GeekBrains

Основы языка Swift

Методичка к уроку 4
Классы и структуры

Оглавление

Введение

На предыдущем уроке мы познакомились с коллекциями, узнали что такое массив, множество и словарь и чем они отличаются. Рассмотрели замыкания, а также познакомились с функциями высшего порядка.

На этой лекции вы узнаете про:

- Кортежи
- Структуры
- Классы
- Вложенные типы

Термины, используемые в лекции

Кортежи (или tuples) - это набор значений, которые могут рассматриваться как один объект.

Структура - именованный тип в Swift, который позволяет хранить связанные свойства и поведение.

Класс - именованный тип в Swift, который позволяет хранить связанные свойства и поведение, очень похож со структурой.

Кортежи

Кортежи (или tuples) - это набор значений, которые могут рассматриваться как один объект. Один из популярных способов использования кортежа - возврат значения и функции. Подразумевается, что функция может вернуть один объект, а значит у нас нет возможности вернуть, например, сразу два числа

```
7  var a = 7
8  var b = 3
9
10 func multiply() -> Int, Int {   5 Consecutive statements on a line must be separ...
11     return a * 7, b * 7         2 Expected expression
12 }
```

Но, благодаря кортежам такая возможность появилась, можно вернуть сразу несколько значений из функции, так как tuple рассматривается, как один объект

```
7  var a = 7
8  var b = 3
9
10 func multiply() -> (Int, Int) {
11     return (a * 7, b * 7)
12 }
13 print(multiply())
```

```
(49, 21)
```

Кортеж задается при помощи круглых скобок, например, нужно создать кортеж, который хранит String - название кофе и Double - стоимость кофе.

```
var a = ("Latte", 105.50) // 1 способ
var b: (String, Double) = ("Latte", 105.50) // 2 способ
var c = (name: "Latte", cost: 120) // 3 способ
var d: (name: String, cost: Double) = ("Latte", 105.50) // 4
способ
```

1 способ: создание кортежа без указания типов, это разновидность безымянного кортежа, то есть у хранящихся значений нет имен.
2 способ: создание, безымянного кортежа с указанием типов
3 способ: создание кортежа без указания типов, это разновидность названных кортежей, то есть у каждого значения есть свое имя, например для значения “Latte” - имя “name”, а для стоимости “cost”
4 способ: создание названного кортежа с указанием типов

В кортеже может быть сразу несколько значений, например, в кортеже ниже имеется сразу три значения

```
var a = ("Latte", 105.50, true)
```

Получение и изменение значений в безымянном кортеже

В безымянном кортеже нет имен, поэтому у значений есть порядковый номер и обращение к значению tuple происходит через этот номер. Синтаксис обращения к элементу следующий:
<название кортежа>.<порядковый номер>

Например, есть кортеж coffee, в нем хранится название кофе и стоимость. Название кофе, а именно “Latte” указано первым, поэтому обращение к нему будет через 0, стоимость же указана второй, поэтому она следующая по порядку, 1.

```
var coffee = ("Latte", 105.50)
coffee.0 // Latte
coffee.1 // 105.50
```

Изменить значение можно также по порядковому номеру по следующему синтаксису:
<название кортежа>.<порядковый номер> = <новое значение>
Например, в кортеже необходимо поменять стоимость кофе, для этого мы сначала получаем значение по порядковому номеру стоимости, а именно 1, а затем присваиваем новое значение

```
var coffee = ("Latte", 105.50)
coffee.1 = 110
print(coffee) // ("Latte", 110.0)
```

Получение и изменение значений в названном кортеже

В безымянном кортеже есть имена, поэтому обращение к значению происходит через его имя. Синтаксис обращения к элементу следующий:
<название кортежа>.<имя значение>

Например, есть кортеж coffee, в нем хранится название кофе и стоимость. Название кофе, а именно “Latte” хранится под именем “name”, поэтому обращение к нему будет через “name”, для стоимости указано имя “cost”, поэтому обращение к ней через “cost”.

```
var coffee = (name: "Latte", cost: 105.50)
coffee.name // Latte
coffee.cost // 105.50
```

