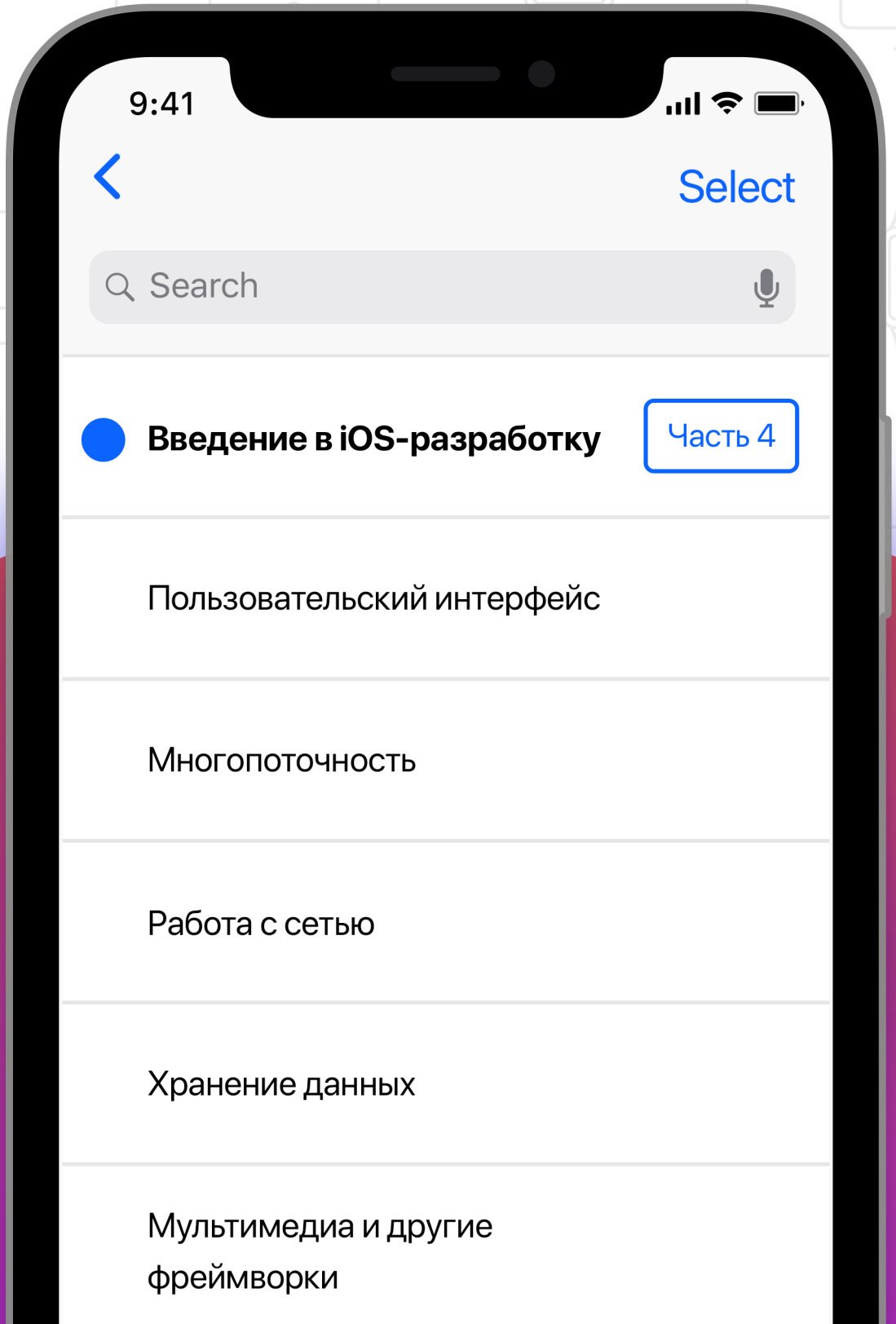
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

Программа iOS-разработчик

Конспект



Оглавление

Глава 4

НЕДЕЛЯ 4

4.1. Жизненный цикл объектов

Привет! В этой лекции мы поговорим о жизненном цикле объектов, инициализаторах и деинициализаторах. Перед началом отмечу, что мы используем термины «конструктор»/«инициализатор», а также «деструктор»/«деинициализатор» как синонимы и не делаем между ними различий.

4.1.1. Инициализатор

Инициализация — это процесс подготовки класса, структуры или перечисления к использованию. Во время него выставляется начальное значение свойствам объекта, для того чтобы перевести его в нужное состояние. Классы дополнительно могут иметь деинициализатор, который вызывается перед уничтожением объекта. Он может быть нужен для освобождения ресурсов. Конструкторы похожи на методы, но объявляются без ключевого слова func. На слайдах вы видите простейший пример — инициализатор без параметров.

```
1  class View {
2      var height: Float
3      var width: Float
4
5      init() {
6          height = 0
7          width = 0
8      }
9      deinit {
10         print("Объект View удален")
11     }
12 }
```

4.1.2. Краткая форма и деструктор

Вызвать его можно так же, как и обычный статический метод, но в Swift есть более краткая форма создания объектов, без указания ключевого слова init. Предпочтительнее использовать второй вариант. Также в этом классе объявлен деструктор. Он вызывается, если присвоить переменной nil. Подробнее об этом мы расскажем в лекции по управлению памятью.

```
let view = View.init()
let anotherView = View()

var optionalView: View? = View()
optionalView = nil
```

4.1.3. Значение по умолчанию

Если свойство всегда принимает одно и то же начальное значение, то лучше использовать значение по умолчанию. Для этого оно просто указывается при объявлении свойства.

```
class View {
    var height = 0.0
    var width = 0.0
    init() {
    }
}
let view = View()
```

4.1.4. Значение по умолчанию

Пустой конструктор описывать необязательно. Swift сам сгенерирует инициализатор по умолчанию, но только в том случае, если всем свойствам вы выставили начальное значение по умолчанию и не объявили ни одного конструктора самостоятельно.

```
class View {
    var height = 0.0
    var width = 0.0
}
let view = View()
```

