func greeting(for person: String) -> String {
    "Привет, " + person + "!"
}
print(greeting(for: "Дейв"))
// Выведет "Привет, Дейв!"

func anotherGreeting(for person: String) -> String {
    return "Привет, " + person + "!"
}
print(anotherGreeting(for: "Дейв"))
// Выведет "Привет, Дейв!"
```

Поведение функции `greeting(for:)` заключается в том, чтобы просто вернуть приветственное сообщение, что означает, что мы можем использовать сокращенную запись этой функции. Функция `anotherGreeting(for:)` возвращает то же самое приветственное сообщение, используя ключевое слово return. Таким образом, если вы пишите функцию, которая состоит из одного лишь возвращаемого значения, то вы можете опустить слово return.

Код, который вы написали с неявным возвращаемым значением должен иметь это самое возвращаемое значение. Например, вы не можете использовать `print(13)` как неявное возвращаемые значения. Однако вы можете использовать функцию, которая никогда не возвращает значение,

например, `fatalError("Oh no!")`, в качестве неявного возвращаемого значения, потому что Swift знает, что неявного возврата не происходит.

2.3 ЯРЛЫКИ АРГУМЕНТОВ И ИМЕНА ПАРАМЕТРОВ ФУНКЦИЙ

Каждый параметр функции имеет *ярлык аргумента* и *имя параметра*. Ярлык аргумента используется при вызове функции. Каждый параметр при вызове функции записывается с ярлыком аргумента, стоящим перед ним. Имя параметра используется при реализации функции. По умолчанию параметры используют имена их параметров в качестве ярлыка аргумента.

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // Внутри тела функции firstParameterName и secondParameterName
    // ссылаются на значения аргументов, первого и второго параметров.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

Все параметры должны иметь уникальные имена. Несмотря на то, что несколько параметров могут иметь один ярлык аргумента, уникальные ярлыки аргумента метки помогают сделать ваш код более читабельным.

Вы пишете ярлык аргумента перед именем параметра через пробел:

```
func someFunction(argumentLabel parameterName: Int) {
    // В теле функции parameterName относится к значению аргумента
    // для этого параметра.
}
```

Вот вариант функции `greet(person:)`, которая принимает имя человека и его родной город, затем возвращает приветствие:

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Выводит "Hello Bill!  Glad you could visit from Cupertino."
```

Использование ярлыков аргументов позволяет функции вызываться в более выразительной манере, в виде предложения, при этом все же предоставляя тело функции в более читаемом виде и с более понятыми намерениями.

Если вы не хотите использовать имя параметра в качестве ярлыка аргумента – используйте подчеркивание (_) вместо явного ярлыка аргумента для этого параметра.

```
func someFunction(_ firstParameterName: Int, secondParameterName:
Int) {
    // В теле функции firstParameterName и secondParameterName
    // ссылаются на значения аргументов для первого и второго
параметров.
}
someFunction(1, secondParameterName: 2)
```

Если у параметра есть ярлык аргумента, то аргумент *должен* иметь ярлык при вызове функции.

2.3.1 Значения по умолчанию для параметров

При объявлении функции любому из ее параметров можно присвоить *значение по умолчанию*. Если у параметра есть значение по умолчанию, то при вызове функции этот параметр можно опустить.

```
func          someFunction(parameterWithoutDefault:          Int,
parameterWithDefault: Int = 12) {
    // Если вы пропускаете второй аргумент при вызове функции, то
    // значение parameterWithDefault будет равняться 12 внутри
тела функции.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6)
// parameterWithDefault равен 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault
равен 12
```

Расположите параметры, у которых нет дефолтных значений в начале списка параметров функции до параметров с дефолтными значениями. Параметры, не имеющие значения по умолчанию, как правило, более важны для значения функции - их запись в первую очередь облегчает распознавание функции уже вызванной ранее, независимо от того, опущены ли какие-то параметры по умолчанию.

2.3.2 Вариативные параметры

Вариативным называют параметр, который может иметь сразу несколько значений или не иметь ни одного. С помощью вариативного параметра можно передать в функцию произвольное число входных значений. Чтобы объявить параметр как вариативный, нужно поставить три точки `(...)` после его типа.

Значения, переданные через вариативный параметр, доступны внутри функции в виде массива соответствующего типа. Например, вариативный параметр numbers типа `Double`... доступен внутри функции в виде массива-константы numbers типа `[Double]`.

В приведенном ниже примере вычисляется *среднее арифметическое* (или же *среднее*) последовательности чисел, имеющей произвольную длину:

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// возвращает 3.0, что является средним арифметическим этих пяти чисел
arithmeticMean(3, 8.25, 18.75)
// возвращает 10.0, что является средним арифметическим этих трех чисел
```

Функции могут иметь несколько вариативных параметров. Первый параметр, который идет после вариативного параметра должен иметь ярлык аргумента. Ярлык аргумента позволяет однозначно определить, какие аргументы передаются вариативному, а какие - параметрам, которые идут после вариативного параметра.

2.3.3 Сквозные параметры

Параметры функции по умолчанию являются константами. Попытка изменить значение параметра функции из тела этой функции приводит к ошибке компиляции. Это означает, что вы не сможете изменить значение параметра по ошибке. Если вы хотите, чтобы функция изменила значение параметра, и вы хотите, чтобы эти изменения сохранились после того, как закончился вызов функции, определите этот параметр в качестве *сквозного параметра*.

Для создания сквозного параметра нужно поставить ключевое слово inout перед типом объявлением параметра. Сквозной параметр передает значение *в функцию*, которое затем изменяется в ней и возвращается *из функции*, заменяя исходное значение.

Вы можете передать только переменную в качестве аргумента для сквозного параметра. Вы не можете передать константу или значения литерала в качестве аргумента, так как константы и литералы не могут быть изменены. Вы ставите амперсанд (&) непосредственно перед именем переменной, когда передаете ее в качестве аргумента сквозного параметра, чтобы указать, что он может быть изменен с помощью функции.

Сквозные параметры не могут иметь значения по умолчанию, а вариативные параметры не могут быть сквозными, с ключевым словом inout.

Вот пример функции под названием `swapTwoInts(_:_:)`, у которой есть два сквозных целочисленных параметра – `a` и `b`:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
```

```
    a = b
    b = temporary
}
```

Функция `swapTwoInts(_:_:)` просто меняет значение переменной `b` на значение `a`, а значение `a` – на значение `b`. Для этого функция сохраняет значение `a` в локальной константе `temporaryA`, присваивает значение `b` переменной `a`, а затем присваивает значение `temporaryA` переменной `b`.