Изменить значение можно также по имени по следующему синтаксису:
<название кортежа>.<имя значения> = <новое значение>
Например, в кортеже необходимо поменять стоимость кофе, для этого мы сначала получаем значение по имени значения, а именно “cost”,, а затем присваиваем новое значение

```
var coffee = (name: "Latte", cost: 105.50)
coffee.cost = 110
print(coffee) // (name: "Latte", cost: 110.0)
```

Кортеж, как возвращаемое значение из функции

Кортеж может быть возвращаемым значением из функции, причем кортеж может быть как безымянным, так и названным. Для того, чтобы вернуть кортеж, нужно указать его после “->”
Например, необходимо создать функцию, которая будет создавать кортеж с новым видом кофе, исходя из полученного названия и стоимости, для этого после “->” необходимо указать тип, в данном случае это (String, Double), то есть сначала имя, а потом стоимость

```
func createCoffee(name: String, cost: Double) -> (String,
Double) {
    (name, cost)
}
let cappuccino = createCoffee(name: "Cappuccino", cost: 110)
```

Теперь можно обращаться к значениям кортежа cappuccino через их порядковый номер

```
print(cappuccino.0) // Cappuccino
print(cappuccino.1) // 110.0
```

Чтобы вернуть из функции названный массив необходимо после “->” указать не только тип, но и имена для значений кортежа, например, изменим функцию createCoffee выше и будем возвращать не безымянный, а названный кортеж. Для этого перед типами указываем имена.

```
func createCoffee(name: String, cost: Double) -> (name: String,
cost: Double) {
    (name: name, cost: cost)
}
let cappuccino = createCoffee(name: "Cappuccino", cost: 110)
```

Теперь можно обращаться к значениям кортежа cappuccino через их имена

```
print(cappuccino.name) // Cappuccino
print(cappuccino.cost) // 110.0
```

Структуры

Кортежи это круто, у теперь у нас есть несколько переменных, которые могут хранить название кофе и его стоимость в одном объекте, но теперь представим, что у нас появляется больше условий к напитку, например, добавить сахар или нет, обычное молоко, миндальное или вообще без молока, с сиропом или без. И вот, кортеж начинает расти и расти, а функция по добавлению нового напитка обрастает новыми параметрами и неизвестно, сколько их еще будет. В этом случае на помощь приходит структура.
Структура - именованный тип в Swift, который позволяет хранить связанные свойства и поведение
Для обозначения структуры используется ключевое слово struct. Структура имеет следующую конструкцию:

```
struct <название структуры> {
    // структура
}
```

В случае с кофе структура может выглядеть, например, так:

```
struct Coffee {
    var name: String // имя
    var cost: Double // стоимость
}
```

Таким образом, Coffee - структура, которая содержит название и стоимость кофе

```
print(latte.name) // Latte
```

Для получения cost - latte.cost

```
print(latte.cost) // 110.0
```

Свойства

В структуре Coffee name и cost - это свойства структуры. Обращение к свойствам структуры через точку, например, latte - экземпляр структуры Coffee.

```
let latte = Coffee(name: "Latte", cost: 110)
```

Для того, чтобы получить, например, name - необходимо использовать latte.name

Свойства хранений

Свойства хранения (или stored property) - переменная или константа, то есть оно может быть объявлено и как var, и как let
Такое свойство может иметь дефолтное значение, а может быть объявлено в инициализаторе

```
struct Coffee {
    var name: String
    var cost: Double = 110
}
```

В данном случае cost - свойство с дефолтным значением, то есть по умолчанию стоимость кофе будет 110, а вот имя нужно будет передать в инициализаторе

Если свойство задано, как var - его можно изменить

```
struct Coffee {
    var name: String
    var cost: Double = 110
}

var latte = Coffee(name: "Latte")
latte.cost = 200
print(latte.cost)
```

В данном коде сначала создается экземпляр структуры Coffee - latte, затем стоимость изменяется на 200 и в консоль выводится стоимость latte - после изменения она стала 200

