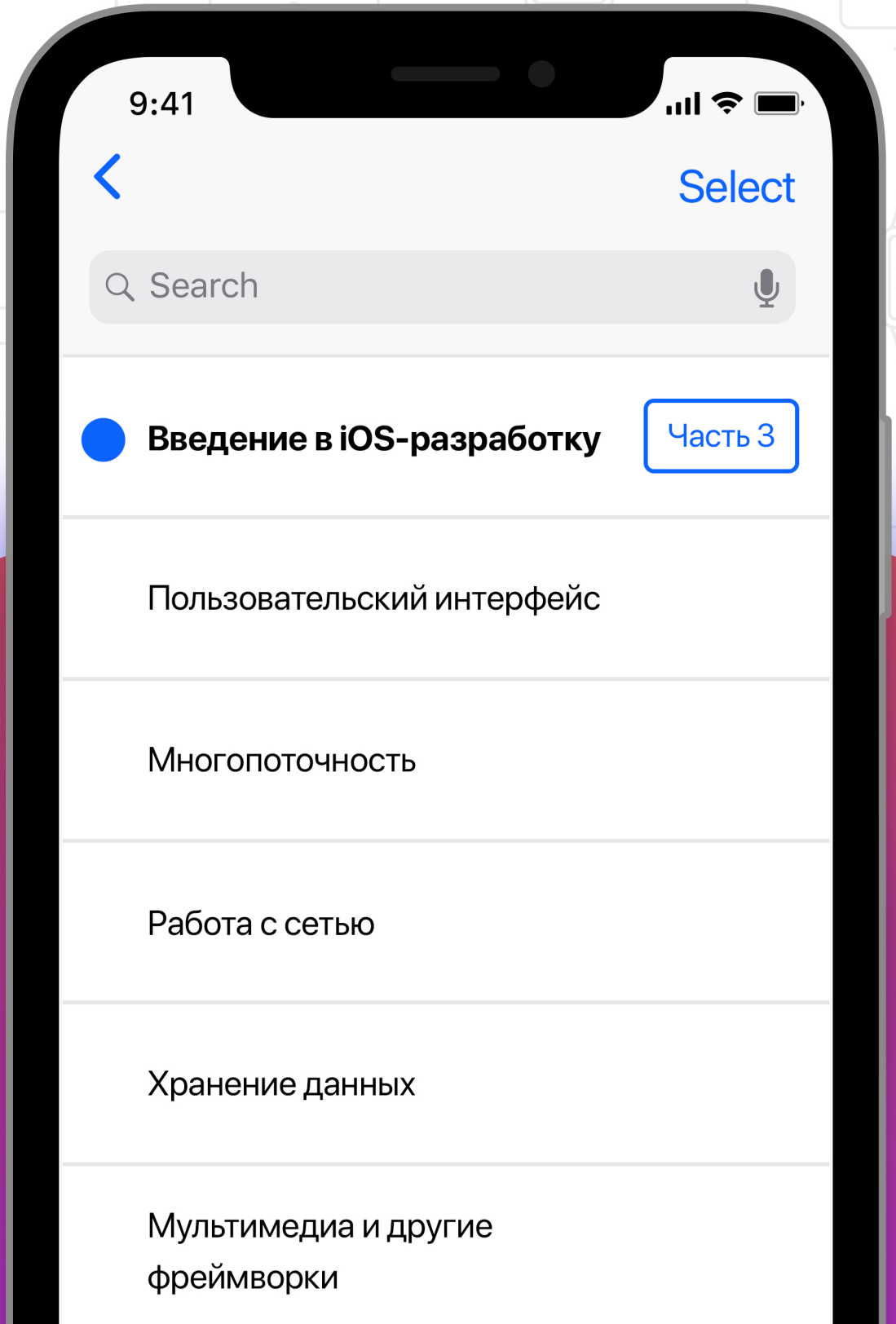
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

Программа iOS-разработчик

Конспект



Оглавление

Глава 4

Неделя 3

4.1. Замыкания

Замыкания похожи на блоки в $Objective-C$ или на лямбда функции из других языков программирования. Т.е. замыкания это просто блоки кода, которые можно передавать и использовать. Назвали их так потому, что они могут захватывать/завязываться на переменные и константы объявленные в их контексте.

Рассмотрим пример. Если бы внутренняя функция никак не обращалась к counter, то после выхода из внешней переменная была бы удалена из памяти. Однако в этом случае Swift сохраняет ее в памяти для того, чтобы увеличивать ее значение при вызове внутренней функции. Это и есть замыкание. При этом вам ничего не нужно дополнительно объявлять. Swift все управление памятью берет на себя.

```swift
func counterFunc() -> (Int) -> String {
    var counter = 0
    func innerFunc(i: Int) -> String {
        counter += i // counter захвачен в замыкании
        return "running total: \(counter)"
    }
    return innerFunc
}

let closureFromFunc = counterFunc()

print("\(closureFromFunc(3))") // running total: 3
print("\(closureFromFunc(3))") // running total: 6
print("\(closureFromFunc(3))") // running total: 9
```

Рассмотрим еще один пример замыкания. В этот раз оно будет анонимным. В Swift такие замыкания называются closure expression. Для его объявления в фигурных скобках указываются: параметры, перечисленные внутри круглых скобок, стрелка, тип возвращаемого значения и ключевое слово in. Так же как в обычной функции, но значения по умолчанию в этом случае указывать нельзя. Далее следует сам код замыкания. В нем мы можем обращаться к переданным параметрам по соответствующим именам. Также нам доступны переменные и константы объявленные во внешнем скоупе.

```
let multiplier = 3

let anonymousClusure = {
    (anotherItemInArray: Int) -> Int in
    return anotherItemInArray * multiplier
}

func transform(_ value: Int, using transformator: (Int) -> Int) -> Int {
    return transformator(value)
}

transform(6, using: anonymousClusure) // 18
```

Если вы смотрели предыдущую лекцию о функциях и методах, то наверняка вспомнили, что в ней мы объявляли точно такую же функцию $transform(\_ : using :)$, но передавали в нее функцию. В чем тогда разница между closure expression и функциями? Ответ в том, что разницы нет. Функции это и есть замыкания. Глобальные функции - это замыкания, которые ничего не захватывают. Вложенные функции - это замыкания, имеющие имя и захватывающие контекст в котором объявлены. Анонимные $closureexpression$ тоже захватывают контекст, но не имеют имени. В $Swiftclosureexpressions$ очень активно используются. Поэтому для них было придумано много оптимизаций в синтаксисе, чтобы сделать его более коротким. Для начала уберем ненужную переменную. Замыкание можно объявлять прямо при вызове функции. Ключевое слово in означает окончание объявления параметров и возвращаемого значения. Поэтому можно писать код замыкания в той же строке. Естественно не нужно это делать если в вашем замыкании больше одного выражения.

```
transform(6, using: {(anotherItemInArray:
Int) -> Int in return anotherItemInArray * multiplier})
```

Т.к. мы передаем замыкание в функцию Swift может сам вывести ее тип. Достаточно просто перечислить имена параметров.

```
transform(6, using: {anotherItemInArray in return anotherItemInArray * multiplier })
```

Если в вашем замыкании всего одно выражение, то ключевое слово return можно опустить. Результат его выполнения сам вернется из замыкания.

```
transform(6, using: {anotherItemInArray in  anotherItemInArray * multiplier })
```

Также Swift неявно объявляет параметры в виде $0, $1, $2 и т.д. если вы не объявите их сами. Они соответствуют параметрам замыкания по порядку.

```
transform(6, using: {$0 * multiplier })
```

Часто замыкание передается в функцию в качестве последнего или единственного аргумента. В этом случае его можно вынести за скобки. А если оно единственный аргумент даже не указывать их вообще.

```
transform(6) {$0 * multiplier }
```

Это особенно удобно использовать если в замыкании много кода и в одну строку его не записать.

```
transform(6) {
    let temp = $0 * multiplier

/*

Много полезных вычислений

*/

return temp
}
```

Несмотря на то, что Swift берет на себя управление памятью все же нужно понимать как работают замыкания, чтобы не допустить ошибки. Замыкание захватывает именно тот объект, который ему доступен при его создании. Например, если мы создадим еще одно замыкание, вызвав counterFunc(), то во время выполнения этой функции будет создана новая переменная counter. И новое замыкание захватит именно ее, а не ту, что была захвачена в прошлый раз.

```swift
let anotherCounter = counterFunc()
print("\(anotherCounter(3))") // running total: 3
print("\(closureFromFunc(3))") // running total: 12
```

При этом переменную оно захватывает по ссылке. Ее изменения внутри замыкания будут видны и снаружи. В качестве оптимизации может быть сделана копия, но только если Swift уверен, что внутри замыкания переменная не будет изменяться.