Вы можете вызвать функцию `swapTwoInts (_: _:)` с двумя переменными типа `Int`, чтобы поменять их значения. Обратите внимание, что имена `someInt` и `anotherInt` начинаются с амперсанда, когда они передаются в `swapTwoInts (_: _:)` функции:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now
\(anotherInt)")
// Выведет "someInt is now 107, and anotherInt is now 3"
```

В вышеприведенном примере видно, что исходные значения переменных `someInt` и `anotherInt` изменены функцией `swapTwoInts (_: _:)`, несмотря на то, что изначально они были объявлены за ее пределами.

Сквозные параметры – это не то же самое, что возвращаемые функцией значения. В примере с функцией `swapTwoInts` нет ни возвращаемого типа, ни возвращаемого значения, но параметры `someInt` и `anotherInt` все равно изменяются. Сквозные параметры – это альтернативный способ передачи изменений, сделанных внутри функции, за пределы тела этой функции.

2.4 ФУНКЦИОНАЛЬНЫЕ ТИПЫ

У каждой функции есть специальный *функциональный тип*, состоящий из типов параметров и типа возвращаемого значения.

Пример:

```
func addTwoInts(a: Int, _ b: Int) -> Int {
  return a + b
}
func multiplyTwoInts(a: Int, _ b: Int) -> Int {
  return a * b
}
```

В данном примере объявлены две простые математические функции – `addTwoInts` и `multiplyTwoInts`. Каждая из этих функций принимает два значения типа Int и возвращает одно значение типа Int, содержащее результат математической операции.

Обе функции имеют тип `(Int, Int) -> Int`. Эта запись означает следующее:

«функция с двумя параметрами типа `Int`, возвращающая значение типа `Int`».

Вот еще один пример, но уже функции без параметров и возвращаемого значения:

```
func printHelloWorld() {
  print("hello, world")
}
```

Эта функция имеет тип `() -> Void`, т. е. «функция без параметров, которая возвращает `Void`».

2.4.1 Использование функциональных типов

В *Swift* с функциональными типами можно работать так же, как и с другими типами. Например, можно объявить константу или переменную функционального типа и присвоить ей функцию соответствующего типа:

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

Эта запись означает следующее:

«Объявить переменную `mathFunction`, имеющую тип «функция, принимающая два значения типа `Int`, и возвращающая одно значение типа `Int`». Присвоить этой новой переменной указатель на функцию `addTwoInts`».

Функция `addTwoInts` имеет тот же тип, что и переменная mathFunction, поэтому с точки зрения языка *Swift* такое присваивание корректно.

Теперь функцию можно вызывать с помощью переменной `mathFunction`:

```
print("Result: \(mathFunction(2, 3))")
// Выведет "Result: 5"
```

Той же переменной можно присвоить и другую функцию такого же типа – аналогично нефункциональным типам:

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Выведет "Result: 6"
```

Как и в случае с любым другим типом, вы можете не указывать тип явно, а предоставить *Swift* самостоятельно вывести функциональный тип при присваивании функции константе или переменной:

```
let anotherMathFunction = addTwoInts
// для константы anotherMathFunction выведен тип (Int, Int) -> Int
```

2.4.2 Функциональные типы как типы параметров

Функциональные типы наподобие `(Int, Int) -> Int` могут быть типами параметров другой функции. Это позволяет определять некоторые аспекты реализации функции непосредственно во время ее вызова.

Следующий код печатает на экране результаты работы приведенных выше математических функций:

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int,
_ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Выведет "Result: 8"
```

В этом примере объявлена функция `printMathResult(_:_:_:)`, у которой есть три параметра. Первый параметр под названием mathFunction имеет тип `(Int, Int) -> Int`. Соответственно, аргументом этого параметра может быть любая функция такого же типа. Второй и третий параметры называются `a` и `b` и относятся к типу `Int`. Они служат для передачи двух входных значений для математической функции.

При вызове `printMathResult(_:_:_:)` получает в качестве входных данных функцию `addTwoInts(_:_:)` и два целочисленных значения 3 и 5. Затем она вызывает переданную функцию со значениями 3 и 5, а также выводит на экран результат 8.

Задача функции `printMathResult(_:_:_:)` заключается в том, чтобы печатать результат работы математической функции соответствующего типа. При этом конкретные детали этой математической функции не имеют значения – главное, чтобы она была подходящего типа. Все это позволяет безопасно управлять работой функции `printMathResult(_:_:_:)` непосредственно во время вызова.

2.4.3 Функциональные типы как возвращаемые типы

Функциональный тип можно сделать возвращаемым типом другой функции. Для этого нужно записать полный функциональный тип сразу же после возвратной стрелки (->) в возвращаемой функции.

В следующем примере объявлены две простые функции – `stepForward(_:)` и `stepBackward(_:)`. Функция `stepForward(_:)` возвращает входное значение, увеличенное на единицу, а функция `stepBackward(_:)` – уменьшенное на единицу. Обе функции имеют тип `(Int) -> Int`:

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

Следующая функция под названием `chooseStepFunction(backward:)` имеет возвращаемый тип `(Int) -> Int`. Функция `chooseStepFunction(backward:)` возвращает функцию `stepForward(_:)` или функцию `stepBackward(_:)` в зависимости от значения логического параметра `backward`:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

Теперь с помощью `chooseStepFunction(backward:)` можно получать функцию, которая будет сдвигать значение влево или вправо:

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue
> 0)
// moveNearerToZero ссылается на функцию stepBackward()
```

В предыдущем примере мы определяли, нужно ли прибавить или отнять единицу, чтобы последовательно приблизить переменную `currentValue` к нулю. Изначально `currentValue` имеет значение 3, т. е. сравнение `currentValue > 0` даст `true`, а функция `chooseStepFunction(backward:)`, соответственно, возвратит функцию `stepBackward(_:)`. Указатель на возвращаемую функцию хранится в константе `moveNearerToZero`.

Так как `moveNearerToZero` теперь ссылается на нужную функцию, можно использовать эту константу для отсчета до нуля:

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

2.5 Вложенные функции

Все ранее рассмотренные в этом разделе функции являются *глобальными*, т. е. определенными в глобальном контексте. Но помимо глобальных можно объявлять и функции, находящиеся внутри других функций, или же *вложенные*.