4.1.5. Memberwise инициализатор

Также Swift умеет генерировать memberwise инициализатор для структур — это инициализатор, в который нужно передать начальное значение для всех свойств. Для этого в структуре не должно быть объявлено ни одного конструктора. Ну а значения по умолчанию в этом случае указывать необязательно. Свои параметры в инициализаторе можно указать точно так же, как и в обычном методе. Обратите внимание, что Swift позволяет выставлять значение константы в инициализаторе, если не было указано значение по умолчанию.

```
struct User {
    var name: String
    var email: String
    var age: Int
}
var user = User(name: "Bob", email:
"bog@mail.com", age: 46)
```

4.1.6. Параметры для инициализатора

Все свойства в классах и структурах обязательно должны либо иметь значения по умолчанию, либо получить какое-то начальное значение в инициализаторе. При этом свойства, объявленные как optional, можно не инициализировать: им по умолчанию присваивается значение nil.

```
class View {
    let height: Double
    let width: Double
    init(side: Double) {
        height = side
        width = side
    }
}
let view = View(side: 5)
```

4.1.7. Делегирование инициализаторов

Чтобы облегчить эту задачу и избежать дублирования кода, используется делегирование инициализаторов то есть мы можем вынести одинаковый код из нескольких конструкторов в другой и просто вызвать его. Рассмотрим пример. В нем мы объявляем структуры, хранящие размер

и положение, а структура Rect содержит в себе по одной такой структуре. Обратите внимание на последний инициализатор. Вместо того чтобы выставлять свойства origin и size, внутри мы просто вызываем другой конструктор с вычисленными параметрами. Этот подход сильно помогает при большом количестве похожих конструкторов. И не забывайте, что инициализатор можно вызвать только из другого инициализатора, но не из метода. В инициализации классов есть свои особенности, связанные с наследованием. Их мы рассмотрим в другой лекции.

```
struct Size {
    var width = 0.0
    var height = 0.0
}
struct Point {
    var x = 0.0
    var y = 0.0
}
```

```
struct Rect {
    var origin = Point()
    var size = Size()

    init() {}

    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }

    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
}
}
```

4.1.8. Failable инициализатор

Иногда возникает такая ситуация, что с переданными в конструктор параметрами нельзя инициализировать объект. Для этого случая в Swift можно вернуть из конструктора nil. Такие

инициализаторы называются failable и помечаются знаком вопроса. Тип создаваемого объекта будет optional. Строго говоря, инициализаторы ничего не возвращают. Мы пишем return nil, только чтобы показать, что инициализация завершилась с ошибкой, но нам не нужно возвращать ничего, если объект был создан успешно.

```
struct Size {
    var width = 0.0
    var height = 0.0
    init() {}
    init?(width: Double, height: Double) {
        guard width >= 0 && height >= 0 else {
            return nil
        }
        self.width = width
        self.height = height
    }
}
let size = Size(width: -10, height: 50)
size // nil
```

4.1.9. Инициализатор для перечислений

С перечислениями с rawValue неявно создается конструктор init(rawValue:), так что нет необходимости в том, чтобы самим проверять, входит ли значение в перечисление.

```
enum TemperatureUnit: Character {
    case kelvin = "K"
    case celsius = "C"
    case fahrenheit = "F"
}
let unknownUnit = TemperatureUnit(rawValue: "X")
if unknownUnit == nil {
    print("Неизвестная единица измерения")
}
```

4.1.10. Вызов Failable инициализатора

Из такого конструктора можно вызвать другой, опциональный или обычный конструктор.

В случае, если любой из них завершился с ошибкой, вся инициализация прерывается. Можно вызвать failable конструктор и из обычного, но для этого придется воспользоваться force unwrap. Как и в большинстве других ситуаций, использование его крайне нежелательно.

```
class MyClass {
    init?(someValue: Int) {
    }
    convenience init() {
        self.init(someValue: 5)!
    }
}
```

4.1.11. Замыкания в инициализаторах

Еще одна интересная возможность Swift — это использование замыканий для выставления начальных значений свойствам. Рассмотрим пример. У класса View есть свойство frame, но его значение не выставляется, как в предыдущих примерах, а вычисляется. Обратите внимание на скобки после окончания объявления замыкания. Они означают, что замыкание нужно вызывать при инициализации. Если написать выражение без них, то это будет присвоение самого замыкания в свойство frame, а не его результата. Следует помнить, что в момент вызова замыкания объект еще не создан, поэтому вы не можете обращаться к другим свойствам или вызывать методы. В этом видео мы рассмотрели создание и удаление объектов в Swift. Продолжение темы смотрите в лекции по инициализации классов после наследования.

```
class View {
     var frame: Rect = {
        let randomX = Double(arc4random_uniform(100))
        let randomY = Double(arc4random_uniform(100))

        let position = Point(x: randomX, y: randomY)

        return Rect(center: position, size: Size(width: 50.0, height: 50.0)!)
    }()
}
```

4.2. Управление памятью

Привет! В этой лекции мы рассмотрим управление памятью. Как Swift решает, что объект можно удалить и как мы можем повлиять на эту логику. В Swift нет сборщика мусора, но управление памятью происходит автоматически. Для этого используется Automatic Reference Counting, или сокращенно ARC. Эта система пришла в Swift из Objective C.