Однако у нас есть возможность изменить это поведение. Для этого используется список захвата. Он располагается перед списком параметров замыкания. Если его нет, то перед ключевым словом in. В квадратных скобках через запятую перечисляются имена переменных и констант. Все они будут скопированы в замыкание как константы.

В качестве бонуса есть возможность дать им другое имя. Но это только для удобства и ни на что не влияет.

```swift
var a = 0
var b = 0
let closure = {
    [newNameForA = a] in

    print(newNameForA, b)
}

a = 10
b = 10

closure() // 0 10
```

Помните, что замыкания сами по себе reference type. Так же как и обычные функции. Поэтому несмотря на то, что мы объявили anotherCounter как константу значение переменных, находящихся в ней, может меняться. Неизменно лишь то, на какую область памяти ссылается константа. И если мы присвоим значение anotherCounter другой константе или переменной, то они будут ссылаться на одно и тоже замыкание.

```swift
let secondReferenceToAnotherCounter = anotherCounter
print("\(secondReferenceToAnotherCounter(3))") // running total: 6
print("\(secondReferenceToAnotherCounter(3))") // running total: 9
print("\(anotherCounter(3))") // running total: 12
```

В предыдущем примере замыкания переданные в функцию transform выполнялись во время выполнения этой функции. Т.е. синхронно. К тому моменту когда управление возвращалось на-

зад замыкание уже было не нужно и Swift мог удалить его из памяти. Однако это не всегда так. Рассмотрим пример. На слайде вы видите класс у которого инициализатор принимает замыкание. Однако это замыкание не используется сразу, а сохраняется в свойство класса. Для этого его нужно пометить как @escaping. Замыкания, которые не помечены @escaping могут быть более агрессивно оптимизированы.

```
class ClosureRunner {
    private var closure: () -> Void

init(closure: @escaping () -> Void) {
    self.closure = closure
}

    func runClosure() {
        closure()
    }
}

let closureRunner = ClosureRunner() {
    print("Closure runned")
}

closureRunner.runClosure() // Closure runned
```

До версии Swift 3.0 логика была обратная. По умолчанию замыкания были escaping, а если это было не нужно, то необходимо было пометить его как @noescape для оптимизации. Но потом было решено, что безопасное и быстрое поведение по умолчанию лучше и логику изменили.

4.2. Autoclosure

autoclosure - это синтаксическая конструкция, облегчающая написание и чтение кода. Проще всего понять что это на примере. Допустим, необходимо реализовать функцию, принимающую два Bool и возвращающую результат логического умножения этих параметров. На вызывающей стороне будет такой код:

```
let someVar = false
let someVar2 = 26

and(someVar, someVar2 % 2 == 0) // false
```

```
5  and(someVar, someVar2 % 2 == 0) // false
```

При этом первый аргумент равен *false*, а второй сначала вычисляется во время выполнения программы, а потом передается в функцию. Т.к. значение логического умножения равняется false если хотя бы один из аргументов false, то нет никакого смысла вычислять значение второго параметра функции по тому, что результат будет все равно отрицательным из-за первого.

Для того, чтобы избавиться от ненужных вычислений можно передать второй аргумент в виде замыкания.

```
1  and(someVar,  {someVar2 % 2 == 0  }) // false
```

В этом случае значение второго аргумента будет вычисляться внутри функции, при вызове замыкания, только если первый аргумент равен true. Однако такой способ неудобен - нужно каждый раз добавлять фигурные скобки. Для решения этой проблемы и нужен autoclosure.

```
1  func and(_ lhs: Bool, _ rhs: @autoclosure () -> Bool) -> Bool {
2      guard lhs else {
3          return false
4      }
5
6      return rhs()
7  }
```

Рассмотрим реализацию функции and. Второй аргумент является замыканием с ключевым словом *@autoclosure*. Благодаря ему выражение, переданное в качестве второго аргумента не будет выполняться сразу, а обернется в замыкание. Внутри функции просто проверяется значение первого аргумента, и вызывается замыкание если оно true. Таким образом можно избавиться от лишних скобок и оптимизировать вычисление значения функции.

Еще один пример использование *autoclosure* - это стандартная функция assert

```
1  public func assert(_ condition: @autoclosure () -> Bool, _ message: @autoclosure
2      () -> String = default, file: StaticString = #file, line: UInt = #line)
```

Эта функция проверяет значение выражения и если оно ложно останавливает выполнение приложения. Ее особенность в том, что работает она только во время отладки. В релизной версии приложения эта функция игнорируется. Для этого condition имеет тип autoclosure и вычисляется только если приложение запущено для отладки. Параметр message тоже является autoclosure. Его значение вычисляется только если condition равен false.

Скорее всего вам не придется объявлять функцию с autoclosure, но вызов таких функций вполне обычная ситуация.

4.3. Определение операторов

Ранее мы познакомились с основными операторами, которые используются в Swift. Их достаточно для реализации любого функционала, который вам нужен. Но вы можете объявить свои операторы. Они позволят вам описать необходимый функционал в достаточно компактной форме. Этому посвящена наша лекция.

4.3.1. Типы операторов

Различают три основные формы объявления операторов: инфиксный, постфиксный и префиксный. Инфиксный — это бинарный оператор, который записывается между двумя операндами, например, оператор «+» при сложении двух чисел. Для инфиксного оператора можно определить порядок выполнения. Он помогает определить, какое действие будет выполнено раньше при группировке нескольких инфиксных операторов. По умолчанию назначается группа default precedence, порядок ее выполнения чуть выше, чем у тернарного оператора. Следующий префиксный оператор — это унарный оператор, записываемый перед операндом, например, оператор not в виде восклицательного знака. Постфиксный оператор записывается после операнда, например, восклицательный знак для выполнения операции force unwrap.

```
1  infix operator operator name: precedence group
2  prefix operator operator name
3  postfix operator operator name
```

4.3.2. Определение оператора

После объявления нового оператора вы реализовываете его с помощью объявления статического метода, который имеет то же имя, что и оператор. Метод относится к типу значения, которого он принимает в качестве аргумента. Например, метод для умножения int на double можно реализовать либо в структуре int, либо в структуре double. А теперь немного подробнее о реализации. Для определения нового действия с оператором вы можете либо воспользоваться перегрузкой операторов, либо создать новый оператор. Перегрузка позволяет использовать один и тот же оператор для выполнения операций с разными типами. Например знак «+» может использоваться как для сложения чисел, так и для сложения строк. Определим для нашей структуры собственную реализацию оператора сложения. Для этого создадим метод, принимающий два аргумента. Имя метода совпадает с оператором для перегрузки. Также возможна перегрузка

стандартных унарных операторов. При определении метода нужно лишь указать модификатор префикс или постфикс.

```
struct RGBColor {
    var red = 0.0
    var green = 0.0
    var blue = 0.0
}
extension RGBColor {
    static func + (left: RGBColor, right: RGBColor) ->
    RGBColor {
        return RGBColor(red: left.red + right.red, green:
        left.green + right.green, blue: left.blue +
        right.blue)
    }
}
let blue = RGBColor(red: 0.0, green: 0.0, blue: 1.0)
let red = RGBColor(red: 1.0, green: 0.0, blue: 0.0)
let purple = blue + red
print(purple.red, purple.green, purple.blue)
// 1.0 0.0 1.0
```

4.3.3. Операторы для поддержки протоколов

Для поддержки некоторых протоколов вам понадобится выполнить перегрузку операторов. Например, при реализации протокола Equatable и Comparable необходимо реализовать оператор сравнения. Реализация выполняется так же, как и для других инфиксных операторов: передается два аргумента, а результатом выполнения является bool.

```
struct RGBColor {
    var red = 0.0
    var green = 0.0
    var blue = 0.0
}

extension RGBColor: Equatable {
    static func == (left: RGBColor,
    right: RGBColor) -> Bool {
        return left.red == right.red &&
```

```
        left.green == right.green &&
        left.blue == right.blue
    }
}
```

```
let blue = RGBColor(red: 0.0, green: 0.0, blue: 1.0)
let secondBlue = RGBColor(red: 0.0, green: 0.0, blue: 1.0)
let red = RGBColor(red: 1.0, green: 0.0, blue: 0.0)

print(blue == secondBlue ? "equal" : "not equal")
// equal
print(blue == red ? "equal" : "not equal")
// not equal
```