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

Приведенный выше пример с функцией `chooseStepFunction(backward:)` можно переписать со вложенными функциями:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue
> 0)
// moveNearerToZero теперь ссылается на вложенную функцию
stepForward()
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

2.6 ЗАМЫКАНИЯ

Замыкания – это самодостаточные блоки с определенным функционалом, которые могут быть переданы и использованы в вашем коде. Замыкания в Swift похожи на блоки в *C* и *Objective-C*, и лямбды в других языках программирования.

Замыкания могут захватывать и хранить ссылки на любые константы и переменные из контекста, в котором они объявлены. Эта процедура известна как *заключение* этих констант и переменных, отсюда и название *«замыкание»*. *Swift* выполняет всю работу с управлением памятью при захвате за вас.

Глобальные и вложенные функции, которые были представлены в главе Функции, являются частным случаем замыканий. Замыкания принимают одну из трех форм:

– глобальные функции являются замыканиями, у которых есть имя и которые не захватывают никакие значения;

– вложенные функции являются замыканиями, у которых есть имя и которые могут захватывать значения из включающей их функции;

– замыкающие выражения являются безымянными замыканиями, написанные в облегченном синтаксисе, которые могут захватывать значения из их окружающего контекста.

Замыкающие выражения в *Swift* имеют четкий, ясный, оптимизированный синтаксис в распространенных сценариях. Эти оптимизации включают:

– вывод типа параметра и возвращающего значения из контекста;
– неявные возвращающиеся значения однострочных замыканий;
– сокращенные имена параметров;
– синтаксис последующих замыканий.

2.6.1 Замыкающие выражения

Вложенные функции являются удобным способом для обозначения и объявления самоорганизованных блоков кода, которые являются частью более крупной функции. Тем не менее, иногда полезно писать короткие версии функциональных конструкций, без полного объявления и указания имени. Это особенно верно, когда вы работаете с функциями которые принимают другие функции в виде одного из своих параметров.

Замыкающие выражения, являются способом написания встроенных замыканий через краткий и специализированный синтаксис. Замыкающие выражения обеспечивают несколько синтаксических оптимизаций для написания замыканий в краткой форме, без потери ясности и намерений. Примеры замыкающих выражений ниже, показывают эти оптимизации путем рассмотрения метода `sorted(by:)` при нескольких итерациях, каждая из которых изображает ту же функциональность в более сжатой форме.

2.6.2 Метод sorted

В стандартной библиотеке *Swift* есть метод `sorted(by:)`, который сортирует массив значений определенного типа, основываясь на результате сортирующего замыкания, которые вы ему передадите. После завершения процесса сортировки, метод `sorted(by:)` возвращает новый массив того же типа и размера как старый, с элементами в правильном порядке сортировки. Исходный массив не изменяется методом `sorted(by:)`.

Примеры замыкающих выражений ниже используют метод `sorted(by:)`для сортировки массива из *String* значений в обратном алфавитном порядке. Вот исходный массив для сортировки:

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

Замыкание метода `sorted(by:)` принимает два аргумента одного и того же типа, что и содержимое массива, и возвращает `Bool` значение, которое решает поставить ли первое значение перед вторым, или после второго. Замыкание сортировки должно вернуть true, если первое значение должно быть *до* второго значения, и `false` в противном случае.

Этот пример сортирует массив из `String` значений, так что сортирующее замыкание должно быть функцией с типом `(String, String) -> Bool`.

Один из способов обеспечить сортирующее замыкание, это написать нормальную функцию нужного типа, и передать ее в качестве аргумента метода `sorted(by:)`:

```
func backward(_ s1: String, _ s2: String) -> Bool {
   return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames равен ["Ewa", "Daniella", "Chris", "Barry",
"Alex"]
```

Если первая строка (`s1`) больше чем вторая строка (`s2`), функция `backward(_:_:)` возвращает `true`, что указывает, что `s1` должна быть перед `s2` в сортированном массиве. Для символов в строках, «больше чем» означает «появляется в алфавите позже, чем». Это означает что буква «*B*» «больше чем буква «A», а строка «Tom» больше чем строка «Tim». Это делает обратную алфавитную сортировку, с «Barry» поставленным перед «Alex», и так далее.

Тем не менее, это довольно скучный способ написать то, что по сути, является функцией с одним выражением (a > b). В этом примере, было бы предпочтительнее написать сортирующее замыкание в одну строку, используя синтаксис замыкающего выражения.

2.6.3 Синтаксис замыкающего выражения

Синтаксис замыкающего выражения имеет следующую общую форму:

```
{ (параметры) -> тип результата in
выражения
}
```

Синтаксис замыкающего выражения может использовать сквозные параметры. Значения по умолчанию не могут быть переданы. Вариативные параметры могут быть использованы в любом месте в списке параметров. Кортежи также могут быть использованы как типы параметров и как типы возвращаемого значения.

Пример ниже показывает версию функции `backward(_:_:)` с использованием замыкающего выражения:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) ->
Bool in
```

```
    return s1 > s2
})
```

Обратите внимание, что объявление типов параметров и типа возвращаемого значения для этого однострочного замыкания идентично объявлению из функции `backward(_:_:)`. В обоих случаях, оно пишется в виде `(s1: String, s2: String) -> Bool`. Тем не менее, для однострочных замыкающих выражений, параметры и тип возвращаемого значения пишутся внутри фигурных скобок, а не вне их.

Начало тела замыкания содержит ключевое слово `in`. Это ключевое слово указывает, что объявление параметров и возвращаемого значения замыкания закончено, и тело замыкания вот-вот начнется.

Поскольку тело замыкания настолько короткое, оно может быть записано в одну строку:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 })
```

Это показывает, что общий вызов метода `sorted` остался прежним. Пара скобок по-прежнему обособляют весь набор параметров метода.