4.2.1. Ссылки на объект

Давайте рассмотрим, как она работает. Идея проста: когда вы создает объект ARC создает счетчик указателей на него. Каждый раз, когда вы присваиваете этот объект какой-нибудь переменной, константе или свойству, этот счетчик увеличивается. При удалении указателя — уменьшается. Если он достигнет нуля, то такой объект считается ненужным и удаляется из памяти. Рассмотрим пример. В нем мы объявим простой класс с инициализатором и деинициализатором, чтобы можно было понять, был ли объект удален. В комментариях указано количество ссылок на объект. Когда reference1 присваивается reference2, то новый объект не создается — это reference type. Вместо этого ARC просто увеличивает количество ссылок. Когда обнуляются обе ссылки, экземпляр будет удален.

```swift
class TestClass {
    init() {
        print("TestClass создан")
    }
    deinit {
        print("TestClass удален")
    }
}
```

```swift
var reference1: TestClass? =
TestClass() // 1

var reference2 = reference1 // 2
print("Две ссылки")

reference1 = nil // 1
print("Одна ссылка")

reference2 = nil // 0
print("Ссылок нет")
```

4.2.2. Reference cycle

Рассмотрим пример посложнее. Объявлены классы для хранения пользователя и устройства. У пользователя есть массив: его устройство и имя, у устройства — модель и владелец. Для таких целей лучше использовать структуры, но в данном примере используются классы, чтобы не усложнять код описанием ситуации, где действительно нужны классы.

```
class User {
    let name: String
    var devices: [Device]
    init(name: String) {
        self.name = name
        devices = [ ]
        print("Создан пользователь \(name)")
    }
}
```

```
class Device {
    let model: String
    let owner: User
    init(model: String, owner:
    User) {
        self.model = model
        self.owner = owner
        print("Создано устройство \(model).
        Его владелец \(owner.name)")
    }
}
```

4.2.3. Reference cycle

Создадим пользователя — Боба, добавим ему iPhone 7 и попробуем, как раньше, обнулить ссылки. Запустив этот пример у себя в плейграунде, вы убедитесь, что объекты не были удалены после обнуления ссылок. Произошло это потому, что количество ссылок пользователя и его девайс не равны 0. У объекта Боб есть ссылка на iPhone 7, у айфона — ссылка на его владельца, то есть они ссылаются друг на друга, и никогда не будут удалены. Такая ситуация называется strong reference cycle. Не забывайте, что возникнуть она может не только с парой объектов, ссылающихся друг на друга — их может быть любое количество.

```
var bob: User? = User(name: "Bob")
var iPhone7: Device? = Device(model:"iPhone 7", owner: bob!)
bob!.devices.append(iPhone7!)

bob = nil
iPhone7 = nil
```

4.2.4. Weak reference

Для того чтобы решить эту проблему, существуют слабые ссылки. В отличие от обычных, сильных, ссылок они не учитываются ARC. Если на объект ссылаются слабые ссылки, но нет ни одной сильной, то объект все равно будет удален. При этом все weak-ссылки будут обнулены. Сделаем owner слабой ссылкой. Обратите внимание на то, что owner теперь объявлена как переменная, а не константа, и тип ее — optional. Это логично, ведь слабая ссылка будет обнуляться при удалении объекта, на который она ссылается.

```
class User {
    let name: String
    var devices: [Device]
    init(name: String) {
        self.name = name
        devices = [ ]
        print("Создан пользователь \(name)")
    }
    deinit {
        print("Удален пользователь \(name)")
    }
}
```

```
class Device {
    let model: String
    weak var owner: User?
    init(model: String, owner:
    User) {
        self.model = model
        self.owner = owner
        print("Создано устройство \(model). Его владелец \(owner.name)")
```

```
9       }
10      deinit {
11          print("Удалено устройство \(model)")
12      }
13  }
```

4.2.5. Weak reference

Теперь все объекты будут корректно удаляться. Запустив, исправленную версию в плейграунде, можно в этом убедиться. Помимо weak, есть еще ключевое слово unowned. Разница между ними в том, что unowned-ссылки не обнуляются при удалении объекта, на который они указывают. Так как Swift — безопасный язык, то при обращении к такой переменной, он не позволит повредить данные и завершит выполнение программы. Это поведение можно изменить при помощи ключевого слова unowned unsafe. В этом случае мы получим undefined behavior, то есть поведение приложения нельзя будет предсказать. Но у unowned есть одно преимущество: не обязательно использовать optional — это позволяет упростить код. В общем случае следует руководствоваться следующим правилом: если время жизни ссылки такое же, как и у объекта, на который она указывает, то можно использовать unowned.

```
1  var bob: User? = User(name: "Bob")
2  var iPhone7: Device? = Device(model:"iPhone 7", owner: bob!)
3  bob!.devices.append(iPhone7!)
4  bob = nil
5  iPhone7 = nil
```

4.2.6. Reference cycle

Но не только классы являются ссылочными типами в Swift — при использовании замыканий тоже может возникнуть strong reference cycle. Рассмотрим пример. Есть класс counter, у него свойствосодержащее closure expression. В этом замыкании есть обращение к self. Это приводит к тому, что замыкание захватит self и возникнет strong reference cycle.

```
1  class Counter {
2      var value: Int = 0
3      lazy var valueChanger = {
4          self.value += 1
```

```
    }
    deinit {
        print("Counter удален")
    }
}
var counter: Counter? = Counter()
counter = nil
```

4.2.7. Weak reference

Для решения этой проблемы тоже используется weak и unowned. Указываются они в списке захвата, перед названием константы. В данном случае использовался unowned, потому что замыкание не может существовать без объекта Counter. Однако не следует добавлять такой захват в каждое замыкание. Если в нем нет обращения к объекту, в котором оно хранится, то и проблем с памятью не возникнет. Также обратите внимание на то, является ли замыкание escaping. Если нет, то и strong refernce cycle возникнуть не может. Все вышесказанное относилось только к reference type. У значимых типов нет посчета ссылок — при присваивании новой переменной создается копия, поэтому у них всегда один владелец. При его удалении просто удалятся все объекты, которыми он владеет. На этом лекция по управлению памятью закончена. Мы разобрались, как работает счетчик ссылок, что такое strong reference cycle и какие существуют способы его избежать.

```
class Counter {
    var value: Int = 0
    lazy var valueChanger = {
        [unowned self] in
        self.value += 1
    }
    deinit {
        print("Counter удален")
    }
}
var counter: Counter? = Counter()
counter = nil
```

4.3. Наследование

Привет! В этом видео мы поговорим о наследовании. Класс может наследовать методы, свойства и другие характеристики от другого класса. Класс, наследующий от другого, называется подклассом или сабклассом. Класс, от которого наследуется, называется родительским или суперклассом. Наследование — это основное отличие классов от других типов Swift. Классы могут вызывать методы, использовать свойства и сабскрипты либо могут переопределять их для расширения возможностей. Компилятор проверит существуют ли методы при переопределении во избежание ошибок. Любой класс, который не наследуется от другого класса, является базовым. В отличие от Objective-C вам не обязательно указывать при создании нового класса один из базовых, например, NSObject или NSProxy.