4.3.4. Определение собственного оператора

В некоторых ситуациях вам может понадобиться определение собственного оператора, а не перегрузка уже имеющегося. Новый оператор объявляется с помощью ключевого слова operator, перед которым указывается его тип: префикс, инфикс или постфикс. Далее следует описание реализации оператора в статическом методе. В переименовании операторов есть некоторые ограничения. Ознакомиться с ними вы сможете в документации Apple по ссылке, приведенной после лекции.

```
infix operator **: MultiplicationPrecedence
extension Double {
    static func ** (left: Double, right: Double) ->
    Double {
    return pow(left, right)
    }
}
print(5 ** 5)
// 3125.0
```

4.3.5. Определение precedence group

Для определения порядка выполнения сгруппированных операторов используются группы

приоритета. Они указываются при объявлении оператора. Приоритет определяет очередность выполнения выражений при отсутствии скобок для группировки. Precedence group указывается только при объявлении инфикс-операторов, для префикс и постфикс этого делать не нужно. Важно заметить, что если к одному операнду применяется и постфиксный, и префиксный операторы, то постфиксный оператор будет выполняться первым. Подобные группы могут быть определены самостоятельно. Группа приоритетов объявляется следующим образом. В параметрах lower и higher передается группа, по отношению к которым будет задаваться приоритет. Могут указываться только группы, не входящие в текущий модуль. По умолчанию в Swift реализовано несколько групп приоритетов, например, для операторов «+» и «−» — это группа addition precedence, а для операторов «*» и «/» — группа multiplication precedence. Ссылка на полный список стандартных групп приведена после лекции.

```
precedencegroup precedence group name {
    higherThan: lower group names
    lowerThan: higher group names
    associativity: associativity
    assignment: assignment
}

precedencegroup PowerPrecedence {
    higherThan : AdditionPrecedence
    lowerThan : MultiplicationPrecedence
    associativity : right
}
```

4.3.6. Associativity

Параметр associativity отвечает за порядок выполнения выражений с одинаковым приоритетом. Вы задаете параметр с помощью одного из ключевых слов: left, right или none. Рассмотрим на примере: выражения группы имеют одинаковый приоритет. Для оператора «−» задан параметр associativity, равный left. То есть сначала будет выполнено выражение слева, а затем к результату этого выражения будет прибавлено значение справа. Если бы параметр associativity был равен right, то сначала выполнялось бы выражение справа, а затем из значения слева вычитался бы результат этого выражения. Если параметры устанавливаются в значении none, то подобные выражения не могут примыкать друг к другу. Примером может служить оператор сравнения, продемонстрированный на слайде.

```
associativity : left
7 - 2 + 3 = 8 //Вычисляется как ((7 - 2) + 3) = 8
associativity : right
7 - 2 + 3 = 2 //Вычисляется как (7 - (2 + 3)) = 2
associativity : none
5 < 3 < 2 // Ошибка
```

4.3.7. Assignment

Последним параметром является параметр assignment, который может принять значения true или false. На слайде показана разница при выполнении оператора в зависимости от значения этого параметра. Дополнительную информацию об этом вы можете получить на Github в разделе swift-evolution, ссылку на которую вы найдете в материалах для этой лекции.

```
assignment : true
foo?.bar += 2 //эквивалентно foo?(.bar += 2)
assignment : false
foo?.bar += 2 //эквивалентно (foo?.bar) += 2
```

4.4. Свойства

4.4.1. Виды свойств

Свойства (англ. properties) связывают значения с конкретным классом, структурой или перечислением. Различают хранимые и вычисляемые свойства. Хранимые свойства запоминают значение константы или переменной, в то время как вычисляемые свойства рассчитывают значение при обращении. Вычисляемые свойства доступны классам структурам и перечислениям, а хранимые только классам и структурам. Хранимые и вычисляемые свойства обычно привязываются к инстансам,но свойства могут быть связаны непосредственно с типом. Такие свойства называются свойствами типов. Также для хранимых свойств можно определить обозреватель(observers). Они будут наблюдать за изменением значений свойства, а при изменении, выполнять дополнительные действия.

4.4.2. Хранимые свойства

В простейшем понимании хранимое свойство - это константа или переменная принадлежащая инстансу. Для свойства можно указать значение по умолчанию или присвоить значение в инициализаторе.

```
struct SimpleStruct {
    var firstValue: Int = 1
    let secondValue: Int
}
var simpleStruct = SimpleStruct(firstValue: 0, secondValue: 2)
simpleStruct.firstValue = 6
```

Хранимые свойства в структурах Если вы объявите структуру и присвоите значение константе. то значение свойств у этой структуры поменять вы не сможете.

```
let constantStruct = SimpleStruct(firstValue: 0, secondValue: 2)
constantStruct.firstValue = 3 // Error: change 'let' to 'var' to make it mutable
```

Это связано с тем, что структуры являются значимыми типами. Если инстанс структуры помечается, как константа, то и свойства структуры будут константами. Для классов поведение отличается, так как классы - ссылочный тип. При присвоении инстанса класса константе, значения свойств, объявленных, как переменные могут быть изменены.

Хранимые свойства с отложенной инициализацией

Отложенная инициализация позволяет установить значение для свойства в момент первого обращения к нему. Такие свойства помечаются ключевым словом lazy.

```
class DataStore {
    var counter = 0
}

class DataManager {
    lazy var dataStore = DataStore()
}

let dataManager = DataManager()
```

```
dataManager.dataStore.counter = 1
dataManager.dataStore.counter = 2
```

Свойства с отложенной инициализацией всегда объявляются как переменные. Ведь значение подобного свойства может быть определено после инициализации объекта, а для констант это непозволительно. Использовать подобные свойства удобно, если их значения зависят от других значений и не могут быть определены на этапах создания объекта. Также отложенную инициализацию стоит применять для свойств, чьё создание может занять продолжительное время и потребовать много ресурсов.

4.4.3. Вычисляемые свойства

Помимо хранимых свойств могут быть определены вычисляемые свойства, они не хранят определенное значение, а вычисляют его при обращении к переменной.

```
class AnotherDataStore {
    var counter: Int {
        get {
            return numbers.count
        }
        set(newCounter) {
            numbers = []
            for index in 0...newCounter {
                numbers.append(index)
            }
        }
    }

    var numbers = [Int]()
}

var anotherDataStore = AnotherDataStore()
anotherDataStore.counter = 5
print(anotherDataStore.counter)
```

Если у вычисляемого свойства в сеттере не задается имя для нового значения, оно задается по-умолчанию, с названием newValue. Имена передаваемых переменных также можно назначить самостоятельно. Когда для вычисляемого свойства устанавливается геттер, но не устанавлива-

ется сеттер, оно доступно только для чтения. Значения вычисляемого свойства устанавливается как переменная, а не как константа, так как значение не фиксировано. При создании вычисляемого свойства только с геттером, синтаксис можно немного упростить, убрав ключевое слово get.

4.4.4. Наблюдатели свойств

Свойства следят за своими значениями и могут сообщать об этом. Обозреватели свойств вызываются при установке значения, даже если значение не изменилось. Обозреватели можно использовать для любых хранимых свойств, кроме свойств с отложенной инициализацией. Также обозреватели могут быть добавлены для унаследованных свойств. Для наблюдения за изменения можно использовать willSetили didSet.

```
class ExplicitCounter {
    var total: Int = 0 {
        willSet(newTotal) {
            print("Скоро установится новое значение \(newTotal)")
        }
        didSet {
            if total > oldValue {
                print("Значение изменилось на:\(total - oldValue)")
            }
        }
    }
}

let explicitCounter = ExplicitCounter()
explicitCounter.total = 5
explicitCounter.total = 9
```

WillSet вызывается непосредственно перед изменение свойства DidSet вызывается сразу после изменения свойства. В willSet передается константное значение. Для значение может быть использовано уникальное имя, если оно не установлено, значение передается с именем newValue. Для didSet также передается старое значение, но оно имеет другое имя по-умолчанию, oldValue. Если внутри didSet установить новое значение наблюдаемой переменной, оно заменит только что установленное. Когда свойство с обозревателями передается функции в качестве in-out параметра, у свойства будут вызваны методы willSet и didSet. Это связано с особенностью реализации in-out параметров. При завершении функции всегда записывается новое значение и соответствен-

но вызываются обозреватели.