2.6.4 Определение типа из контекста

Поскольку сортирующее замыкание передается как аргумент метода, *Swift* может вывести типы его параметров и тип возвращаемого значения, через тип параметра метода `sorted(by:)`. Этот параметр ожидает функцию имеющую тип `(String, String) -> Bool`. Это означает что типы `(String, String)` и `Bool` не нужно писать в объявлении замыкающего выражения. Поскольку все типы могут быть выведены, стрелка результата (->) и скобки вокруг имен параметров также могут быть опущены:

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 })
```

Всегда можно определить типы параметров и тип возвращаемого значения, когда мы передаем замыкание функции в виде однострочного замыкающего выражения. В результате, когда замыкание используется как аргумент метода, вам никогда не нужно писать однострочное замыкание в его полном виде.

Тем не менее, вы всё равно можете явно указать типы, если хотите. И делать это предполагается, если это поможет избежать двусмысленности для читателей вашего кода. В случае с методом `sorted(by:)`, цель замыкания понятна из того факта, что сортировка происходит, и она безопасна для читателя, который может с уверенностью предположить, что замыкание, вероятно, будет работать со значениями `String`, поскольку оно помогает сортировать массив из строк.

2.6.5 Неявные возвращаемые значения из замыканий с одним выражением

Замыкания с одним выражением могут неявно возвращать результат своего выражения через опускание ключевого слова return из их объявления, как показано в этой версии предыдущего примера:

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 })
```

Здесь, функциональный тип аргумента метода `sorted(by:)`дает понять, что замыкание вернет `Bool` значение. Поскольку тело замыкания содержит одно выражение `(s1 > s2)`, которое возвращает `Bool` значение, то нет никакой двусмысленности, и ключевое слово return можно опустить.

2.6.6 Сокращенные имена аргументов

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

Если вы используете эти сокращенные имена параметров с вашим замыкающим выражением, вы можете пропустить список параметров замыкания из его объявления, а количество и тип сокращенных имен параметров будет выведено из ожидаемого типа метода. Ключевое слово in также может быть опущено, поскольку замыкающее выражение полностью состоит из его тела:

```
reversedNames = names.sorted(by: { $0 > $1 })
```

Здесь, $0 и $1 обращаются к первому и второму String параметру замыкания.

2.6.7 Операторные функции

Здесь есть на самом деле более короткий способ написать замыкающее выражение выше. Тип `String` в *Swift* определяет свою специфичную для строк реализацию оператора больше (>) как функции, имеющей два строковых параметра и возвращающей значение типа `Bool`. Это точно соответствует типу метода, для параметра метода `sorted(by:)`. Таким образом, вы можете просто написать оператор больше, а *Swift* будет считать, что вы хотите использовать специфичную для строк реализацию:

```
reversedNames = names.sorted(by: >)
```

2.6.8 Последующее замыкание

Если вам нужно передать замыкающее выражение в качестве последнего аргумента функции и само выражение замыкания длинное, то оно может быть записано в виде *последующего замыкания*. Последующее замыкание - замыкание, которое записано в виде замыкающего выражения вне (и после) круглых скобок вызова функции, даже несмотря на то, что оно все еще является аргументом функции. Когда вы используете синтаксис последующего замыкания, то вы не должны писать ярлык аргумента замыкания в качестве части вызова самой функции. Функция может включать в себя несколько последующих замыканий, однако, первые несколько примеров используют по одному последующему замыканию:

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // тело функции
}
```

Вот как вы вызываете эту функцию без использования последующего замыкания:

```
someFunctionThatTakesAClosure(closure: {
    // тело замыкания
})
```

Вот как вы вызываете эту функцию с использованием последующего замыкания:

```
someFunctionThatTakesAClosure() {
    // тело последующего замыкания
}
```

Сортирующее строки замыкание может быть записано вне круглых скобок функции `sorted(by:)`, как последующее замыкание:

```
reversedNames = names.sorted() { $0 > $1 }
```

Если выражение замыкания является единственным аргументом функции, и вы пишете его используя синтаксис последующего замыкания, то вы можете опустить написание круглых скобок вызова самой функции после ее имени.

```
reversedNames = names.sorted { $0 > $1 }
```

Последующие замыкания полезны в случаях, когда само замыкание достаточно длинное, и его невозможно записать в одну строку. В качестве примера приведем вам метод `map(_:)` типа `Array` в языке *Swift*, который принимает выражение замыкания как его единственный аргумент. Замыкание

вызывается по одному разу для каждого элемента массива и возвращает альтернативную отображаемую величину (возможно другого типа) для этого элемента. Природа отображения и тип возвращаемого значения определяется замыканием.

После применения замыкания к каждому элементу массива, метод `map(_:)` возвращает новый массив, содержащий новые преобразованные величины, в том же порядке, что и в исходном массиве.

Вот как вы можете использовать метод `map(_:)` вместе с последующим замыканием для превращения массива значений типа `Int` в массив типа `String`. Массив `[16, 58, 510]` используется для создания нового массива `["OneSix", "FiveEight", "FiveOneZero"]`:

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

Код выше создает словарь отображающий цифры и их английскую версию имен. Так же он объявляет массив целых значений для преобразования в массив строк.

Вы можете использовать массив `numbers` для создания значений типа `String`, передав замыкающее выражение в метод `map(_:)` массива в качестве последующего замыкания. Обратите внимание, что вызов `numbers.map` не включает в себя скобки после `map`, потому что метод `map(_:)` имеет только один параметр, который мы имеем в виде последующего замыкания:

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}

//тип строк был выведен как [String]
//значения ["OneSix", "FiveEight", "FiveOneZero"]
```

Метод `map(_:)` вызывает замыкание один раз для каждого элемента массива. Вам не нужно указывать тип входного параметра замыкания, `number`, так как тип может быть выведен из значений массива, который применяет метод map.

В этом примере переменная `number` инициализирована при помощи значения параметра замыкания number, так что значение может быть изменено внутри тела замыкания. (Параметры функций и замыкания всегда являются

константами.) Выражение замыкания так же определяет возвращаемый тип `String` для указания типа, который будет храниться в массиве на выходе из метода `map(_:)`.

Замыкающее выражение строит строку, названную output, каждый раз, когда оно вызывается. Оно рассчитывает последнюю цифру number, используя оператор деления с остатком `( number % 10 )` и использует затем эту получившуюся цифру, чтобы найти соответствующую строку в словаре `digitNames`. Это замыкание может быть использовано для создания строкового представления любого целого числа, большего чем 0.

Вызов словаря `digitNames` синтаксисом сабскрипта сопровождается знаком (!), потому что сабскрипт словаря возвращает опциональное значение, так как есть такая вероятность, что такого ключа в словаре может и не быть. В примере выше мы точно знаем, что `number % 10` всегда вернет существующий ключ словаря `digitNames`, так что восклицательный знак используется для принудительного извлечения значения типа `String` в возвращаемом опциональном значении сабскрипта.