Если свойство задано, как константа - изменить его нельзя, даже если latte - переменная.
В данном случае в структуре Coffee name - константа, а cost - переменная.
Создается экземпляр структуры Coffee - latte, как переменная, а затем идет попытка изменить name на “Cappuccino”, но так как name - константа, то попытка проваливается

```
47 struct Coffee {
48     let name: String
49     var cost: Double = 110
50 }
51
52 var latte = Coffee(name: "Latte")
53 latte.name = "Cappuccino"      Cannot assign to property: 'name' is a 'let' constant
```

Если экземпляр структуры задан, как константа - свойства изменить будет нельзя, даже если они заданы, как переменные.
В данном случае в структуре Coffee name и cost переменные.
Создается экземпляр структуры Coffee - latte, как константа, а затем идет попытка изменить name на “Cappuccino”, но так как latte - константа, то попытка проваливается.

```
47  struct Coffee {
48      var name: String
49      var cost: Double = 110
50  }
51
52  let latte = Coffee(name: "Latte")
53  latte.name = "Cappuccino"        Cannot assign to property: 'latte' is a 'let' constant
```

Вычисляемые свойства

Вычисляемые свойства (или computed property) - это свойства, которые предоставляют геттер и опциональный сеттер. Такие свойства по сути не хранят значения, а вычисляют их.
Вычисляемое свойство задается при помощи фигурных скобок, конструкция может быть следующая:

```
var/let <название переменной>: <тип переменной> {
        get {
            // что необходимо рассчитать и получить
        }
        set(<значение, которое передано>) {
            // что необходимо сделать при установке значения
        }
    }
```

Например, расширим структуру Coffe, добавим параметр isSugar - есть ли сахар, isIce - есть ли лед, isMaxSize - максимальный ли размер кофе и size - размер кофе.

```
struct Coffee {
    var name: String
    var isSugar: Bool
    var isIce: Bool
    var cost: Double = 110
    var isMaxSize: Bool = false
    var size: Int {
        get {
            let sugarSize = isSugar ? 50 : 20
```

```
            let iceSize = isIce ? 50 : 10
            let coffeeSize = 100 + sugarSize + iceSize
            return coffeeSize
        }
        set(newSize) {
            isMaxSize = newSize >= 200
        }
    }
}
```

size - вычисляемое свойство, у него есть get и set. В get происходит вычисление значения объема кофе, а в set - что необходимо сделать, когда в size пытаются что-то присвоить.

get вызывается, когда пытаемся получить свойство, например, напечатать в консоль.

Создадим latte - экземпляр структуры Coffee, в которой будет сахар и не будет льда. Затем выведем в консоль значение вычисляемого свойства get

```
struct Coffee {
    var name: String
    var isSugar: Bool
    var isIce: Bool
    var cost: Double = 110
    var isMaxSize: Bool = false
    var size: Int {
        get {
            let sugarSize = isSugar ? 50 : 20
            let iceSize = isIce ? 50 : 10
            let coffeeSize = 100 + sugarSize + iceSize
            return coffeeSize
        }
        set(newSize) {
            isMaxSize = newSize >= 200
        }
    }
}

let latte = Coffee(name: "Latte", isSugar: true, isIce: false)
print(latte.size) // 160
```

В консоль будет выведено 160, так как когда мы вызываем свойство size - происходит расчет, описанный в get, то есть:

1. Рассчитывается sugarSize, так как isSugar - true, значит он равен 50

2. Рассчитывается iceSize, так как isIce -false, значит он равен 10
3. Рассчитывается coffeeSize, а именно 100 + 50 + 10 = 160
4. Возвращается рассчитанное значение

🔥 Даже если isSugar и isIce будет true и size будет 200 - isMaxSize не будет установлен в true, так как блок с set вызван не будет.

Блок set будет вызван, когда в значение пытаются что-то установить, например, изменим значение size на 200

```
var latte = Coffee(name: "Latte", isSugar: true, isIce: false)
latte.size = 200
print(latte.isMaxSize) // true
```

В данном случае вызывается блок set, далее идет проверка, является ли значение которое хочется установить больше или равно 200, если да - isMaxSize станет true, в противном случае false.
В данном кейсе значение, которое хочется установить равно 200, а значит и isMaxSize станет true.

🔥 Хоть и вызывается присваивание (=), при следующем вызове свойство size снова вернет 160, так как заново вызовется блок get и произойдет перерасчет значения.