4.4.5. Глобальные и локальные переменные

Ранее описанные возможности доступны для глобальных и локальных переменных. Глобальные переменные задаются вне метода, функции или контекста класса. Локальные переменные объявляются внутри функции или замыкания. Для локальных и глобальных переменных можно задать обозреватели или сделать переменные вычисляемыми. Все точно также, как и для свойств. Значения вычисляются при обращении к переменной. Глобальные переменные и константы всегда вычисляются при обращении, в отличии от свойств глобальные переменные не нужно помечать ключевым словом lazy. Локальные переменные не могут обладать отложенной инициализацией.

4.4.6. Свойства для типов

Свойства объектов принадлежат объектам и создаются каждый раз при инициализации нового объекта. Но можно создать свойство, которое будет единственным для всех объектов заданного типа и значение которого будет инициализировано только 1 раз. Такие свойства, называются свойствами типов. Они аналогичны переменным и константам из языка Си, объявленным со словом static. Вычисляемые свойства типов могут быть объявлены только, как переменные. Хранимые свойства типов могут быть объявлены, как переменные, так и как константы. В отличии от свойств у инстансов, для переменных обязательно указание значения по-умолчанию потому, что у типов нет инициализатора. Соответственно нет возможности для задания первоначального значения. Хранимые свойства получают значения при первом обращении к ним. Это происходит только 1 раз

4.4.7. Синтаксис

В Swift свойства типов описываются внутри описания типа. Каждое свойство привязно к типу в котором оно описано. Каждое свойство привязывается к типу, для которого оно описывается.

```
1  struct SomeStruct {
2          static var storedTypeProperty = 1
3          static var squareOfProperty: Int {
4              return storedTypeProperty * storedTypeProperty
5      }
6  }
7  class SomeClass {
```

```
8       static var storedTypeProperty = 2
9       static var squareOfProperty: Int {
10          return storedTypeProperty * storedTypeProperty
11      }
12      static var overridableHalfOfProperty: Int {
13          get {
14              return storedTypeProperty * storedTypeProperty
15          }
16          set {
17              storedTypeProperty = newValue / 2
18          }
19      }
20  }
21
22  print (SomeStruct.storedTypeProperty)
23
24  SomeClass.overridableHalfOfProperty = 10
25  print(SomeClass.squareOfProperty)
```

Обозначаются они ключевым словом static, либо в случае с классами можно использовать ключевое слово class, что позволит наследникам переопределять свойство. Обращение осуществляется при помощи dot-синтаксиса. Однако, обращение идет не к инстансу, а к типу.

4.5. Протоколы

Протокол - это перечисление свойств, методов и других требований. Они могут быть выполнены в классе, структуре или перечислениях. Говорят, что тип поддерживает(conform) протокол если он предоставляет реализацию всех требований. Обратите внимание на то, что в самом протоколе ничего не объявлено. Это просто список требований. Все свойства и методы должны быть реализованы непосредственно в типе, который поддерживает протокол. Чтобы объявить новый протокол нужно использовать ключевое слово protocol. После него указывается имя, с большой буквы, и в фигурных скобках требования. Объявление очень похоже на структуры и классы.

```
1  protocol MyProtocol {
2
3  }
```

Давайте создадим структуру, поддерживающую этот протокол. Для этого нужно указать его при объявлении структуры после двоеточия. Таким образом можно перечислить через запятую

несколько протоколов.

```
struct MyStruct: MyProtocol {

}
```

Объявим более полезный протокол. Добавим в него несколько требований. Для этого нужно их указать так же как при объявлении в структуре или классе. После типа в фигурных скобка указывается может ли это свойство быть только для чтения или нужно дать доступ и на установку значения.

```
protocol MyProtocolWithProperties {
    var settableProperty: Int {set get }
    var gettableProperty: Double {get }
```

Добавим новую структуру. В ней объявим пару свойств для поддержки протокола.

```
struct MyStructWithProperties: MyProtocolWithProperties {
    var settableProperty: Int {
        set {
        }

        get {
            return 5
        }
    }

    var gettableProperty: Double
}
```

Протокол не может требовать вычисляемое или обычное свойство. Поэтому мы в структуре можем использовать и то и другое. Обратите также внимание на то, что можно определить сеттер для свойства даже если протоколе указано read only.

В протокол можно добавить требования не только для экземпляров, но и для самих типов. Для этого нужно добавить ключевое слово static перед свойством.

```
protocol MyProtocolWithTypeProperty {
    static var typeProperty: Int {get }
}
```

Давайте объявим протокол с требованиями к методам.

```
protocol MyProtocolWithMethods {
    mutating func instanceMethod(parameter: Int)
    static func typeMethod() -> Double
}
```

Они похожи на объявление методов в структурах и классах, но без фигурных скобок и реализации. Также можно использовать слово static для методов типов, но значения по умолчанию указывать нельзя.

Добавим структуру с поддержкой нескольких протоколов.

```
struct MyStructWithPropertiesAndMethods: MyProtocolWithProperties,
MyProtocolWithMethods {
    var settableProperty: Int
    var gettableProperty: Double

    mutating func instanceMethod(parameter: Int) {
        gettableProperty = Double(parameter)
    }

    static func typeMethod() -> Double {
        return 4.0
    }
}
```

Реализация методов может быть любой. Протокол требует только наличия самого метода, но не логики в нем. Т.к. мы добавили mutating при объявлении протокола мы можем изменять значения свойств в этом методе.

Помимо обычных методов протокол может требовать наличие инициализатора.

```
protocol MyProtocolWithInit {
    init(paramenter: Int)
}
```

Объявляется он также - без фигурных скобок и реализации.

Однако есть одна особенность. В классе поддерживающем этот протокол нужно обязательно объявлить такой инициализатор как required.

```
class MyClassWithInit: MyProtocolWithInit {
    let someConst: Int

```

```
4       required init(paramenter: Int) {
5           someConst = paramenter
6       }
7   }
```

Это нужно для того, чтобы все наследники тоже предоставили собственную реализацию. Подробнее о наследовании и инициализаторах смотрите в соответствующих лекциях.

Несмотря на то, что протоколы не включают в себя никакую имплементацию, а только требования мы все равно можем использовать их как любой другой тип в Swift: можно объявить переменную или константу этого типа, можно принимать и возвращать их из функций и т.д.

Объявим переменную типа MyProtocolWithProperties.

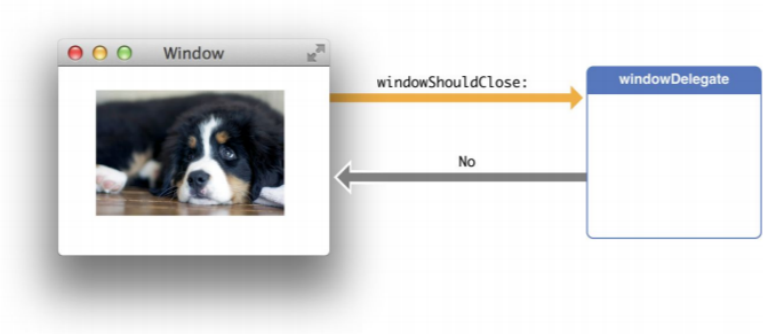
```
1   var someObject: MyProtocolWithProperties = MyStructWithProperties(gettableProperty:
     ↪  5.0)
2   someObject.settableProperty
3
4   someObject = MyStructWithPropertiesAndMethods(settableProperty: 1,gettableProperty:
     ↪  2.0)
5   someObject.settableProperty
```

Присвоим ей экземпляр структуры MyStructWithProperties. При этом информация о том, экземпляр какого класса или структуры лежит в этой переменной теряется.

Вызывающая сторона может обратиться только к тем свойствам и методам, которые есть в требованиях протокола.

Мы можем присвоить этой переменной экземпляр другой структуры. При этом для вызывающей стороны ничего не изменится. Известно лишь то, что у этого объекта будут свойства gettableProperty и settableProperty.

Это один из вариантов использования протоколов. Он часто встречается в разработке под iOS в виде архитектурного паттерна - делегат. Реализуем пример делегата.