4.3.1. Базовый класс

В нашем примере мы объявили базовый класс Animal, который дальше будет расширять свой функционал сабклассов.

```
class Animal {
    var weight = 0.0
    var description: String {
        return "weight is: \(weight)"
    }
    func makeSound() {
        print ("Default animal sound")
    }
}
```

4.3.2. Создание инстанса

Теперь создаём instance класса и получаем возможность обращения к его свойствам и методам. Базовый класс определяет стандартное поведение для класса, но оно не всегда достаточно для полноценной работы с ним.

```swift
let animal = Animal()
print(animal.description)
```

4.3.3. Подкласс

А наследование — это процесс определения нового класса на основе существующего. Для этого у нового класса после двоеточия указывается класс, от которого он будет наследован. Класс Dog наследуется от класса Animal. В дополнение к характеристика, которые он получает от родительского класса, он определяет и собственные свойства. Теперь появляется возможность работы не только с новыми, но и с унаследованными свойствами.

```swift
class Dog: Animal {
    var color = UIColor.red
}
let dog = Dog()
dog.weight = 10.0
dog.color = UIColor.yellow
print(dog.description)
// Animal weight is: 10.0
```

4.3.4. Переопределение в подклассе

Сабкласс может определять собственную реализацию для методов и переменных. Этот процесс называется переопределением. Для переопределения характеристики она помечается ключевым словом override. Таким образом вы сообщаете о том, что собираетесь изменить реализацию существующей характеристики, а не определяете новую. Override помогает компилятору проверить то, что у родительского класса имеется данный метод или переменная и не позволяет вам определить характеристику с уже существующим именем, если вы не используете это ключевое слово. Иногда при переопределении вам может понадобиться доступ к базовой реализации метода. В этом случае вам поможет ключевое слово super, которое даёт вам доступ к базовому классу. Переопределение методов помогает изменить реализацию на подходящую вашему классу. Посмотрим это на простом примере. В базовом классе у нас определён метод makeSound. Но у него пустая реализация. Создаём подкласс и переопределяем реализацию данного метода. Далее проверяем, что из этого выходит.

```swift
class Cat: Animal {
    override func makeSound() {
        print("Meow!")
    }
}
let cat = Cat()
cat.makeSound() // Meow!
```

4.3.5. Переопределение getter и setter

Переопределние свойства позволит написать собственные сеттеры и геттеры для свойств, а также переопределить наблюдатели для этих свойств, чтобы получить оповещения об изменении в базовом классе. Переопределять можно как вычисляемые, так и хранимые свойства. Важно лишь правильно указать тип и имя переопределяемого свойства. Для свойств, имеющих базовой реализацией только геттер, можно определить и геттер и сеттер. Однако, для свойств, доступных как для чтения, так и для записи, нельзя переопределить только геттер. Если переопределяется сеттер, то геттер также должен быть определён. Простейшая реализация может быть выполнена при помощи обращения к свойству базового класса.

```swift
class EnglishDog: Dog {
    override var description: String {
        get {
            return "weight in pounds: \(weight * 2.2)"
        }
    }
}
let bulldog = EnglishDog()
bulldog.weight = 5
print(bulldog.description)
```

4.3.6. Переопределение обозревателей

Переопределение свойств можно использовать для задания собственных обозревателей. Вы получите уведомление об изменении свойств независимо от их внутренней реализации. Важно отметить, что свойств, доступных только для чтения или определённых как константы, нельзя назначить обозревателю didSet и willSet, так как значения таких свойств не могут быть изменены.

Также не получится одновременно переопределить сеттер и установить обозреватель для него. Если вы хотите узнать об изменениях переменной, то отслеживайте эти изменения прямо в сеттере, который создали. В нашем примере мы переопределяем переменную и устанавливаем для неё обозреватель, в котором лимитирует максимально возможное значение для установки веса птиц.

```
class LimitedEnglishDog: EnglishDog {
    var weightLimit: Double?
    override var weight: Double {
        didSet {
            if let weightLimit = weightLimit, (weight >
            weightLimit) {
                weight = oldValue
                print("Слишком большой вес! Пора на прогулку!")
            }
        }
    }
}
let bloodhound = LimitedEnglishDog()
bloodhound.weightLimit = 119
bloodhound.weight = 130 // Слишком большой вес!
                        // Пора на прогулку!
```

4.3.7. Ограничение переопределения

Переопределение методов и свойств класса можно ограничить. Ключевое слово final необходимо именно для этого. В случае переопределения таких методов или свойств, компилятор отобразит ошибку. А также целый класс может быть отмечен как final. Любая попытка наследования от него приведёт к ошибке. В этом видео мы познакомились с основами наследования и переопределения методов и свойств классов. В следующем материале будет рассказано о том, как лучше использовать эти знания на практике.

```
class DontOverrideMyProperties {
    final var description: String {
        return "Лучшее описание"
    }
}
class IDontReadClassNames:
```

```
DontOverrideMyProperties {
    override var description: String {
        return "Почему здесь ошибка?"
    }
}
```

4.4. Инициализация классов

Привет! В этом видео мы продоложим тему инициализации объектов, рассмотрим, какие есть особенности в создании экземпляров классов. В классах, как и в структурах, обязательно при инициализации выставить начальное значение для всех свойств. Но у классов всё усложняется из-за наследования.