Часто бывает, что вычисляемое свойство не должно ничего устанавливать, а только рассчитывать значение.
Например, избавимся от свойства isMaxSize, так как оно все равно не всегда имеет актуальное значение, поэтому удалим его из структуры. Блок set в таком случае также теряет свою актуальность.
Когда вычисляемое свойство должно только возвращать какое-то значение - можно оставить только блок get, при этом ключевое свойство “get” указывать не нужно. Изменим структуру и оставим только вычисление size.

```
struct Coffee {
    var name: String
    var isSugar: Bool
    var isIce: Bool
    var cost: Double = 110
    var size: Int {
        let sugarSize = isSugar ? 50 : 20
```

```
        let iceSize = isIce ? 50 : 10
        let coffeeSize = 100 + sugarSize + iceSize
        return coffeeSize
    }
}
```

Такое вычисляемое свойство является get-only property, поэтому присвоить в size ничего нельзя, несмотря на то, что size это var

```
32 var latte = Coffee(name: "Latte", isSugar: true, isIce: false)
33 latte.size = 200          Cannot assign to property: 'size' is a get-only property
34
```

Также computed property может быть только var, даже если оно get-only

```
24     let size: Int {          'let' declarations cannot be computed properties
25         let sugarSize = isSugar ? 50 : 20
26         let iceSize = isIce ? 50 : 10
27         let coffeeSize = 100 + sugarSize + iceSize
28         return coffeeSize
29     }
30 }
```

Наблюдатели

Наблюдатели следят за изменением значений свойств. есть два наблюдателя: willSet и didSet. Чтобы объявить наблюдателя свойств можно воспользоваться следующей конструкцией:

```
var <имя переменной>: <тип переменной> = <значение по
умолчанию> {
        willSet {}
        didSet {}
    }
```

Дефолтное значение объявлять не обязательно, его может и не быть.

willSet - вызывается перед сохранением значаения, а didSet после.
Давайте добавим наблюдателей в переменную cost, чтобы следить за изменением стоимости кофе. Для этого после дефолтного значения открываем фигурную скобку, указываем наблюдателя willSet и закрываем фигурные скобки.
В willSet будет выводится новая стоимость кофе, для этого в круглых скобках после willSet необходимо задать имя переменной для нового значения, в данном случае это newCost

```
struct Coffee {
    var name: String
    var isSugar: Bool
```

```
    var isIce: Bool
    var cost: Double = 110 {
        willSet(newCost) {
            print("Новая стоимость кофе \(newCost)")
        }
    }
}

var latte = Coffee(name: "Latte", isSugar: true, isIce: false)
latte.cost = 150
```

В консоль будет выведено “Новая стоимость кофе 150,0”.

Теперь добавим наблюдатель didSet, чтобы знать, какая стоимость была до изменений, для этого указываем didSet и в круглых скобках указываем имя для старого значения стоимости кофе.

```
struct Coffee {
    var name: String
    var isSugar: Bool
    var isIce: Bool
    var cost: Double = 110 {
        willSet(newCost) {
            print("Новая стоимость кофе \(newCost)")
        }
        didSet(oldCost) {
            print("Старая стоимость кофе \(oldCost)")
        }
    }
}

var latte = Coffee(name: "Latte", isSugar: true, isIce: false)
latte.cost = 150
```

В ходе выполнения данного кода сначала будет выведено следующее:

```
Новая стоимость кофе 150.0
Старая стоимость кофе 110.0
```

Методы

В структурах помимо переменных можно указывать и функции,например, напишем функцию, которая будет выводить в консоль имя и стоимость кофе, для этого в структуре Coffee напишем ключевое слово func, затем название функции, например, getCoffee. Далее в теле функции напишем принт, который будет печатать название и стоимость.

```
struct Coffee {
    var name: String
    var isSugar: Bool
    var isIce: Bool
    var cost: Double = 110

    func getCoffee() {
        print("Название: \(name), стоимость: \(cost)")
    }
}
```

Вызов функции происходит через точку, создадим две переменных, latte и cappuccino и вызовем функцию getCoffee

```
var latte = Coffee(name: "Latte", isSugar: true, isIce: false)
var cappuccino = Coffee(name: "Cappuccino", isSugar: false,
isIce: false)
latte.getCoffee()
cappuccino.getCoffee()
```