Строка, полученная из словаря `digitNames`, добавляется в начало переменной `output`, путем правильного формирования строковой версии числа наоборот. (Выражение `number % 10` дает нам 6 для 16, 8 для 58 и 0 для 510).

Переменная number после вычисления остатка делится на 10. Так как тип значения `Int`, то наше число округляется вниз, таким образом 16 превращается в 1, 58 в 5, 510 в 51.

Процесс повторяется пока `number /= 10` не станет равным 0, после чего строка `output` возвращается замыканием и добавляется к выходному массиву функции `map(_:)`.

Использование синтаксиса последующих замыканий в примере выше аккуратно инкапсулирует функциональность замыкания сразу после функции `map(_:)`, которой замыкание помогает, без необходимости заворачивания всего замыкания внутрь внешних круглых скобок функции `map(_:)`.

Если функция принимает несколько последующих замыканий, вы можете пропустить ярлык параметра для первого из них, а для остальных уже указать нужно. Например, функция ниже загружает изображение в фотогалерею:

```
func loadPicture(from server: Server, completion: (Picture) -> Void, onFailure: () -> Void) {
    if let picture = download("photo.jpg", from: server) {
        completion(picture)
    } else {
        onFailure()
    }
}
```

Когда вы вызываете эту функцию для загрузки изображений, вы используете два замыкания. Первое замыкание – это обработчик, который отображает изображение после успешной загрузки. Второе замыкание – обработчик, который отображает ошибку пользователю, если произошла ошибка во время загрузки.

```
loadPicture(from: someServer) { picture in
    someView.currentPicture = picture
} onFailure: {
    print("Couldn't download the next picture.")
}
```

В этом примере метод `loadPicture(from:completion:onFailure:)` передает свою сетевую задачу в фоновый поток и вызывает одно из замыканий, когда сетевая задача выполнена. Написание функции таким способом позволяет вам разделять код, который ответственен за обработку ошибки во время сетевой задачи от кода, который отвечает за успешную загрузку.

2.7 ЗАХВАТ ЗНАЧЕНИЙ

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

В *Swift* самая простая форма замыкания может захватывать значения из вложенных функций, написанных внутри тела других функций. Вложенная функция может захватить любые значения из аргументов окружающей ее функции, а так же константы и переменные, объявленные внутри тела внешней функции.

Вот пример функции makeIncrementer, которая содержит вложенную функцию `incrementer`. Вложенная функция `incrementer()` захватывает два значения `runningTotal` и `amount` из окружающего контекста. После захвата этих значений incrementer возвращается функцией makeIncrementer как замыкание, которое увеличивает `runningTotal` на amount каждый раз как вызывается.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
   var runningTotal = 0
   func incrementer() -> Int {
      runningTotal += amount
      return runningTotal
   }
   return incrementer
}
```

Возвращаемый тип makeIncrementer `Void -> Int`. Это значит, что он возвращает *функцию*, а не простое значение. Возвращенная функция не имеет параметров и возвращает Int каждый раз как ее вызывают.

Функция `makeIncrementer(forIncrement:)` объявляет целочисленную переменную `runningTotal`, для хранения текущего значения инкрементора, которое будет возвращено. Переменная инициализируется значением 0.

Функция `makeIncrementer(forIncrement:)` имеет единственный параметр Int с внешним именем `forIncrement` и локальным именем amount. Значение аргумента передается этому параметру, определяя на сколько должно быть увеличено значение `runningTotal` каждый раз при вызове функции.

Функция makeIncrementer объявляет вложенную функцию `incrementer`, которая непосредственно и занимается увеличением значения. Эта функция просто добавляет `amount` к `runningTotal` и возвращает результат.

Если рассматривать функцию `incrementer()` отдельно, то она может показаться необычной:

```
func incrementer() -> Int {
    runningTotal += amount
    return runningTotal
}
```

Функция `incrementer()` не имеет ни одного параметра и она ссылается на runningTotal и amount внутри тела функции. Она делает это, захватывая существующие значения от `runningTotal` и amount из окружающей функции и используя их внутри. Захват ссылки дает гарантию того, что `runningTotal` и amount не исчезнут при окончании вызова makeIncrementer и гарантирует, что `runningTotal` останется переменной в следующий раз, когда будет вызвана функция `incrementer()`.

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

Приведем пример makeIncrementer в действии:

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

Этот пример заставляет константу `incrementByTen` ссылаться на функцию инкрементора, которая добавляет 10 к значению переменной `runningTotal` каждый раз как вызывается. Многократный вызов функции показывает ее в действии:

```
incrementByTen()
```

```
// возвращает 10
incrementByTen()
// возвращает 20
incrementByTen()
// возвращает 30
```

Если вы создаете второй инкрементор, он будет иметь свою собственную ссылку на новую отдельную переменную `runningTotal` :

```
let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()
//возвращает значение 7
```

Повторный вызов первоначального инкрементора `( incrementByTen )` заставит увеличиваться его собственную переменную `runningTotal` и никак не повлияет на переменную, захваченную в `incrementBySeven` :

```
incrementByTen()
//возвращает 40
```

Если вы присваиваете замыкание свойству экземпляра класса, и замыкание захватывает этот экземпляр по ссылке на него или его члены, вы создаете сильные обратные связи между экземпляром и замыканием. *Swift* использует списки захвата, для разрыва этих сильных обратных связей.

2.7.1 Замыкания – ссылочный тип

В примере выше `incrementBySeven` и `incrementByTen` константы, но замыкания, на которые ссылаются эти константы имеют возможность увеличивать значение переменных `runningTotal`, которые они захватили. Это из-за того, что функции и замыкания являются *ссылочными типами*.

Когда бы вы ни присваивали функцию или замыкание константе, или переменной, вы фактически присваиваете ссылку этой константе или переменной на эту функцию или замыкание. В примере выше выбор замыкания, на которое ссылается `incrementByTen`, константа, но не содержимое самого замыкания.

Это так же значит, что если вы присвоите замыкание двум разным константам или переменным, то оба они будут ссылаться на одно и то же замыкание:

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
//возвращает 50

incrementByTen()
//возвращает 60
```

Пример выше показывает, что вызов `alsoIncrementByTen` то же самое, что и вызов `incrementByTen`. Потому что и та и другая функция ссылаются на одно и то же замыкание: и то, и другое замыкание возвращают один и тот же `runningTotal`.