4.4.1. Convenience инициализатор

Для того чтобы упростить соблюдение этого правило используется понятие designated initializers и convenience initializers. Designated initializer — это основной инициализатор класса. Он выставляет начальные значения всем свойствам, объявленным в этом классе, и вызывает конструктор родительского класса, для того чтобы он выставил значения своим полям. Так образуется вертикальная цепочка инициализаторов. Остальные конструкторы считаются дополнительными и объявляются с ключевым словом convenience. Одни должны либо вызвать основной конструктор, либо какой-то другой вспомогательный. Таким образом, выстраивается горизонтальная цепочка инициализации. В классе должен быть как минимум один designated initializer. Как правило, и объявляется один, но в некоторых случаях бывает и два-три таких конструктора. Convenience initializers объявлять необязательно, если у вас нет какого-то специфичного кейса, который было бы удобно вынести в отдельный конструктор.

```
struct FuelTank {
    let fuelConsumption: Float
    let fuelTankCapacity: Float
}
class Vehicle {
    let fuelTank: FuelTank
    init(fuelTank: FuelTank) {
        self.fuelTank = fuelTank
    }
}
```

```
11  extension Vehicle {
12      convenience init(fuelConsumption: Float,
13      fuelTankCapacity: Float) {
14          self.init(fuelTank: FuelTank(fuelConsumption: fuelConsumption,
15                                                        fuelTankCapacity: fuelTankCapacity))
16      }
17  }
```

4.4.2. Designated и Convenience инициализаторы

Рассмотрим пример. В классе Vehicle объявлен конструктор, принимающий стуктуру FuelTank в качестве параметра для инициализации. В расширении мы создаём инициализатор, который будет самостоятельно создавать структуры из переданных параметров и вызывать designated initializer. Ещё один принцип, помогающий уменьшить количество возможных ошибок при инициализации классов, это двухфазная инициализация. Во время первой фазы выставляются начальные значения для всех свойств. Во второй фазе каждый субкласс получает возможность изменить эти значения. Рассмотрим этот процесс подробнее, представим, что мы вызвали convenience конструктор. В этот момент и до конца первой фазы объект считается не инициализированным. Мы не можем обращаться к свойствам методом этого экземпляра. Дальше конструктор обязан вызвать другой конструктор этого класса, до того как он начнёт выставлять значения свойствам, иначе его изменения затрутся в других конструкторах. Допустим, он вызывает designated initializer. Он должен выставить начальное значение всем свойствам, объявленным в этом классе и затем вызвать один из родительских конструкторов. Он должен это сделать до того, как начнёт выставлять значения свойствам, унаследованным от родительских классов. Иначе, его изменения затрутся в родительских инициализаторах. Дальше вызовы конструкторов идут вверх по иерархии классов. После того, как выполнил инициализацию базовый класс, первая фаза заканчивается. Объект считается полностью инициализированным. Дальше по мере выхода из вызванных конструкторов каждый субкласс выставляет свои изменения для унаследованных им переменных. В этот момент уже можно вызывать методы и читать значения свойств. Так как это происходит в обратном порядке, то более специализированные классы получают преимущество в выставлении значений, перетирая значения родительских классов. В большинстве случае инициализаторы не наследуются в дочернем классе. Это сделано потому, что в дочернем классе могут быть объявлены дополнительные свойства, а в родительском конструкторе они не инициализируются.

```
1  class MyClass {
2      var intProperty: Int
```

```
3       init(intProperty: Int) {
4           self.intProperty = intProperty
5       }
6       convenience init() {
7           self.init(intProperty: 2)
8       }
9   }
```

4.4.3. Инициализаторы в подклассах

Пример: Объявлен класс с двумя конструкторами, один - *designated*, другой - *convenience*.

Для того чтобы в его субклассе был конструктор с таким же названием, нужно предоставить новую реализацию и добавить ключевое слово override. При этом нужно обязательно вызвать какой-нибудь родительский инициализатор, так как это designated *initializer*. Однако, для convenience слово *override* писать не нужно. Субкласс, строго говоря, ничего не переопределяет в этом случае. Если вы предоставляете значения по умолчанию для всех свойств объявленных в субклассе, то для наследования и инициализаторов используются следующие два правила. Если субкласс не объявляет ни одного designated инициализатора, то он наследует их от родительского класса. Если субкласс переопределяет все *designated* инициализаторы, то он наследует и все *convenience* инициализаторы.

```
1   class MySubClass: MyClass {
2       var doubleProperty: Double
3       convenience init() {
4           self.init(intProperty: 5)
5       }
6       override init(intProperty: Int) {doubleProperty = 3
7           super.init(intProperty: intProperty)
8       }
9   }
```

4.4.4. Переопределение Failable инициализатора

Failable инициализатор можно переопределить как failable или как обычный, однако, во втором случае нужно сделать force unwrap при вызове родительского конструктора. Если он выполнился с ошибкой, то приложение завершит работу.

```swift
class A {
    init?() {
    }
}
class B: A {
    override init() {
        super.init()!
    }
}
```

4.4.5. Required инициализаторы

Последяя особенность конструкторов, которую мы сегодня обсудим, это required инициализаторы. Если конструктор помечен ключевым словом required, то в его субклассах нужно обязательно предоставить его реализацию. Override в этом случае добавлять не нужно, так как это было бы излишне. Однако, не забывайте, что в некоторых случаях инициализаторы наследуются. Если вам это подходит, то воспользуйтесь этой возможностью, вместо того чтобы писать пустые конструкторы. Сегодня мы обсудили особенности инициализации классов, разобрались, как воспользоваться наследованием конструктором и как их правильно переопределять.

```swift
class C {
    required init(someValue: Int) {
    }
}
class D: C {
    required init(someValue: Int) {
        super.init(someValue: someValue)
    }
}
```

4.5. Extensions

Всем привет. Лекция посвящена экстеншенам. Экстеншены, или расширения, добавляют новый функционал для существующего класса, структуры, перечисления или протокола. Можно даже расширить возможности типа, к исходным кодам которого у вас нет доступа. Экстеншены похожи на категории из Objective-C, но в отличие от категорий не имеют имен.

4.5.1. Что могут добавлять Extensions

Что могут добавлять экстеншены? Вычисляемые свойства как для типа, так и для инстанса. Определять новые методы, создавать инициализаторы, добавлять сабскрипты, определять и использовать новые вложенные типы, а также добавлять поддержку протокола для существующего типа. Для протоколов с помощью расширений можно создать готовую реализацию для требований или добавить функционал, которым сможет воспользоваться тип, поддерживающий протокол. У экстеншенов есть ограничения. Они могут добавлять новый функционал, но не могут переопределять его.