Для начала опишем класс, который будет обрабатывать массив строк.

```
class StringProcessor {
    var delegate: StringProcessorDelegate?

    func process(_ strings: [String]) -> String {
        guard strings.count > 0, let delegate = delegate else {
            return ""
        }

        var temp = [String]()
        for string in strings {
            let t = delegate.transform(string)
            temp.append(t)
        }

        return temp.joined(separator: delegate.concatenateSeparator)
    }
}
```

У него будет свойство хранящее делегат. В методе process он каждую строку трансформирует с помощью делегата и складывает результат в одну строку используя разделитель.

В требованиях в протоколе укажем метод для трансформации и разделитель в виде свойства.

```
protocol StringProcessorDelegate {
    var concatenateSeparator: String {get }

    func transform(_ string: String) -> String
```

Объявим также несколько классов поддерживающих этот протокол.

```
class UpperCaseStringProcessorDelegate: StringProcessorDelegate {
    var concatenateSeparator: String {
        return " "
    }

    func transform(_ string: String) -> String {
        return string.uppercased()
    }
}

```

```
11
12  class FirstLetterStringProcessorDelegate: StringProcessorDelegate {
13      var concatenateSeparator: String {
14          return ""
15      }
16
17      func transform(_ string: String) -> String {
18          guard let firstCharacter = string.first else {
19              return ""
20          }
21          return String(firstCharacter)
22      }
23  }
```

Первый будет трансформировать все строки в верхнем регистре. Разделителем будет пробел.

Второй возвращает из трансформации только первый символ в строке. Разделитель у него пустая строка.

Попробуем все в действии. Создадим процессор, его делегаты и тестовый массив.

```
1  var processor = StringProcessor()
2  let upperCaseDelegate = UpperCaseStringProcessorDelegate()
3  let firstLetterDelegate = FirstLetterStringProcessorDelegate()
4  let testArray = ["This", "is", "simple", "test", "strings"]
```

Сначала просто вызовем обработку без делегатов. Получим ожидаемую пустую строку.

```
1  processor.process(testArray) // ""
```

Выставим первый делегат в процессоре. В результате получим строку состоящую из заглавных букв.

```
1  processor.delegate = upperCaseDelegate
2  processor.process(testArray) // "THIS IS SIMPLE TEST STRINGS"
```

Попробуем тоже самое со вторым. Получим строку состоящую из первых букв тестовых слов.

```
1  processor.delegate = firstLetterDelegate
2  processor.process(testArray) // "Tists"
```

Как видите мы вынесли логику из процессора в делегат. Это дает нам возможность легко изменять поведение, просто поменяв один делегат на другой. Со стороны процессора при этом

ничего не меняется. Он знает только, что его делегат будет предоставлять ему методы, перечисленные в протоколе, с ними он и взаимодействует. Помимо простой переменной мы можем объявить и целую коллекцию объектов, поддерживающих какой-то протокол. Создадим массив делегатов процессора.

```
var arrayOfDelegates = [StringProcessorDelegate]()
```

Теперь мы можем положить в него любой объект поддерживающий этот протокол.

```
arrayOfDelegates.append(upperCaseDelegate)
arrayOfDelegates.append(firstLetterDelegate)

for delegate in arrayOfDelegates {
    delegate.concatenateSeparator
}
```

Тип объектов будет соответствовать протоколу. Вы не будете знать, что за объект вы получили из массива. Конечно, в Swift есть приведение типов, но если у вас при написании кода возникает такая необходимость значит вы что-то делаете не так.

Apple называет Swift протоколо-ориентированным языком программирования. Протоколы это не просто абстрактный интерфейс класса или структуры. Добавляя поддержку протоколов в какой-либо тип мы добавляем ему определенную функциональность. Например, Equatable для проверки на равенство, ExpressibleByIntegerLiteral для инициализации типа с помощью целочисленного литерала, Hashable для получения хэш-суммы объекта и т.д. Объявляя какой-нибудь метод мы просто перечисляем поддержку какой функциональности мы ждем от получаемого объекта. При этом нам уже не важен его тип. В лекции по расширению функциональности мы покажем как добавить поддержку протокола уже существующему типу. А пока посмотрим как использовать композицию протоколов.

Композиция протоколов - это объединение нескольких протоколов в один тип. Их еще называют экзистенциальными типами. Объявляются они перечислением протоколов через амперсанд.

```
let compoundObject: MyProtocolWithProperties & MyProtocolWithMethods
compoundObject = MyStructWithPropertiesAndMethods(settableProperty: 1,
 ↪  gettableProperty: 2.0)
```

Переменная compoundObject может содержать только объект поддерживающий оба протокола.

В $Swift4$ добавили возможность объявить экзистенциальный тип, содержащий не только протоколы, но и класс. Это может очень пригодиться при работе с системными классами. В следующем примере мы объявляем переменную с таким типом.

```
let compoundClassObject: UIView & MyProtocol
```

Она может содержать только экземпляры класса $UIView$ либо ее сабклассов. При этом они обязательно должны поддерживать протокол $MyProtocol$. Вы еще встретите подобные конструкции в следующих лекциях и будет понятнее как это использовать.

Мы можем проверить поддерживает ли объект какой-то протокол или привести, например, $AnyObject$ к нужному протоколу. Делается это так же как с любым другим типом - с помощью операторов is, $as!$, $as?$.

```
let arrayOfAny: [Any] = [4, firstLetterDelegate, "String"]

for object in arrayOfAny {
    if let processorDelegate = object as? StringProcessorDelegate {
        processor.delegate = processorDelegate
        processor.process(testArray) // "Tists"
    }
}
```

Рассмотрим простой пример. У нас есть массив объектов, который по какой-то причине имеет тип $[Any]$. Но мы знаем, что в нем должны быть объекты, которые поддерживающие какой-то протокол.

Воспользуемся as? и if let для безопасного преобразования типа. Даже если в этот массив попало что-то другое, как в нашем примере, то ничего страшного не произойдет. Не стоит использовать as! даже если уверены, что ничего лишнего вам прийти не могло.

В Swfit все требования протокола обязательны к исполнению. Если вы хотите иметь возможность использовать опциональные протоколы, как в Objective-C, то вам придется воспользоваться атрибутом $@objc$. Он был добавлен в Swift для обеспечения взаимодействия с Objective-C.

Объявим протокол, добавив к нему $@objc$. Теперь он будет доступен для $Objective - C$ кода. Все опциональные требования тоже должны быть помечены этим атрибутом.

```
@objc protocol ObjCProtocol {
    @objc optional func optionalFunc() -> Int
    func normalSwiftRequiredFunc()
}
```

Методы и свойства, объявленные таким образом, будут иметь опциональный тип. Т.е. в нашем примере это будет (() -> Int)? . Обратите внимание, что опциональным стал не возвращаемый тип, а сам метод.

У опциональных протоколов есть ограничения. Они могут поддерживаться только классами

из Objective-C, либо Свифтовыми классами помеченными атрибутом @objc. При этом структуры и перечисления не могут поддержать такой протокол. Объявим такой класс.

```
@objc class MyObjcExposedClass: NSObject, ObjCProtocol {
    func normalSwiftRequiredFunc() {
    }
}

let objcClass: ObjCProtocol = MyObjcExposedClass()
let res = objcClass.optionalFunc?()
res // nil
```

Наследование мы рассмотрим в будущих лекциях, но сейчас мы обязаны воспользоваться им. Синтаксис наследования в Swift похож на другие языки программирования и скорее всего покажется вам знакомым. Тут мы объявляем класс наследник NSObject и добавляем ему поддержку нашего опционального протокола.

NSObject это основной базовый класс в Objective-C. Он него наследуется большинство других классов. Скорее всего вы будете работать либо с ним, либо с одним из его сабклассов.

Как видите мы не добавили опциональный метод в наш класс. При вызове нужно учитывать такую ситуацию. Для этого укажем после имени метода ?. В этом случае ничего вызываться не будет и в результате будет nil. Обратите внимание, что тип переменной указан явно и является ObjCProtocol. Если бы переменная была MyObjcExposedClass, то Swift мог бы определить, что нужный метод отсутствует и не дал бы выполнить код.

4.5.1. Коллекции. Основы

Коллекции. Основы “Коронное приветствие лектора” Этим видео мы начинаем цикл лекций о коллекциях. Сейчас мы рассмотрим основы и особенности коллекций в Swift, а в следующих видео перейдем к протоколам на которых они построены.