В консоль будет выведено следующее:

```
Название: Latte, стоимость: 110.0
Название: Cappuccino, стоимость: 110.0
```

mutating

Давайте попробуем написать функцию в структуре Coffee, которая будет изменять название кофе, функция будет принимать на вход новое название, ав теле изменять переменную name

```
func changeName(newName: String) {
        name = newName
    }
```

Но данный код не запустится, будет ошибка компиляции

```
func changeName(newName: String) {
    name = newName
}
```
Cannot assign to property: 'self' is immutable

Это произошло так как нельзя просто так изменять в методе структуры свойство этой же структуры. Чтобы такое стало возможным необходимо перед функцией указать ключевое слово mutating, то есть конструкция следующая:

```
mutating func <название функции>() {
        // тело функции
    }
```

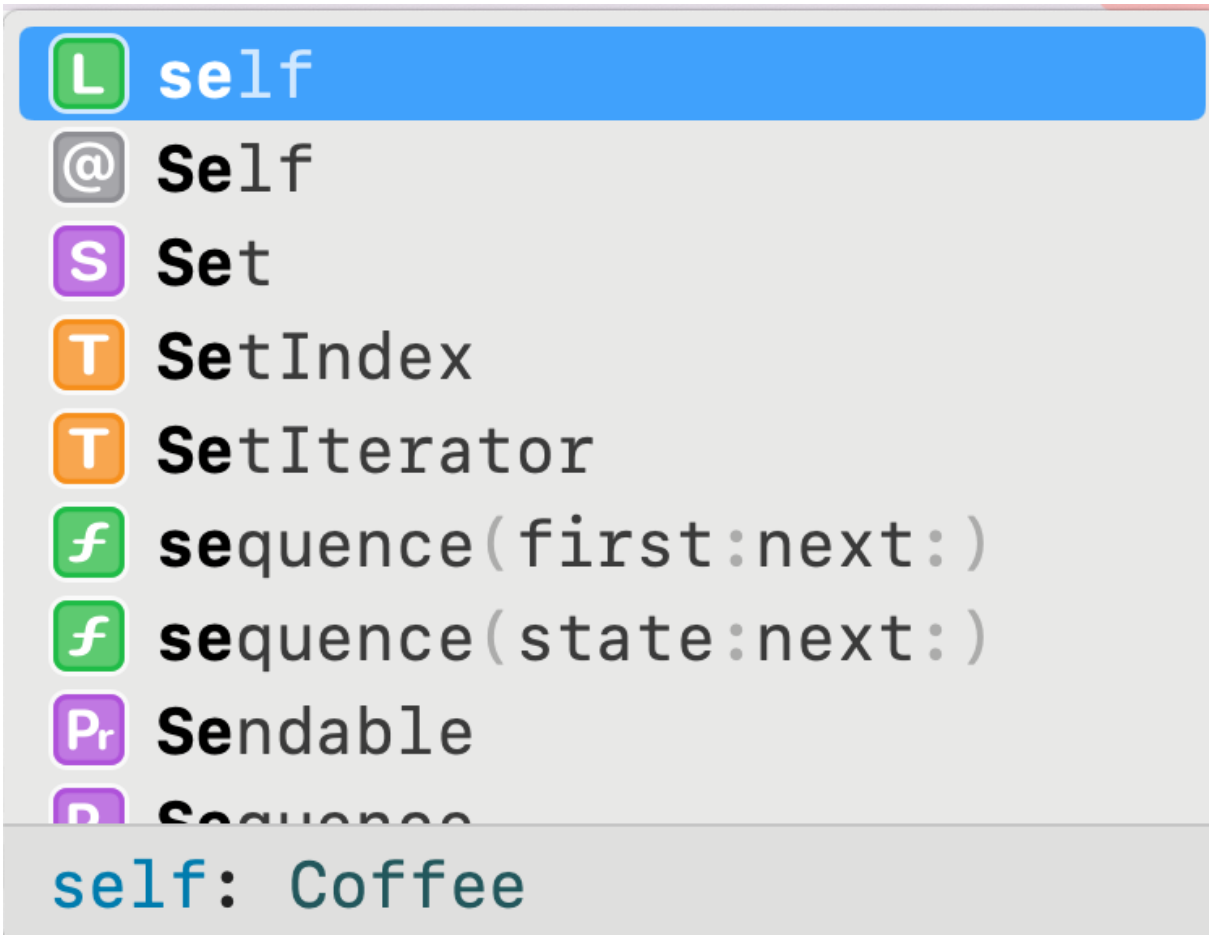
В случае с кофе код будет выглядеть следующим образом, перед func changeName указываем ключевое слово mutating и ошибка исчезает

```
mutating func changeName(newName: String) {
        name = newName
    }
```