- Вычисляемые свойства
- Новые методы
- Создавать инициализаторы
- Добавлять сабскрипты
- Определять и использовать вложенные типы
- Добавлять поддержку протоколов

4.5.2. Объявление Extension

Расширения объявляются с помощью ключего слова extension, далее следует тип, а также можно объявить о поддержке протоколов.

```
1  extension SomeType: SomeProtocol,
2  AnotherProtocol {
3   //
4   // Код
5   //
6  }
```

4.5.3. Расширяем тип String

Расширения могут добавлять вычисляемые свойства для существующих типов, однако нельзя добавить хранимые свойства или обозреватели для свойств. Простой пример показан на слайде. Мы добавили новые свойства для типа string. Теперь для любого инстанса этого типа будет возможность использовать новый функционал.

```swift
extension String {
    var upperCaseDescription: String {
        return "Upper case: " + self.uppercased()
    }
}
print("Hello world".upperCaseDescription)
// Upper case: HELLO WORLD
```

4.5.4. Создаем инициализатор

Следующее, что мы можем сделать с помощью расширений — это добавить новые инициализаторы. Это позволит создавать для типов инициализаторы с такими парметрами, которые будут удобны для вас. Вы можете передать в инициализатор собственный тип или расширить инициализатор с помощью дополнительных параметров. Добавлять можно только convenience-инициализаторы. Designated-инициализаторы или деинициализаторы добавлять не получится, они должны быть реализованы в оригинальной имплементации типа. Если вы добавляете инициализатор для значимого типа, у которого устанавливаются все значения для хранимых свойств и для него не определяются какие-то дополнительные инициализаторы, тогда вы можете в своей реализации вызвать инициализатор по умолчанию для этого типа. Если вы создаете подобный инициализатор, то на вас лежит ответственность за инициализацию для каждого из свойств для этого типа.

```swift
extension String {
    init(left: String, right: String) {
        self = left + right
    }
}
let helloWorld = String.init(left: "Hello",
right: "World")
print(helloWorld)
// HelloWorld
```

4.5.5. Инициализатор в extenstion

На слайдах показан пример, в котором определена структура. Так как каждое свойство структуры снабжается значением по умолчанию, мы можем создать собственный инициализатор в

расширении для этой структуры и вызвать сгенерированный автоматический инициализатор для этого типа. Подобные решения помогут вам в создании объектов иным способом, нежели изначально было задумано.

```swift
struct Display {
    var width: Float = 0.0
    var horizontalResolution: Float = 0.0
    var ppi: Float = 0.0
}
extension Display {
    init(width: Float, horizontalResolution: Float) {
        var ppi = horizontalResolution / width
        self.init(width: width, horizontalResolution: horizontalResolution, ppi: ppi)
    }
}
let display = Display(width: 15.0, horizontalResolution: 1920)
print("PPI: ", display.ppi)
// PPI: 128.0
```

4.5.6. Методы в extension

Помимо инициализаторов и свойств с помощью расширения можно добавить новые методы. Берем один из стандартных типов Swift и объявляем для него новые методы. Эти методы могут изменять непосредственно инстанс, используя ключевое слово self. Если тип относится к значимому, то есть является структурой или перечислением, то метод, вносящий изменения в self или изменяющий свойства этого типа, должен быть помечен ключевым словом mutating, точно так же как для оригинальной реализации типа. Пример, показанный на слайде, демонстрирует, как это должно быть выполнено.

```swift
extension String {
    func makeMePickle() {
        print("I'm pickle", self)
    }
    mutating func doubleString() {
        self = self + " " + self
    }
}
var name = "Rick"
name.doubleString()
```

```
name.makeMePickle()
// I'm pickle Rick Rick
```

4.5.7. subscript в extension

С сабскриптами все обстоит точно так же. Создаем расширение и описываем сабскрипт для существующего типа. И снова, воспользуемся одним из стандартных типов Swift и добавим новые возможности для него.

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}
print(4815162342[4])
// 6
```

4.5.8. nested type в extension

Следующей возможностью для расширения являются вложенные типы. В данном случае нет каких-либо сложностей или ограничений со стороны языка. Код, необходимый для реализации, показан на слайде.

```
struct Display {
    var width = 0.0
    var height = 0.0
    var ppi = 0.0
}
extension Display {
    struct Resolution {
        var horizontal = 0.0
        var vertical = 0.0
}
```

```
    var resolution: Resolution {
        return Resolution(horizontal: width * ppi, vertical: height * ppi)
    }
}
var resolution = Display.Resolution(horizontal: 1920, vertical: 1080)
var display = Display(width: 16, height: 9, ppi: 320.0)
print(display.resolution)
// Resolution(horizontal: 5120.0, vertical: 2880.0)
```

4.5.9. Группировка в private extension

Теперь хотелось бы рассказать о практическом применении расширений. Чаще всего мы используем экстеншен для организации кода. Конечно, вы можете написать весь код в одном блоке, но если вы сгруппируете реализацию некоторых методов в отдельные расширения, это поможет вам лучше ориентироваться в коде и проще читать его. Например, мы можем создать private extension, в котором будут реализованы только те методы, которые не должны быть доступны извне. Пример на слайде демонстрирует подобную реализацию расширения. Даже если мы не писали код, то одного взгляда на него будет достаточно, для того чтобы понять, с какими методами смогут взаимодействовать другие модули, а с какими нет.

```
struct Display {
    var width = 0.0
    var height = 0.0
    var ppi = 0.0
}
private extension Display {
    func diagonalSize() -> Double {
        return sqrt(width * width + height * height)
    }
}
var display = Display(width: 16, height: 9,
ppi: 320.0)>
print("\(display.diagonalSize())")
// Diagonal size: 18.3575597506858
```

4.5.10. Группировка схожих по функционалу методов

Следующим примером может служить группировка схожих по функционалу методов. Например, мы создали класс и выделили в отдельный экстеншен те функции, которые, по нашему мнению, будут выполнять схожие задачи.