В Swift standard library есть три всем известные структуры данных: массивы, словари и множества. Они содержат соответственно упорядоченный набор элементов, неупорядоченный набор пар ключ-значение и неупорядоченный набор уникальных элементов.

В отличии от Objective-C в Swift нет разделения на изменяемые и неизменяемые коллекции. Это будет зависеть

от того объявили ли вы коллекцию как константу или как переменную.

```
var mutableArray: [Int] // Изменяемый массив целых чисел
let immutableDictionary: [String: String] // Неизменяемый словарь строк
```

```
var mutableSet: Set<Int> // Изменяемое множество целых чисел
```

Не забывайте, что неизменяемость относится только к содержимому самой коллекции. Т.е. если коллекция содержит объекты, то гарантируется только то, что указатель
на них не изменится. При этом данные внутри самих объектов могут изменяться.
Тоже самое происходит при копировании коллекций. В Swift все коллекции являются value type. Поэтому при передаче коллекции в функцию можно не беспокоиться, что функция изменит вашу коллекцию даже если она объявлена как var.

```
var arrayOfInt = [1, 2, 3]
var copyOfArrayOfInt = arrayOfInt
copyOfArrayOfInt[1] = 5
copyOfArrayOfInt // [1, 5, 3]

arrayOfInt // [1, 2, 3]
```

Для инициализации коллекций можно использовать литералы. При этом, как и для простых переменных, тип коллекции указывать не обязательно.

```
let fibs = [0, 1, 1, 2, 3, 5] // [Int]
let JSON = ["language":"Swift", "version":"4.0"]
```

Множеству можно присваивать литерал массива. При этом все повторяющиеся элементы будут отброшены.

```
let uniqueNumbers: Set = [1, 2, 3, 3, 4] // Полностью тип указывать не обязательно.
                                 //Swift поймет, что uniqueNumbers должен быть
                                  ↪  Set<Int>
uniqueNumbers // [2, 3, 1, 4] Порядок элементов во множестве не гарантируется
```

Элементы множества и ключи в коллекции должны поддерживать протокол Hashable. Это необходимо т.к.
имплементация множеств и коллекций построена на хэш таблицах.
Для получения элементов можно использовать subscipt.

```
fibs[2] // 1
JSON["version"] // "4.0"
```

Помимо обычного subscript в коллекциях есть возможность получить сразу диапазон элементов по индексам.

```
let slice = fibs[2..<fibs.endIndex]
slice // [1, 2, 3, 5]
type(of: slice) // ArraySlice<Int>
```

В общем случае такой вызов возвращает Slice. Для массива определена коллекция ArraySlice. С ним можно осуществлять те же операции, что и с обычным массивом. При создании Slice не происходит копирования элементов. Вместо этого он хранит указатель на оригинальный массив, начало и конец выбранного диапазона.

Разумеется мы можем использовать в Swift все коллекции из Foundation. Однако, следует всегда помнить, что их

поведение отличается. Например, при использовании NSArray нужно помнить о том, что он не value type и может измениться несмотря на то, что мы думаем, что это NSArray и он неизменяемый. Для решения этой проблемы нужно делать явную копию NSArray.

```
let nsmutableArray = NSMutableArray(array: [1, 2, 3])
let immutableReference: NSArray = nsmutableArray
let immutableCopy = nsmutableArray.copy() as! NSArray

nsmutableArray[0] = 6
immutableReference // [6, 2, 3]

immutableCopy // [1, 2, 3]
```

Некоторые коллекции перенесены в Swift как value type и даже поддерживают протоколы коллекций. IndexSet используется для хранения индексов. Его преимущества перед простым Set в том, что он хранит индексы как набор диапазонов. Но вы при этом можете работать с ним как с обычным множеством.

```
var indices = IndexSet()
indices.insert(integersIn: 1..<5)
indices.insert(integersIn: 11..<15)

let evenIndices = indices.filter {$0 % 2 == 0 } // [2, 4, 12, 14]
evenIndices.contains(4) // true
```

Еще один пример это CharacterSet. Он используется для хранения набора Unicode совместимых символов. Он поддерживает операции над множествами однако коллекцией не является.

В обратную сторону перенос тоже работает. До Swift 3.0 для использования Swift коллекций в Objective-C все его объекты должны были быть наследниками AnyObject. Сейчас любая коллекция может быть использована в Objective-C. При этом

примитивные типы будут автоматически обернуты в box class т.к. Obj-c коллекции могут содержать только объекты.

Прежде чем мы перейдем к разбору протоколов на которых построены коллекции я хочу рассказать про особенность Swift связанную с доступом по индексу. В Swift математика с индексами и обращение по индексу к массиву не рекомендуется. Обращение к несуществующему элементу по индексу в массиве приводит к крэшу, а безопасный аналог этой операции намеренно не был добавлен. Так же в Swift 3.0 были убраны операции инкремента и декремента. Все это подталкивает разработчиков к отказу от C-style for цикла с коллекциями и от использования subscript.

Как вы увидите в следующих уроках в протоколах коллекций есть много полезных функций и свойств чтобы писать Swifty код, и не использовать индексы. Например, для того чтобы пройтись по всем элементам в массиве нужно использовать цикл for in, для получения первого или последнего элемента - first и last, для трансформирования - map, filter, reduce.

В этой лекции мы научились основам работы с коллекциями в Swift, вспомнили о тех, что пришли из Objective-C и вкратце затронули тему взаимодействия этих языков. “Коронное прощание лектора”

4.6. Коллекции. Sequence

“Коронное приветствие лектора” В этой лекции я начну рассказ о протоколах на которых построены коллекции.

Протокол Sequence является основой для всех коллекций. Несмотря на большой список методов Sequence предъявляет всего одно требование - метод makeIterator(). Для остальных методов уже написана дефолтная реализация. Поэтому мы начнем со знакомства с итераторами.

```
1  public protocol IteratorProtocol {
2      associatedtype Element
3      public mutating func next() -> Self.Element?
4  }
```

В протоколе IteratorProtocol объявлен всего один метод next(). Он возвращает следующий элемент или nil если его нет. Обратите внимание, что тип элементов последовательности объявлен в итераторе. Поэтому в самой Sequence элементы объявляются через него.

```
1  public func drop(while predicate: (Self.Iterator.Element) throws -> Bool)
2                                                              rethrows ->
                                                                ↪  Self.SubSequence
```

Итераторы возвращают только следующий элемент и никогда не сбрасываются в начало. Они могут предоставлять элементы бесконечно или возвращать всегда nil.

Рассмотрим пример простейшего итератора. Он не связан ни с какой последовательностью. Он генерирует числа кратные заданному значению. При этом nil он никогда не вернет. Если вызывать next() в цикле while let

element = iterator.next(), то произойдет переполнение переменной base и приложение упадет.

```swift
struct MultiplicityIterator: IteratorProtocol {
    var base = 1

    public mutating func next() -> Int? {
        defer {base += base }
        return base
    }
}

var i = MultiplicityIterator()
i.base = 4

i.next() // Optional(4)
i.next() // Optional(8)
i.next() // Optional(16)
```

Большинство итераторов имеют value semantics, но это не всегда так. AnyIterator - это итератор который оборачивает другой итератор. Его можно использовать для того, чтобы скрыть детали реализации.

```swift
let sequence = stride(from: 0, to: 10, by: 1) // Последовательность чисел от 0 до 10
                                              // с шагом 1

var iterator1 = sequence.makeIterator()
var anyIterator1 = AnyIterator(iterator1)

var iterator2 = iterator1
var anyIterator2 = anyIterator1

iterator1.next() // Optional(0)
iterator1.next() // Optional(1)

iterator2.next() // Optional(0)
```

```
14  iterator2.next() // Optional(1)
15
16  anyIterator1.next() // Optional(0)
17  anyIterator2.next() // Optional(1)
18  anyIterator1.next() // Optional(2)
19  anyIterator1.next() // Optional(3)
```

Поэтому не следует передавать итераторы из одной части программы в другую. Они должны использоваться локально и сразу отбрасываться после использования.