Бывают случаи, когда имя свойства и имя переменной, которая передается в функции совпадают, например, изменим newName на name и тогда могут посыпаться ошибки

```
mutating func changeName(name: String) {
    name = name
}
```
Cannot assign to value: 'name' is a 'let' constant

В таком случае на помощь приходит ключевое слово self. self - это сам объект, в котором происходит вызов, например, если указать self внутри структуры Coffee - мы увидим, что это структура Coffee



Для того, чтобы решить проблему с name = name перед первым name, то есть перед свойством структуры, которое необходимо изменить, нужно указать self. То есть будет self.name = name, если расшифровать - это будет значить: присвоить свойству name, которое содержится в структуре Coffee значение name. которое передано в функции

```
mutating func changeName(name: String) {
        self.name = name
    }
```

Инициализатор

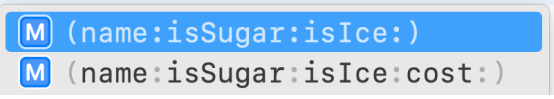
Инициализация - подготовительный этап для создания экземпляра класса, структуры, перечисления. Для объявления инициализатора используется ключевое слово init.

```
init(<переменные>) {
   // инициализатор
    }
```

В инициализаторе происходит настройка экземпляра, например установка значений для всех свойств хранения.

В структуре есть инициализатор по умолчанию, то есть можно не указывать инициализатор специально, например, если вернуться к структуре Coffee: у нее сеть 4 свойства, у одного из которых есть значение по умолчанию. При создании экземпляра класса предлагается два инициализатора, с cost и без. Если будет выбран с cost, то стоимость будет уже не 110, а переданное значение, в противном случае cost останется 110

```
let coffee = Coffee(
```



Давайте добавим свойство size, в котором будет хранится размер кофе. size будет перечислением с тремя кейсами S, M, L.

```
enum CoffeeSize {
    case s
    case m
    case l
}

struct Coffee {
    var name: String
    var isSugar: Bool
    var isIce: Bool
    var cost: Double = 110
    var size: CoffeeSize
}
```

А теперь в CoffeeSize добавим свой собственный инициализатор, который на основе переданного размера (small, medium, large) будет устанавливать нужное значение.

Для этого в CoffeeSize указываем ключевое слово init и внутри реализуем нужную логику, то есть если small значение устанавливается в s, если medium, то m, а в случае large - l, по умолчанию будет m

```
enum CoffeeSize {
    case s
    case m
    case l

    init(size: String) {
        switch size {
        case "small": self = .s
        case "medium": self = .m
        case "large": self = .l
```

```
            default: self = .m
            }
        }
}
```

Теперь создадим переменную, в которой будет хранится размер кофе

```
enum CoffeeSize {
    case s
    case m
    case l

    init(size: String) {
        switch size {
        case "small": self = .s
        case "medium": self = .m
        case "large": self = .l
        default: self = .m
        }
    }
}

let size = CoffeeSize(size: "large")
print(size) // l
```

А теперь создадим latte с таким размером и тут у нас два варианта:

1. Передавать в инициализаторе Coffee переменную size, которая была создана ранее

```
let size = CoffeeSize(size: "large")
var latte = Coffee(name: "Latte", isSugar: true, isIce: false,
size: size)
```

2. Использовать упрощенный инициализатор и не передавать переменную, а сразу указать размер через точку

```
var latte = Coffee(name: "Latte", isSugar: true, isIce: false,
size: .l)
```