2.7.2 Сбегающие замыкания

Когда говорят, что замыкание *сбегает* из функции, то это значит, что это замыкание было передано в функцию в качестве аргумента и вызывается уже после того, как функция вернула значение. Когда вы объявляете функцию, которая имеет замыкание в качестве одного из параметров, то вы пишете `@escaping` до типа параметра, для того чтобы указать, что замыкание может *сбежать*.

Если замыкание хранится в переменной, которая была объявлена вне функции, а затем эта переменная была передана в качестве аргумента в функцию, то получается, что замыкание, которое посредством переменной передается в функцию, сбегающее. В качестве примера можно рассмотреть функции, которые выполняют асинхронные операции в завершающем обработчике, который является замыканием. То есть получается, что функция завершает свою работу, после чего вызывается завершающий обработчик. Или другими словами обработчик не вызывается, пока не завершится работа функции, таким образом получается, что данному замыканию нужно сбежать из области работы функции, чтобы отработать позже. Например:

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: @escaping
() -> Void) {
  completionHandlers.append(completionHandler)
}
```

Функция `someFunctionWithEscapingClosure(_:)` принимает и добавляет в массив замыкание, объявленное за пределами функции. Если вы не поставите маркировку `@escaping`, то получите ошибку компиляции.

Сбегающее замыкание, которое имеет ссылку на `self` требует отдельного рассмотрения, если `self` ссылается на экземпляр класса. Захватывая `self` в сбегающем замыкании, вы можете случайно создать зацикленность сильных ссылок.

Обычно замыкание захватывает переменные неявно, просто используя их внутри тела, но в случае с `self` вам нужно делать это явно. Если вы хотите захватить `self`, напишете `self` явно, когда используете его, или включите `self` в лист захвата замыкания. Когда вы пишете self явно, вы явно указываете свое намерение, а так же помогаете сами себе тем, что напоминаете проверить наличие цикла сильных ссылок. Например, в коде ниже замыкание переданное в метод `someFunctionWithEscapingClosure(_:)` ссылается на `self` явно. А вот замыкание, переданное в метод

`someFunctionWithNoneEscapingClosure(_:)` является несбегающим, что значит, что оно может ссылаться на self неявно.

```
func someFunctionWithNoneEscapingClosure(closure: () -> Void) {
    closure()
}

class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNoneEscapingClosure { x = 200 }
    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// Выведет "200"

completionHandlers.first?()
print(instance.x)
// Выведет "100"
```

Ниже приведена версия `doSomething()`, которая захватывает `self`, включая его в лист захвата замыкания, а затем неявно ссылается на него:

```
class SomeOtherClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { [self] in x = 100 }
        someFunctionWithNoneEscapingClosure { x = 200 }
    }
}
```

Если `self` является экземпляром структуры или перечисления, то вы можете всегда ссылаться на `self` неявно. Однако, сбегающие замыкания не могут захватить изменяемую ссылку на `self`, когда `self` является экземпляром структуры или перечисления. Структуры и перечисления не допускают общей изменчивости.

```
struct SomeStruct {
    var x = 10
    mutating func doSomething() {
        someFunctionWithNoneEscapingClosure { x = 200 }  // Ok
        someFunctionWithEscapingClosure { x = 100 }     // Error
    }
}
```

Вызов функции `someFunctionWithEscapitngClosure` в примере выше вызовет ошибку, так как находится замыкание внутри mutable метода, таким образом `self` так же получается изменяемым (*mutable*). Ошибка получается из-за того, что мы нарушаем правило, которое гласит, что в структурах сбегающие замыкания не могут захватывать изменяемую ссылку на `self`.

2.8 АВТОЗАМЫКАНИЯ (AUTOCLOSURES)

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

Нет ничего необычного в *вызове* функций, которые принимают автозамыкания, но необычным является *реализовывать* такие функции. Например, функция `assert(condition:message:file:line:)` принимает автозамыкания на место `condition` и `message` параметров. Ее параметр condition вычисляется только в сборке дебаггера, а параметр `message` вычисляется, если только `condition` равен `false`.

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

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry",
"Daniella"]
print(customersInLine.count)
// Выведет "5"

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// Выведет "5"

print("Now serving \(customerProvider())!")
// Выведет "Now serving Chris!"
print(customersInLine.count)
// Выведет "4"
```

Даже если первый элемент массива `customersInLine` удаляется кодом внутри замыкания, элемент массива фактически не удаляется до тех пор, пока само замыкание не будет вызвано. Если замыкание так и не вызывается, то выражение внутри него никогда не выполнится и, соответственно, элемент не

будет удален из массива. Обратите внимание, что `customerProvider` является не `String`, а `() -> String`, то есть функция не принимает аргументов, но возвращает строку. Вы получите то же самое поведение, когда сделаете это внутри функции:

```
// customersInLine равен ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: { customersInLine.remove(at: 0) } )
// Выведет "Now serving Alex!"
```

Функция `serve(customer:)` описанная выше принимает явное замыкание, которое возвращает имя клиента. Версия функции `serve(customer:)` ниже выполняет ту же самую операцию, но вместо использования явного замыкания, она использует автозамыкание, поставив маркировку при помощи атрибута `@autoclosure`. Теперь вы можете вызывать функцию, как будто бы она принимает аргумент `String` вместо замыкания. Аргумент автоматически преобразуется в замыкание, потому что тип параметра `customerProvider` имеет атрибут `@autoclosure`.

```
// customersInLine равен ["Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String)
{
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))
// Выведет "Now serving Ewa!"
```

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

Если вы хотите чтобы автозамыкание могло сбежать, то вам нужно использовать оба атрибута и `@autoclosure`, и `@escaping`.

```
// customersInLine равен ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(_ customerProvider: @autoclosure
@escaping () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))

print("Collected \(customerProviders.count) closures.")
// Выведет "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
```

```
// Выведет "Now serving Barry!"
// Выведет "Now serving Daniella!"
```

В коде выше, вместо того, чтобы вызывать переданное замыкание в качестве аргумента customer, функция `collectCustomerProviders(_:)` добавляет замыкание к массиву `customerProviders`. Массив объявлен за пределами функции, что означает, что замыкание в массиве может быть исполнено после того, как функция вернет значение. В результате значение аргумента `customerProvider` должен иметь «разрешение» на «побег» из зоны видимости функции.