```
extension String {
    func pringString(nTimes: Int) {
        var resultingString = ""
        for _ in 1...nTimes {
            resultingString.append(self)
        }
            print(resultingString)
    }
}
extension String {
    func printDoubleString() {
        pringString(nTimes: 2)
    }
    func printTrippleString() {
        pringString(nTimes: 3)
    }
}
```

```
extension String {
let name = "Snufkin"

name.printDoubleString()
// SnufkinSnufkin

name.printTrippleString()
// SnufkinSnufkinSnufkin
```

4.5.11. Реализация протокола в extension

Отдельным случаем для группировки может служить поддержка протоколов.

Например, для реализации таблицы вам нужно поддержать два протокола: UITableViewDataSource и UITableViewDelegate. Почему бы не вынести поддержку и реализацию этих протоколов в два

разных расширения? Вы смотрите на код и мгновенно понимаете, какую его часть вы хотите изменить, или реализацию какого протокола вам в данный момент нужно посмотреть.

```
extension String {
class MainController:
UIViewController {
    var model = [0, 1, 2, 3, 4, 5]

    override func viewDidLoad() {
        let tableView = UITableView()
        tableView.dataSource = self
    }
}
```

```
extension MainController:
UITableViewDataSource {
    func numberOfSections(in tableView: UITableView) -> Int
    {
        return 1
    }
    func tableView(_ tableView: UITableView, numberOfRowsInSection section:
    Int) -> Int {
        return model.count
    }
    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
        return UITableViewCell()
    }
}
```

4.5.12. Computed properties в extension

Помимо этого, при реализации вашего типа можно в оригинальной имплементации определить хранимые свойства, а в экстеншен вынести реализацию вычисляемых свойств. Скорее всего, при разработке вы найдете удобные для вас случаи использования расширений или почерпнете новые идеи от своих коллег. Мы в свою очередь по мере обучения будем показывать вам новые примеры и способы работы с расширениями.

```swift
struct Display {
    var width = 0.0
    var height = 0.0
    var ppi = 0.0
}
extension Display {
    var diagonalSize: Double {
        return sqrt(width * width + height * height)
    }
}
```

4.6. Контроль доступа

Привет! Это видео посвящено разграничению доступа к методам, переменным и инициализаторам типов. Контроль доступа помогает скрыть детали реализации вашего типа и определить правильный интерфейс. Уровень доступа может быть определен как для типа, так и для отдельных его составляющих.

4.6.1. Уровни доступа

Для начала разберемся с теорией. В Swift разрешено изменять доступ к модулям и исходным файлам. Под модулем подразумевается код, входящий в состав фреймворка или приложения, который распространяется, как единое целое, и может быть добавлен в проект в качестве ключевого слова Import. Каждая цель для сборки приложения считается отдельным модулем. Все, что вы определите в фреймворке для инкапсулирования логики приложения с целью дальнейшего переиспользования, будет доступно в качестве отдельного модуля. Этот модуль смогут использовать как приложения, так и другие фреймворки. Файл с исходным кодом является частью модуля и содержит описание одного или нескольких типов внутри себя. В Swift определены 5 уровней доступа. Open и Public доступы позволяют обращаться к сущностям из любого модуля. Используйте эти модификаторы для описания общих интерфейсов. Public позволяет использовать наследование и переопределение только внутри модуля, в котором они объявлены. Open же может использоваться только классами и членами классов, но позволяет наследоваться и выполнять переопределение любому модулю, который импортировал наш код. Internal дает возможность работать с сущностями в рамках существующего модуля. File-private ограничивает доступ рамками текущего файла. Используется для сокрытия деталей реализации участка кода, когда реализация осуществляется в рамках одного файла. Private же позволяет работать с

сущностями внутри определяющего их блока. Или в расширении, но только в текущем файле. Уровни доступа описаны от самого доступного, до самого ограниченного. Важно запомнить то, что сущность с более высоким уровнем доступа не может быть определена внутри сущности с более низким уровнем доступа. По умолчанию, всем сущностям назначается уровень доступа Internal. Поэтому в большинстве ситуаций определять его явно нет необходимости. При создании приложения с единственной целью — для сборки — Internal удовлетворяет всем необходимым потребностям. И нет необходимости в явном видеоуказании прав на сущности. Однако вы можете пометить части кода ключевыми словами Private и File-private для скрытия деталей реализации внутри модуля. Разработка фреймворков подразумевает создание интерфейса для общего доступа. Поэтому логично отмечать публичные сущности ключевыми словами Open и Public. Что же касается внутренней реализации фреймворка, уровень доступа к ней можно не задавать, а использовать принятый по умолчанию Internal. При написании юнит-тестов, им необходим доступ к тестируемому модулю. Для этого сущности модуля должны быть отмечены как Public или Open. Однако вы можете использовать атрибут Testable при импорте модуля. Тогда внутренняя реализация сущности будет также доступна для юнит-тестов. Однако сущности, помеченные как Private и File-private все еще не будут доступны.

- Open
- Public
- Internal
- File-private
- Private

4.6.2. Установка доступа

Уровень доступа задается при помощи ключевого слова перед указанием типа сущности.

```
1  public func publicFunction()
2  internal let InternalVariable = ""
3  fileprivate class FilePrivateClass {}
4  private func privateFunction() {}
```

4.6.3. Установка доступа

По умолчанию, как мы уже говорили ранее, уровень доступа устанавливается как Internal.

```
class InternalClass {          // internal класс
    var internalProperty = ""      // internal
                                   // свойство
    private var privateProperty = ""
                                   // private
                                   // свойство
    fileprivate func filePrivateMethod() {}
                                   // file-private
                                   // метод
}
```

4.6.4. Установка доступа

Если вы хотите задать уровень доступа для собственного типа, делайте это на этапе его описания. Уровень доступа, указанный для типа, будет установлен для всех сущностей, содержащихся в нем. Исключением является Public. Если вы задали модификатор доступа Public для типа, то автоматически для сущностей, содержащихся в нем, будет задан доступ Internal. Устанавливать уровень доступа Public необходимо вручную для каждой сущности.