В одном объекте может быть не один инициализатор, а сразу несколько, например, предположим, что значение по умолчанию задавать не хотим и тогда нам на помощь придет опциональный инициализатор, он в случае неудачи не проинициализирует объект, а вернет nil. Для того, чтобы указать, что инициализатор опциональный необходимо после init поставить знак вопроса. Добавим в CoffeeSize еще один инициализатор, который в случае неудачи не ставит размер m, а возвращает nil

```
enum CoffeeSize {
    case s
    case m
    case l

    init(size: String) {
        switch size {
        case "small": self = .s
        case "medium": self = .m
        case "large": self = .l
        default: self = .m
        }
    }

    init?(newSize: String) {
        switch newSize {
        case "small": self = .s
        case "medium": self = .m
        case "large": self = .l
        default: return nil
        }
    }
}
```

Теперь создадим две переменные с размерами, ssize и lsize, в первом случае размер укажем “small”, а во втором “sm”

```
let ssize = CoffeeSize(newSize: "small")
print(ssize) // s

let lsize = CoffeeSize(newSize: "sm")
print(lsize) // nil
```

Таким образом, в первом случае размер будет s, а во втором nil.

Уровни доступа

Уровни доступа определяют доступность к свойствам и методам. Для того, чтобы указать уровень доступа для свойства или переменной необходимо указать ключевое слово перед свойством/методом.

```
<уровень доступа> <переменная>
<уровень доступа> <функция>
```

В Swift есть 5 уровней доступа со следующими ключевыми словами:

1. open - разрешает доступ и переопределение в родном модуле и импортирующем модуле. Используется только для классов, их свойств и методов
2. public - разрешает доступ и переопределение в родном модуле, а в импортирующем модуле разрешен только доступ.
3. internal - разрешает использование объекта внутри любого файла исходного моудля, но не в файлах не из этого модуля
4. fileprivate - разрешает использование объекта только в пределах исходного файла
5. private - разрешает использование объекта только в пределах области реализации

💡 По умолчанию уровень доступа internal

Классы

Класс - именованный тип в Swift, который позволяет хранить связанные свойства и поведение, очень похож со структурой, например, в классе так же, как и в структуре можно объявить свойство. Создадим класс Cafe, в котором будет свойство хранения coffee, в котором будет содержаться массив структур Coffee. Сделаем это свойство приватным, чтобы использовать его только внутри класса Cafe.

```
class Cafe {
    private var coffee: [Coffee] = []
}
```

Далее добавим функцию addCoffee, чтобы добавлять новый вид кофе. Для этого указываем ключевое слово func, далее addCoffee, то есть название кофе. В качестве параметра будет передаваться coffee, типа Coffee, то есть конкретный вид кофе.

```
class Cafe {
    private var coffee: [Coffee] = []

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}
```

Смотря на этот код может возникнуть вопрос “Зачем нужна отдельная функция, если можно просто обратиться к переменной и добавить новый элемент в массив?”. В данном случае просто создать экземпляр класса Cafe и обратиться к coffee не получится, ведь coffee имеет уровень доступа private, что делает переменную недоступной вне объявленного класса, только внутри самого класса

```
var cafe = Cafe()
print(cafe.coffee)
```
'coffee' is inaccessible due to 'private' protection level

Также в функции addCoffee можно заметить, что для классов, чтобы изменить свойство, которое содержится в этом классе, не нужно указывать ключевое слово mutating, в отличие от структур.

Инициализатор

Одно из отличий классов от структур - нет инициализатора по умолчанию, то есть, если в структуре, если объявить свойство хранения без дефолтного значения, не будет никакой ошибки, то в случае с классом - ошибка будет. Например, уберем дефолтное значение для переменной coffee

```
class Cafe {
    private var coffee: [Coffee]

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}
```
Class 'Cafe' has no initializers

Чтобы исправить ошибку необходимо написать инициализатор для класса. Добавим init в класс Cafe. В инициализатор будет передаваться массив структур Coffee и это массив будет присваиваться в переменную coffee

```
class Cafe {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}
```