2.9 ПЕРЕЧИСЛЕНИЯ

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

Если вы знакомы с *C*, то вы знаете, что перечисления в *C* присваивают соответствующие имена набору целочисленных значений. Перечисления в *Swift* более гибкий инструмент и не должны предоставлять значения для каждого члена перечисления. Если значение (известное как «сырое» значение) предоставляется каждому члену перечисления, то это значение может быть строкой, символом или целочисленным значением, числом с плавающей точкой.

Кроме того, членам перечисления можно задать соответствующие значения любого типа, которые должны быть сохранены вместе с каждым кейсом перечисления. Вы можете определить общий набор соответствующих значений как часть одного перечисления, каждый из которых будет иметь разные наборы значений ассоциативных типов, связанных с ними.

Перечисления в *Swift* – типы «первого класса». Они обладают особенностями, которые обычно поддерживаются классами, например, вычисляемые свойства, для предоставления дополнительной информации о текущем значении перечисления, методы экземпляра для дополнительной функциональности, относящейся к значениям, которые предоставляет перечисление.

Перечисления так же могут объявлять инициализаторы для предоставления начального значения элементам. Они так же могут быть расширены для наращивания своей функциональности над её начальной реализацией. Могут соответствовать протоколам для обеспечения стандартной функциональности.

2.9.1 Синтаксис перечислений

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

```
enum SomeEnumeration {
    //здесь будет объявление перечисления
}
```

Ниже пример с четырьмя сторонами света:

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

Значения, объявленные в перечислении *(north, south, east, и west)*, называются *кейсами перечисления*. Используйте ключевое слово case для включения нового кейса перечисления.

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

Множественные значения члена перечисления могут записываться в одну строку, разделяясь между собой запятой:

```
enum Planet {
    case mercury, venus, earth, mars, jupiter, saturn, uranus,
neptune
}
```

Каждое объявление перечисления объявляет и новый тип. Как и остальные типы в *Swift*, их имена (к примеру, *CompassPoint* и *Planet*) должны начинаться с заглавной буквы. Имена перечислениям лучше давать особенные, а не те, которые вы можете использовать в нескольких местах, так чтобы они читались как само собой разумеющиеся:

```
var directionToHead = CompassPoint.west
```

Тип *directionToHead* выведен при инициализации одного из возможных значений *CompassPoint*. Если *directionToHead* объявлена как *CompassPoint*, то можем использовать различные значения *CompassPoint* через сокращенный точечный синтаксис:

```
directionToHead = .east
```

Тип *directionToHead* уже известен, так что вы можете не указывать тип, присваивая значения. Так делается для хорошо читаемого кода, когда работаете с явно указанными типами значений перечисления.

2.9.2 Использование перечислений с инструкцией switch

Вы можете сочетать индивидуальные значения перечисления с инструкцией `switch`:

```
directionToHead = .south
switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}
// Выводит "Watch out for penguins"
```

Вы можете прочитать этот код как:

«Рассмотрим значение `directionToHead`. В случае, когда `directionToHead` равняется `.north`, выводится сообщение «Lots of planets have a north». В кейсе, где оно равняется `.south`, выводится сообщение «Watch out for penguins».

Оператор `switch` должен быть исчерпывающим, когда рассматриваются члены перечисления. Если мы пропустим `case .west`, то код не скомпилируется, так как не рассматривается полный перечень членов `CompassPoint`. Требования к конструкции быть исчерпывающей, помогает случайно не пропустить член перечисления.

Если не удобно описывать кейс для каждого члена перечисления, то вы можете использовать кейс `default`, для закрытия всех остальных вариантов перечисления:

```
let somePlanet = Planet.earth
switch somePlanet {
case .earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// Выведет "Mostly harmless"
```

2.9.3 Итерация по кейсам перечисления

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

```
 enum Beverage: CaseIterable {
    case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) beverages available")
// Выведет "3 beverages available"
```

В приведенном выше примере следует писать `Beverage.allCases` для доступа к коллекции, содержащей все кейсы перечисления `Beverage`. Можно использовать `allCases`, как и любую другую коллекцию – элементы коллекции являются экземплярами типа перечисления, поэтому в этом случае они являются значениями `Beverage`. В приведенном выше примере подсчитывается количество кейсов, приведенный ниже пример использует цикл `for` для итерации по всем кейсам.

```
for beverage in Beverage.allCases {
    print(beverage)
}
// coffee
// tea
// juice
```

Вопросы к лабораторной работе № 2

1 Что такое функция?
2 Как объявить функцию?
3 Как объявить функцию с несколькими параметрами?
4 Что такое опциональный кортедж?
5 Для чего служат ярлыки аргументов и имен параметров функции?
6 Что такое вариативные параметры?
7 Что такое сквозные параметры?
8 Для чего служат функциональные типы? Как создать свой функциональный тип?
9 Что такое вложенные функции?
10 Что такое замыкание? Для чего оно служит?
11 Что такое автозамыкание? Что такое сбегающее замыкание?
12 Как объявить перечисление? Для чего служат перечесления?

Задание к лабораторной работе № 2

Задание к лабораторной работе:

– согласно варианта задания написать программу (Вариант = номер в списке группы % 30 + 1);

– разбить функционал приложения на несколько пакетов придерживаясь логики.

– сделать валидацию всех вводимых значений.

Варианты

1 Дано натуральное число n. Выясните, можно ли представить данное число в виде произведения трех последовательных натуральных чисел.

2 Дано натуральное число m. Укажите все тройки натуральных чисел x, y и z, удовлетворяющие следующему условию: m = x3 + y3 + z3.

3 Среди всех четырехзначных номеров машин, определите количество номеров, содержащих только три одинаковые цифры.

4 Среди всех четырехзначных номеров машин, определите количество номеров, содержащих три или более одинаковых цифры.

5 Среди всех четырехзначных номеров машин, определите количество номеров машин, содержащих только две одинаковые цифры.

6 Среди всех четырехзначных номеров машин, определите количество номеров машин, содержащих только две или более одинаковых цифры.

7 Напишите программу нахождения, следующего за данным совершенного числа. Совершенным называется число, сумма делителей которого, не считая самого числа, равна этому числу. Первое совершенное число 6 (6 = 1 + 2 + 3).