```
public class PublicClass {         // public класс
    var internalProperty = ""          // internal
                                       // свойство
    private var privateProperty = ""
                                       // private
                                       // свойство
    public func publicMethod() {}
                                       // public метод
}
```

4.6.5. Функции

А теперь поговорим о кортежах. Уровень доступа для них не указывается явно. Он всегда автоматически определяется из членов, входящих в него. Уровень доступа устанавливается минимально возможный. Например, если в кортеж входят типы с доступом Public и Private, то для

кортежа будет установлен доступ Private. Аналогично и для функций. Уровень доступа автоматически устанавливается на основе параметров и возвращаемого типа. Уровень доступа должен быть указан явно, если он не соответствует общему контексту. В данном случае функция возвращает кортеж с параметрами доступа Public и Private. Для функции явно необходимо указать уровень доступа Private.

```
private func someFucntion() -> (PublicClass, PrivateClass) {
    return (PublicClass(), PrivateClass())
}
```

4.6.6. Перечисления

В перечислениях входящие в него кейсы получают тот же модификатор доступа, что и сам enumeration, к которому они принадлежат. Если используются raw или ассоциированные значения, то их уровень доступа должен быть не меньше, чем у перечисления, их содержащего.

```
fileprivate enum Elements {
    case fire
    case water
    case air
    case earth
}
```

4.6.7. Наследование

Вы можете наследоваться от любого класса, доступного в данном контексте. При этом уровень доступа для наследника не может быть выше, чем у супер-класса. Однако для методов и переменных, которые вы переопределяете, уровень доступа может быть изменен на желаемый. Как вы видите в примере, наследуемому классу разрешается получить доступ к File-private полям базового класса, если доступ осуществляется в рамках необходимого контекста. В данном случае оба класса объявлены в одном файле, и уровень доступа File-private позволяет обращаться к методу.

```
public class AClass {
    fileprivate func someMethod() {}
}
internal class BClass: AClass {
```

```
5       override internal func someMethod() {}
6   }
```

4.6.8. Константы, переменные для типа

Константы, переменные, поля и сабскрипты не могут иметь уровень доступа выше, чем их тип. Если для типа определен доступ Private, то и для инстанса должен быть определен тот же модификатор.

```
1   private class PrivateClass {
2       var privateVar: Int?
3       public var notPublic: Int?     // private доступ
4   }
5   private var privateClass = PrivateClass()
```

4.6.9. Сеттеры и геттеры

Сеттеры и геттеры получают тот же уровень доступа, что и свойства. Но вы можете, например, переопределить уровень доступа для сеттера. Для того, чтобы сделать свойства доступными только для чтения. В нашем примере создается структура со свойствами, доступными только для чтения. Вы можете прочитать значения данной переменной при обращении к инстансу, но изменить значения для Access-counter можно только из контекста структуры Logger.

```
1   struct Logger {
2       private(set) var accessCounter = 0
3       mutating public func readLog() {
4       accessCounter += 1
5       }
6   }
7   var logger = Logger()
8   // logger.accessCounter = 5
9   // Сеттер не доступен
10  logger.readLog()
11  logger.readLog()
12  print (logger.accessCounter)            // 2
```

4.6.10. Private геттер

Геттеру также можно установить приватный уровень доступа, но выполняется это немного иначе. Инициализатором может быть назначен уровень доступа меньший или равный доступу, определенному для типа. Исключением является Required инициализатор. Ему должен соответствовать тот же доступ, что и типу. То же правило касается и инициализаторов по умолчанию. У структур инициализатор будет иметь уровень доступа, соответствующий минимальному доступному для всех свойств, входящих в структуру. Если одно из свойств объявлено как Private, то и инициализатор структуры будет иметь доступ Private.

```
struct PrivateLogger {
    private private(set) var accessCounter = 0
    mutating public func readLog() {
        accessCounter += 1
}
}
var privateLogger = PrivateLogger()
privateLogger.readLog()
privateLogger.readLog()
// print (privateLogger.accessCounter)
// Нет доступа к геттеру
// error: 'accessCounter' is inaccessible
// due to 'private' protection level
```

4.6.11. Протоколы

Для протоколов модификатор доступа указывается при их объявлении. Требования в протоколе не могут быть объявлены с более низким уровнем, чем сам протокол. Это позволяет добиться того, что при поддержке протокола все требования будут доступны для типа, который его поддерживает. В отличие от других типов, при указании доступа Public, он будет распространен на все требования внутри протокола. Определение нового протокола на основе существующего не позволит установить более высокий доступ, чем у базового протокола. Вы не сможете определить Public-протокол, если он наследуется от Private-протокола.

```
fileprivate protocol SomeProtocol {
    var fileprivateVar: Int {get}
    func fileprivateFunc()
}
```

4.6.12. Extensions

Расширения для классов структур и перечислений имеют тот же уровень доступа, что и тип, для которого они создаются. Если для типа указан доступ Private, то и сущности, объявленные в расширении, имеют модификатор Private. Для расширения можно явно указать уровень доступа, если такая необходимость имеется. Но он не может быть выше, чем у расширяемого типа. Например, выполнение подобного кода приведет к ошибке компиляции.

```
private class PrivateClass {
    var privateVar: Int?
    public var notPublic: Int?     // private доступ
}
public extension PrivateClass {
 // extension of prive class
 // cannot be declared public
}
```

4.6.13. Доступ к Private из Extension

Если расширение для типа описано в том же файле, что и сам тип, то оно может получить доступ к сущностям, объявленным с модификатором Private. В качестве примера показана структура и extension для нее, внутри которого удается получить доступ к приватной переменной. В данной лекции мы рассмотрели возможности для ограничения доступа к участкам кода. А также увидели, как они реализованы для разных типов и сущностей.

```
struct SomeStruct {
    private var privateVariable = 0
}
extension SomeStruct {
    mutating func privateAccess() {
        privateVariable += 1
        print("Private variable value:
        \(privateVariable)")                // 1
    }
}
var someStruct = SomeStruct()
someStruct.privateAccess()
// Private variable value: 1
```

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