У AnyIterator есть еще одно применение. У него есть второй конструктор. Он принимает замыкание, возвращающее следующий элемент последовательности. Вместе с AnySequence это позволяет создать новую последовательность без создания нового типа.

```
1  let randomIterator = AnyIterator() {
2  arc4random_uniform(100) }
3  let randomSequence = AnySequence(randomIterator)
4
5  let arrayOfRandomNumbers =
6  Array(randomSequence.prefix(10)) // Массив случайных чисел
```

Вернемся к самим последовательностям. Скорее всего вам не придется взаимодействовать с итераторами самостоятельно. Обычно для прохода по последовательности используется for in цикл который сам создает новый итератор и вызывает у него next() пока не получит nil.

```
1  for randomNumber in randomSequence.prefix(15) {
2      print("Another random number: \(randomNumber)")
3  }
```

Рассмотрим Sequence аналогичный, показанному ранее MultiplicityIterator. В отличие от итератора он будет генерировать числа в виде последовательности которую мы можем, например, преобразовать в массив чисел. Каждый раз вызывая prefix() мы получаем ту же самую последовательность чисел т.к. создается новый итератор.

```
1  struct MultiplicitySequence: Sequence {
2      private let base: Int
3
4      init(base: Int) {
5          self.base = base
6      }
7
```

```
public func makeIterator() -> MultiplicityIterator {
        var iterator = MultiplicityIterator()
        iterator.base = base
        return iterator
    }

}

let multiplicitySequence = MultiplicitySequence(base: 4)
Array(multiplicitySequence.prefix(4)) // [4, 8, 16, 32]
Array(multiplicitySequence.prefix(4)) // [4, 8, 16, 32]
```

Важно понимать, что последовательность не всегда можно представить как массив с данными. Например, у нас есть поток данных из интернета. Его можно обернуть в Sequence. Но получив однажды какой-то элемент мы не можем заставить последовательность выдать нам его снова. Даже если создадим новый экземпляр итератора. В отличии от коллекций

последовательности при каждом проходе могут генерировать любые значения. Такие последовательности называются destructively consumed sequence.

Возможно вы задумались зачем вообще в последовательности нужен итератор и можно ли обойтись без него. Действительно, в некоторых случаях это возможно. Swift вам в этом поможет. Все что нужно сделать это определить в Sequence метод next().

```
struct Countdown: Sequence, IteratorProtocol {

    var count: Int

    mutating func next() -> Int? {
        if count == 0 {
           return nil
        } else {
            defer {count -= 1 }
            return count
        }
    }
}
```

Однако такой подход можно использовать только с разрушающимися последовательностями. Ведь каждый такой экземпляр Sequence можно будет пройти только один раз - метода создающего новый итератор нет. Последовательность сама себе итератор.

Еще одной важной частью Sequence является Subsequence. Он используется как результат

выполнения методов у Sequence, возвращающих часть последовательности, например: prefix и suffix - возвращают первые и последние n элементов; dropFirst и dropLast - возвращают последовательность без

первых и последних n элементов; split - разделяет последовательность по разделителю и возвращает массив subsequence.

Если вы не предоставите собственную реализацию, то по умолчанию в качестве Subsequence будет использоваться $AnySequence < Iterator.Element >$. В некоторых случаях будет логично в качестве Subsequence использовать тот же тип, что и сама коллекция т.е. Subsequence == Self, но в данный момент Swift не поддерживает рекурсивное описание associated type. Эта возможность будет реализована в будущем и код ниже будет компилироваться.

```
// Не компилируется в Xcode 9 Beta 5
protocol Sequence {
    associatedtype SubSequence: Sequence
    where Iterator.Element == SubSequence.Iterator.Element,
                                        SubSequence.SubSequence == SubSequence
    func dropFirst(_ n: Int) -> Self.SubSequence
    // ...
}
```

По понятным причинам получить размер Sequence невозможно. Но в протоколе есть свойство underestimatedCount. Оно позволяет получить примерное количество элементов в последовательности. Оно гарантированно будет не больше реального числа объектов, но может быть меньше. Его безопасно использовать с разрушающимися коллекциями.

Еще одно требование Sequence - это соблюдение алгоритмической сложности. Большинство методов должны иметь сложность O(n). Т.е. время их

выполнения прямо пропорционально размеру последовательности. Но некоторые должны отрабатывать за константное время. Например, $makeIterator()$, $dropFirst()$, $dropLast()$ и свойство $underestimatedCount$. Обратитесь к документации для уточнений если будете реализовывать собственные последовательности.

В данной лекции мы рассмотрели устройство итераторов и последовательностей, методы и их требования к реализации. В следующей лекции мы рассмотрим протокол *Collections*. “Коронное прощание лектора”

4.7. Коллекции. Collection

Коллекции. Collection “Коронное приветствие лектора” В этой лекции мы рассмотрим протокол Collection, индексы и то как создать свою коллекцию.

Коллекция в Swift - это стабильная последовательность элементов, которую можно безопасно итерировать много раз. И в отличии от Sequence они не могут быть бесконечными.

Протокол Collection наследуется от Sequence, добавляя индексы, сабскрипты и обязательство не

разрушать данные при проходе итератора. Благодаря последнему можно использовать этот протокол чтобы показать, что ваша последовательность стабильная даже если вам не нужна остальная функциональность коллекций.

При создании собственной коллекции обязательными для имплементации являются startIndex, endIndex, методы subscript(position:) и index(after:). Для остальных уже есть реализация по умолчанию. К сожалению нет никакого механизма помогающего это понять. Приходится играть в

угадайку с компилятором либо искать ответы в документации к протоколу.

Рассмотрим простейшую коллекцию - стэк. В данной реализации он может хранить в себе строки и выдавать их по принципу последний вошел - первый вышел. Для этого у него есть методы push() и pop(). Хранит он данные в массиве, но это может быть любой другой контейнер, сохраняющий порядок элементов.

```
1  struct Stack {
2      private var privateStorage: Array<String> = []
3
4      mutating func push(_ element: String) {
5          privateStorage.append(element)
6      }
7
8      mutating func pop() -> String? {
9          return privateStorage.popLast()
10     }
11 }
12
13 var stack = Stack()
14
15 stack.push("Hello")
16 stack.push("World")
17 stack.push("?")
18 stack.pop()
19 stack.push("!")
```

Добавим ему поддержку протокола Collection через расширение. Для этого реализуем минимальный необходимый набор для поддержки протокола.

assossiatedtype указывать явно не обязательно. Swift сам поймет, что Element - это String, а Index - это Int.

```swift
extension Stack: Collection {
    var startIndex: Int {
        return 0
    }

    var endIndex: Int {
        return privateStorage.count
    }

    func index(after i: Int) -> Int {
        return i + 1
    }

    subscript(position: Int) -> String {
        return privateStorage[position]
    }
}
```

Помимо них в протоколе имеются и другие *assosiatedtype*. $Iterator$, унаследованный от Sequence, будет иметь тип $IndexingIterator < Stack >$.

Это простой итератор который оборачивает коллекцию и использует ее для получения следующего элемента. *SubSequence* также полученный от Sequence будет иметь тип $Slice\ < Stack >$. Имеет смысл заменить его на тип самой коллекции. Чуть позже я покажу как это сделать. $IndexDistance$ по умолчанию имеет тип Int и используется для указания разницы между двумя индексами. Его менять не нужно. $Indices-$

$DefaultIndices < Int >$. Еще один врапер над коллекцией. Он содержит все валидные индексы коллекции.

Поддержав этот протокол мы получаем бесплатно кучу полезных методов. Можно использовать коллекцию в цикле for in, можно узнать количество элементов в коллекции, использовать в сабскриптах диапазоны, передавать ее в функции в качестве Sequence, сортировать и трансформировать коллекцию. На слайде несколько простых примеров,

а трансформации будут рассмотрены в других лекциях.

```swift
for element in stack {
    print("\(element) ")
}

stack.count // 3
stack.isEmpty // false
```

```
7
8  Array(stack[1...2]) // ["World", "!"]
```

Также можно добавить поддержку ExpressibleByArrayLiteral. Это позволит инициализировать наш стэк с помощью литералов массивов. Все что нам нужно - это добавить один конструктор. Он получает список элементов и в цикле добавляет их в стэк.

```
1  extension Stack: ExpressibleByArrayLiteral {
2      public init(arrayLiteral elements: String...) {
3          for element in elements {
4              push(element)
5          }
6      }
7  }
8
9  let stackFromArrayLiteral: Stack = ["one", "two", "three"]
```

Еще одним важным понятием которое мы сегодня рассмотрим являются индексы. В предыдущих примерах в качестве индекса мы для удобства

использовали Int. Но коллекция может иметь индекс любого сравнимого(Comparable) типа. Именно поэтому не стоит использовать индексы вручную или пытаться найти следующий прибавив единицу. Один из таких примеров это String. Ее индексом является структура String.Index. Для работы с индексами нужно использовать специальные методы коллекции. Рассмотрим некоторые из них.

```
1  let string = "Simple string"
2
3  var indexInString1 = string.index(of: "s")!
4  var indexInString2 = indexInString1
5  indexInString2 = string.index(after: indexInString2)
6  string.formIndex(after: &indexInString2)
7
8  var indexDistance = string.distance(from: indexInString1, to: indexInString2)
9
10 indexInString1 = string.index(indexInString1, offsetBy: indexDistance)
11
12 indexInString1 == indexInString2 // true
```

Еще в коллекции есть два специальных индекса: *startIndex* и *endIndex*. Они указывают на первый элемент и на элемент следующий за последним соответственно. Т.е. *endIndex* не указы-

вает на валидный элемент в коллекции. Это позволяет писать циклы с условиями $while\ index < endIndex$.