И теперь при создании экземпляра класса необходимо передать массив Coffee. Создадим два вида кофе latte и cappuccino, и передадим их в виде массива в cafe

```
let latte = Coffee(name: "Latte", isSugar: true, isIce: false,
size: .l)
let cappuccino = Coffee(name: "Cappuccino", isSugar: false,
isIce: false, size: .m)

var cafe = Cafe(coffee: [latte, cappuccino])
```

convenience init

convenience init - вспомогательный инициализатор, в нем можно определить вызов основного инициализатора с некоторыми параметрами по умолчанию. Например, у нас есть класс Cafe с основным инициализатором, в котором передается массив Coffee и мы можем создать кафе с любым стартовым набором кофе. Но допустим, необходимо открыть сеть кофеен и у всех есть стандартный набор кофе: латте и капучино. И тогда на помощь приходит convenience init. Мы не будем создавать ряд переменных и в каждую массивом передавать один и тот же набор кофе, а просто напишем вспомогательный инициализатор, в котором уже будет вызываться основной инициализатор со стандартным набором кофе.

```
class Cafe {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}
```

Таким образом, мы все еще можем создать и кафе с собственным набором кофе, например только с латте

```
let latte = Coffee(name: "Latte", isSugar: true, isIce: false, size: .l)
var cafe = Cafe(coffee: [latte])
```

И создать кафе со стандартным набором кофе, то есть с латте и капучино

```
var newCafe = Cafe()
```

🔥 В convenience init вызов self.init обязателен

Ленивые свойства

Ленивые свойства - свойство, начальное значение которого не вычисляется до первого использования.
Ленивые свойства полезны, когда:

1. Начальное значение требует сложных вычислений, которые не должны быть проведены, пока не понадобятся
2. Начальное значение зависит от внешних факторов, значения которых неизвестны до конца инициализации

🔥 Ленивое свойство необходимо создавать как переменную, то есть с ключевым словом var, так как значение может быть не получено до окончания инициализации.

Конструкция для объявления следующая:

```
lazy <переменная>
```

Отличие классов от структур

1. Класс - reference type, а структуры - value type, то есть классы - это ссылочный тип, а структура - тип значение, поэтому классам память выделяется в куче, а структурам - в стеке.

2. Для классов доступно наследование, то есть один класс может отнаследоваться от другого.
3. Для классов доступен deinit, который позволяют экземпляру класса освободить любые ресурсы, которые он использовал.
4. В структуре нельзя просто так изменять в методе структуры свойство этой же структуры, для этого необходимо перед методом указывать ключевое слово mutating. В классах его указывать не нужно.
5. У структур есть дефолтный инициализатор, у классов нет.

Вложенные типы

Вложенные типы (или nested types) - типы (структуры, перечисления, классы), которые могут быть вложены в другой тип. Может быть столько уровней вложения, сколько необходимо. Например, у нас есть структура CoffeeSize, это размер кофе

```
enum CoffeeSize {
    case s
    case m
    case l
}

struct Coffee {
    var name: String
    var isSugar: Bool
    var isIce: Bool
    var cost: Double = 110
    var size: CoffeeSize
}
```

Давайте сделаем CoffeeSize вложенным типом? Ведь размер кофе точно относится именно к кофе и не может быть отдельно от него. Для этого внесем перечисление внутрь структуры Coffee

```
struct Coffee {
    enum CoffeeSize {
        case s
        case m
        case l
    }

    var name: String
    var isSugar: Bool
    var isIce: Bool
```

```
    var cost: Double = 110
    var size: CoffeeSize
}
```

Теперь тип CoffeeSize внутри структуры Coffee можно указать просто CoffeeSize, как например в переменной size. А вне структуры Coffee необходимо указывать тип через точку, то есть сначала идет Coffee, а затем CoffeeSize, то есть вложенный тип.

```
let size: Coffee.CoffeeSize = .l
```

Что можно почитать еще?

1. SwiftBook. Контроль доступа
2. SwiftBook. Свойства
3. SwiftBook. Структуры и классы
4. Разница между классом и структурой
5. SwiftBook. Вложенные типы
6. Кортежи

Используемая литература

1. https://docs.swift.org/swift-book/ReferenceManual/Types.html
2. https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html
3. https://docs.swift.org/swift-book/LanguageGuide/NestedTypes.html