8 Проверьте, является ли данное натуральное число простым.

9 Дано натуральное число P. Напишите программу нахождения всех натуральных чисел, не превосходящих P, которые можно представить в виде произведения двух простых чисел.

10 Дано натуральное число P, заданное в восмеричной системе счисления. Напишите программу перевода этого числа в двоичную систему счисления. Встроенный метод перевода не использовать.

11 Дано натуральное число P, заданное в шестнадцатеричной системе счисления. Переведите его в двоичную систему счисления. Встроенный метод перевода не использовать.

12 Дано натуральное число P. Найдите все «совершенные» числа, не превосходящие P. Совершенным, называется число, сумма делителей которого, не считая самого числа, равна этому числу. Первое совершенное число 6 (6 = 1 +2 + 3).

13 Дано натуральное n-значное число P. Проверьте, является ли

данное число палиндромом (перевертышем).

14 Дано натуральное число P. Проверьте, кратно ли P трем, используя признак делимости на 3.

15 Дано натуральное число P. Проверьте, кратно ли P одиннадцати, используя признак делимости на 11 (знакопеременная сумма его цифр делится на 11).

16 Дано натуральное число P. Найдите все простые числа, не превосходящие числа P.

17 Дано натуральное число P. Найдите все делители числа P.

18 Дано натуральное число P. Найдите сумму цифр числа P.

19 Дано натуральное число P. Выбросите из записи числа P цифры 0, оставив прежним порядок остальных цифр.

20 Дано натуральное число P. Проверьте, кратно ли число P девяти, используя признак делимости на 9.

21 Дана обыкновенная дробь n/m Сократите данную дробь.

22 Напишите программу сложения двух обыкновенных несократимых дробей n/m и q/p Результат представить в виде несократимой дроби.

23 Напишите программу вычитания двух обыкновенных несократимых дробей n/m и q/p Результат представить в виде несократимой дроби.

24 Напишите программу умножения двух обыкновенных несократимых дробей n/m и q/p Результат представить в виде несократимой дроби.

25 Напишите программу деления двух обыкновенных несократимых дробей n/m и q/p Результат представить в виде несократимой дроби.

26 Данное натуральное число N переведите из десятичной системы счисления в двоичную. Встроенный метод перевода не использовать.

27 Данное натуральное число N замените суммой квадратов его цифр. Произведите K таких замен.

28 Дано натуральное число n. Найдите все меньшие n числа Мерсена. Простое число называется числом Мерсена, если оно может быть представлено в виде 2p – 1, где p – тоже простое число.

29 Дано натуральное число N. Найдите все составные натуральные числа, меньшие N.

30 Дано число P, найти факториал числа P.

Задание 2. Используя функциональные типы, создайте программу согласно варианту.

1.

а) для сложения целых чисел;

б) для сложения комплексных чисел.

2.

а) для сложения вещественных чисел;

б) для сложения комплексных чисел.

3.

а) для умножения целых чисел;
б) для умножения комплексных чисел.
4.
а) для вычитания целых чисел;
б) для вычитания комплексных чисел.
5.
а) для умножения вещественных чисел;
б) для умножения комплексных чисел.
6.
а) для вычитания вещественных чисел;
б) для вычитания комплексных чисел.
7.
а) для деления целых чисел;
б) для деления комплексных чисел.
8.
а) по номеру года выдает его название по старояпонскому календарю;
б) по названию месяца выдает знак Зодиака.
9.
а) для сложения десятичных дробей;
б) для сложения обыкновенных дробей.
10.
а) для вычитания десятичных дробей;
б) для вычитания обыкновенных дробей.
11.
а) для умножения десятичных дробей;
б) для умножения обыкновенных дробей.
12.
а) для деления десятичных дробей;
б) для деления обыкновенных дробей.
13.
а) для преобразования десятичной дроби в обыкновенную;
б) для преобразования обыкновенной дроби в десятичную.
14.
а) для вычисления натурального логарифма;
б) для вычисления десятичного логарифма.
15.
а) целые числа возводит в степень n;
б) из десятичных чисел извлекает корень степени n.
16.
а) для перевода часов и минут в минуты;
б) для перевода минут в часы и минуты.
17.
а) для последовательности целых чисел находит среднее арифметическое;

б) для строки находит количество букв, содержащихся в ней.

18.

а) для последовательности целых чисел находит максимальный элемент;

б) для строки находит длину самого длинного слова.

19.

а) для последовательности целых чисел находит минимальный элемент;

б) для строки находит длину самого короткого слова.

20.

а) для последовательности целых чисел находит количество четных элементов;

б) для строки находит количество слов, начинающихся на букву «а».

21.

а) для последовательности целых чисел находит количество отрицательных элементов;

б) для строки находит количество слов, заканчивающихся и начинающихся на одну и ту же букву.

22.

а) для последовательности целых чисел находит количество нечетных

элементов;

б) для строки находит количество слов в ней.

23.

а) для последовательности, начинающейся на четное число, выполняет

циклический сдвиг влево на количество элементов равное первому элементу

последовательности.

б) для последовательности, начинающейся на нечетное число, выполняет

циклический сдвиг вправо на количество элементов равное последнему элементу

последовательности.

24.

а) для последовательности целых чисел удаляет все четные элементы из

последовательности;

б) для строки удаляет все четные слова.

25.

а) для последовательности удаляет все четные элементы;

б) для последовательности удаляет все элементы, заключенные между

двумя нулевыми элементами.

26.

а) для последовательности целых чисел находит количество простых чисел;

б) для строки находит количество слов, заканчивающихся и начинающихся на одну и ту же букву.

27.

а) для последовательности целых чисел находит количество нечетных элементов;

б) для строки находит количество пробелов в ней.

28.

а) для последовательности, начинающейся на четное число, выполняет обрезку всех элементов после четного числа.

б) для последовательности, начинающейся на нечетное число, выполняет обрезку всех элементов, начиная с 4-го символа после нечетного числа.

29.

а) для последовательности целых чисел удаляет все четные элементы из последовательности;

б) для строки удаляет все нечетные слова.

30.

а) для целого числа определять является ли оно полиндромом

б) для определить является ли она полиндромом

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание по вариантам (Вариант берётся согласно списку в группе), сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание с кодом, комментариями и скриншотами работающих программ.

3. Вывод по проделанной работе.