До Swift 3.0 индексы могли сами вычислить следующий индекс. Но зачастую это требовало

хранения в индексе ссылки на коллекцию. Это привело к тому, что изменение коллекции при проходе в цикле создавало ненужные копии. Оптимизация копирования при записи уже не действовала в таком случае.

Еще одним важным изменением, связанным с коллекцией является то, что с версии 4.0 String снова стала коллекцией. Возможно вам встретится код написанный на $Swift3.0$. В этой версии для того чтобы пройтись в цикле по всем символам строки нужно было писать for char in string.characters.

Помимо Collection в Swift есть и другие протоколы для коллекций. BidirectionalCollection добавляет в коллекцию возможность получить предыдущий индекс. RandomAccessCollection не добавляет новых методов в коллекцию, но гарантирует доступ к любому индексу за константное время. $MutableCollection$ добавляет возможность изменять содержимое коллекции через сабскрипты. RangeReplaceableCollection позволяет заменять сразу диапазон элементов в коллекции.

Требования к производительности различаются в этих протоколах. Например свойство count RandomAccessCollection имеет сложность O(1), но для других это может быть $O(n)$. Всегда проверяйте описание конкретной реализации коллекции и комментируйте особенности своей если она не укладывается в стандартные требования.

В этой лекции мы рассмотрели протокол Collection, научились работать с индексами и написали свой простой стэк.

4.8. Трансформация коллекций

```swift
// Трансформация коллекций

import UIKit



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


var multipliedResult = [Int]()
for element in arrayOfInts {
    multipliedResult.append(element * 3)
}
```

```swift
multipliedResult // [3, 6, 9, 12, 15]


let sameMultipliedResult = arrayOfInts.map {$0 * 3 }
sameMultipliedResult // [3, 6, 9, 12, 15]


let arrayOfOptionalInts = [10, 9, nil, 7, 6]

//arrayOfOptionalInts.map {
//    if let value = $0 {
//        return value * 3
//    } else {
//        return nil
//    }} // error: ambiguous reference to member 'map'


let multipliedArrayOfOptionalInts = arrayOfOptionalInts.map {
    (value) -> Int? in

    if let value = value {
        return value * 3
    } else {
        return nil
    }}
multipliedArrayOfOptionalInts // Optional: 30, 27, nil, 21, 18


let arrayOfArrayOfInts = [[1, 2, 3],
                          [4, 5],
                          [6, 7, 8]]


arrayOfArrayOfInts.map {
    type(of: $0) // [Int]
}


let increasedArrayOfArrayOfInts = arrayOfArrayOfInts.map {$0.map {$0 + 2 } }
increasedArrayOfArrayOfInts // [[3, 4, 5], [6, 7], [8, 9, 10]]
```

```swift


let arrayOfOptionalArrayOfInts = [[11, 12, 13],
                                  nil,
                                  [15, 16, 17]]

let increasedArrayOfOptionalArrayOfInts = arrayOfOptionalArrayOfInts.map {$0?.map {$0 + 2 } }
increasedArrayOfOptionalArrayOfInts // [[13, 14, 15], nil, [17, 18, 19]]


let multipliedArrayOfNonOptionalInts = arrayOfOptionalInts.flatMap {
    (value) -> Int? in

    if let value = value {
        return value * 3
    } else {
        return nil
    }}
multipliedArrayOfNonOptionalInts // Non-optional: 30, 27, 21, 18


let sameMultipliedResult2 = arrayOfInts.flatMap {$0 * 3 }
sameMultipliedResult2 // [3, 6, 9, 12, 15]


let flatmappedArrayOfArrayOfInts = arrayOfArrayOfInts.flatMap {$0.map {$0 + 2 }}
flatmappedArrayOfArrayOfInts // [3, 4, 5, 6, 7, 8, 9, 10]


let arrayOfNonOptionalArrayOfInts = arrayOfOptionalArrayOfInts.flatMap {$0 }
arrayOfNonOptionalArrayOfInts // [[11, 12, 13], [15, 16, 17]]


let flatmappedArrayOfOptionalArrayOfInts = arrayOfOptionalArrayOfInts.flatMap {$0 ?? [Int]() }
flatmappedArrayOfOptionalArrayOfInts  // [11, 12, 13, 15, 16, 17]


let arrayOfEvenInts = arrayOfInts.filter {$0 % 2 == 0 }
```

```swift
arrayOfEvenInts // [2, 4]


let decreaseSortedArrayOfInts = arrayOfInts.sorted {
    $0 > $1
}
decreaseSortedArrayOfInts // [5, 4, 3, 2, 1]

let sameDecreaseSortedArrayOfInts = arrayOfInts.sorted(by: >)
sameDecreaseSortedArrayOfInts // [5, 4, 3, 2, 1]


[3, 2, 1].sorted() // 1, 2, 3


let sumOfArrayElements = arrayOfInts.reduce(0) {
    intermediateResult, anotherElement in

    return intermediateResult + anotherElement
}
sumOfArrayElements // 15


let sameSumOfArrayElements = arrayOfInts.reduce(0) {$0 + $1 }
sameSumOfArrayElements // 15


let sameSumOfArrayElements2 = arrayOfInts.reduce(0, +)
sameSumOfArrayElements2 // 15



let sameSumOfArrayElements3 = arrayOfInts.reduce(into: 0) {$0 += $1 }
sameSumOfArrayElements3 // 15



let sameArrayOfEvenInts = arrayOfInts.reduce([]) {$1 % 2 == 0 ? $0 + [$1] : $0 }
sameArrayOfEvenInts // [2 , 4]

```

```
132
133 var mutableArrayOfInt = [1, 3, 2]
134 mutableArrayOfInt.swapAt(1, 2)
135 mutableArrayOfInt // [1, 2, 3]
136
137
138 let sequenceOfInts = sequence(first: 1) {$0 + arc4random_uniform(10) }
139 sequenceOfInts.prefix(100).reduce(into: []) {$0 += [($0.last ?? 0) +
140                         $1] }.map(){$0 + 3 }.reversed().forEach {print("\($0)")}
```

О проекте
Академия e-Legion — это образовательная платформа для повышения квалификации в мобильной разработке. Слушайте лекции топовых разработчиков, выполняйте практические задания и прокачивайте свои скиллы. Получите высокооплачиваемую профессию — разработчик мобильных приложений.

Программа “iOS-разработчик”

Блок 1. Введение в разработку Swift

- Знакомство со средой разработки Xcode
- Основы Swift
- Обобщённое программирование, замыкания и другие продвинутые возможности языка

Блок 2. Пользовательский интерфейс

- Особенности разработки приложений под iOS
- UIView и UIViewController
- Создание адаптивного интерфейса
- Анимации и переходы
- Основы отладки приложений

Блок 3. Многопоточность

- Способы организации многопоточности
- Синхронизация потоков
- Управление памятью
- Основы оптимизации приложений

Блок 4. Работа с сетью

- Использование сторонних библиотек
- Основы сетевого взаимодействия
- Сокеты
- Парсинг данных

- Основы безопасности

Блок 5. Хранение данных

- Способы хранения данных
- Core Data
- Accessibility

Блок 6. Мультимедиа и другие фреймворки

- Работа с аудио и видео
- Интернационализация и локализация
- Геолокация
- Уведомления
- Тестирование приложений