



## Урок 3

# Комплексные типы данных

Знакомство с комплексными данными. Структуры.  
Перечисления. Свойства и методы. Конструктор.

## [Перечисления](#)

[Простые перечисления](#)

[Перечисления со значением](#)

[Связанные значения](#)

## [Структуры](#)

[Свойства](#)

[Методы](#)

[Конструкторы](#)

[Уровень доступа](#)

## [Домашнее задание](#)

## [Дополнительные материалы](#)

## [Используемая литература](#)

# Перечисления

## Простые перечисления

Перечисления – это тип, объединяющий набор простых констант. Вы можете их использовать, чтобы код лучше читался и был менее подвержен ошибкам.

Рассмотрим простой пример. Создадим массив кортежей типа **(String, String, String)** для хранения ФИО.

```
let fio = [
    ("Иванов", "Иван", "Иванович"),
    ("Петров", "Петр", "Петрович"),
    ("Сидоров", "Сидор", "Сидорович"),
    ("Александров", "Александр", "Александрович")
]
```

Напишем функцию для вывода этого массива в консоль. Функция будет принимать два параметра: массив с данными и режим, в котором будет осуществляться вывод данных. Режимов должно быть три. В первом режиме выводим в консоль все данные, во втором – только фамилию и имя, в третьем – только фамилию. Режим представлен целым числом.

```
func printFio(_ fio: [(String, String, String)], mode: Int) {
    for i in fio {
        switch mode {
            case 1:
                print(i.0, i.1, i.2)
            case 2:
                print(i.0, i.1)
            case 3:
                print(i.0)
            default:
                fatalError("Поддерживается только два режим")
        }
    }
}

printFio(fio, mode: 1)
```

Все работает. Но у такого подхода есть ряд минусов:

- При вызове функции непонятно, что делает mode и что туда надо передавать.
- Приходится либо каждый раз читать определение, либо экспериментировать, чтобы понять, как работает функция.
- Ничто не мешает по ошибке передать в функцию число больше 3, ошибка вскрыется только во время выполнения программы.

Отличным решением в такой ситуации будет объявить новое перечисление и использовать его для выбора режима функции. Такой вариант намного удобнее:

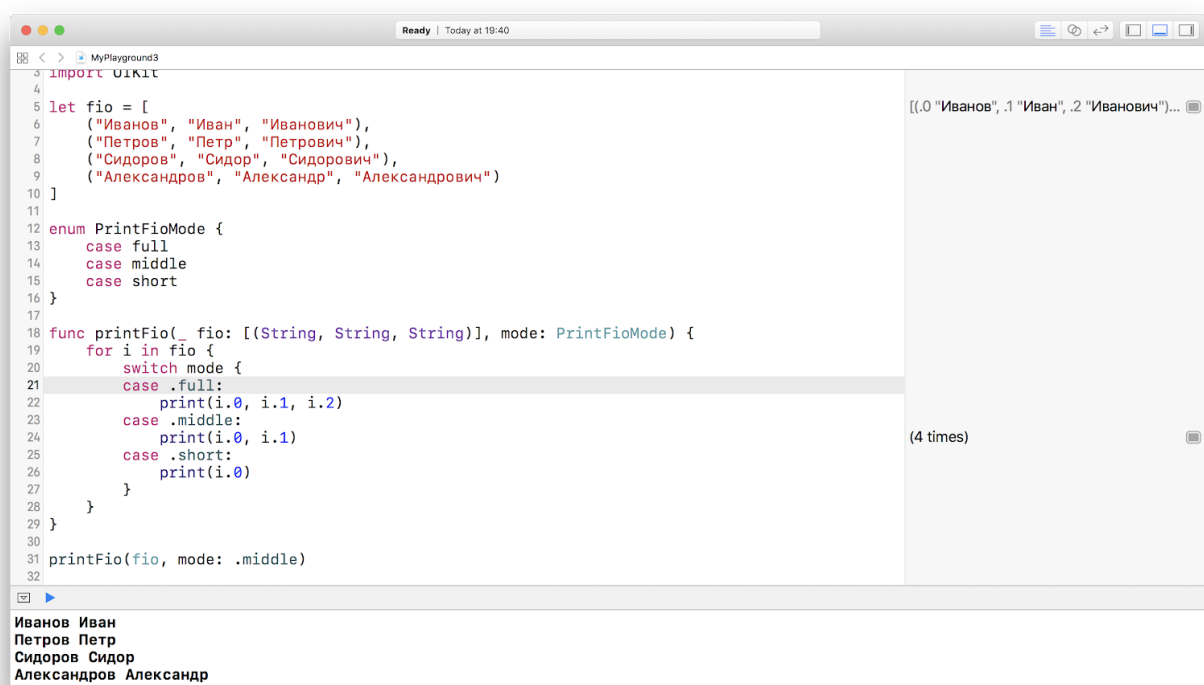
- Типы теперь имеют названия, понятные человеку.
- Автодополнение подскажет все возможные варианты переменной.

- Компилятор проверяет значение, переданное в функцию, и не позволит передать недопустимые данные.
- Так как перечисление содержит всего три варианта, мы перечислим их все в «switch» и можем не указывать секцию «default».

```
enum PrintFioMode {
    case full
    case middle
    case short
}

func printFio(_ fio: [(String, String, String)], mode: PrintFioMode) {
    for i in fio {
        switch mode {
            case .full:
                print(i.0, i.1, i.2)
            case .middle:
                print(i.0, i.1)
            case .short:
                print(i.0)
        }
    }
}

printFio(fio, mode: .middle)
```



## Перечисления со значением

По умолчанию перечисления не содержат никаких значений. Это означает, что вы можете только сравнивать одно значение с другим, чтобы выяснить, совпадают они или нет. Это удобно

использовать вместе с операторами ветвления и сравнения. Но иногда хочется ассоциировать перечисление с простым типом, чтобы использовать его в логике вашей программы.

Модифицируем пример, чтобы выводить в консоль название выбранного режима.

```
// указываем, что наше перечисление имеет тип String
enum PrintFioMode: String {
    case full = "Подробный режим:" // теперь можем присвоить строковое значение
    case middle = "Обычный режим"
    case short = "Сокращенный режим"
}

func printFio(_ fio: [(String, String, String)], mode: PrintFioMode) {
    // мы можем получить строковое значение через свойство rawValue
    print(mode.rawValue, ":\n")
    for i in fio {
        switch mode {
            case .full:
                print(i.0, i.1, i.2)
            case .middle:
                print(i.0, i.1)
            case .short:
                print(i.0)
        }
    }
}

printFio(fio, mode: .middle)
```

Такая возможность бывает крайне полезной. Вы можете перечислять какие-то параметры, даты или предметы, сделав ваш код более понятным и безопасным. При этом, если выбранному варианту соответствуют какие-то данные, вы можете получить и их.

Например, нам необходимо перечислить все планеты Солнечной системы – как правило, **enum** отлично подходит для этой задачи. Но нам также важно знать порядковый номер каждой планеты. Вы можете создать перечисление типа **Int**.

Если вашему перечислению назначен тип, вы можете использовать его для инициализации.

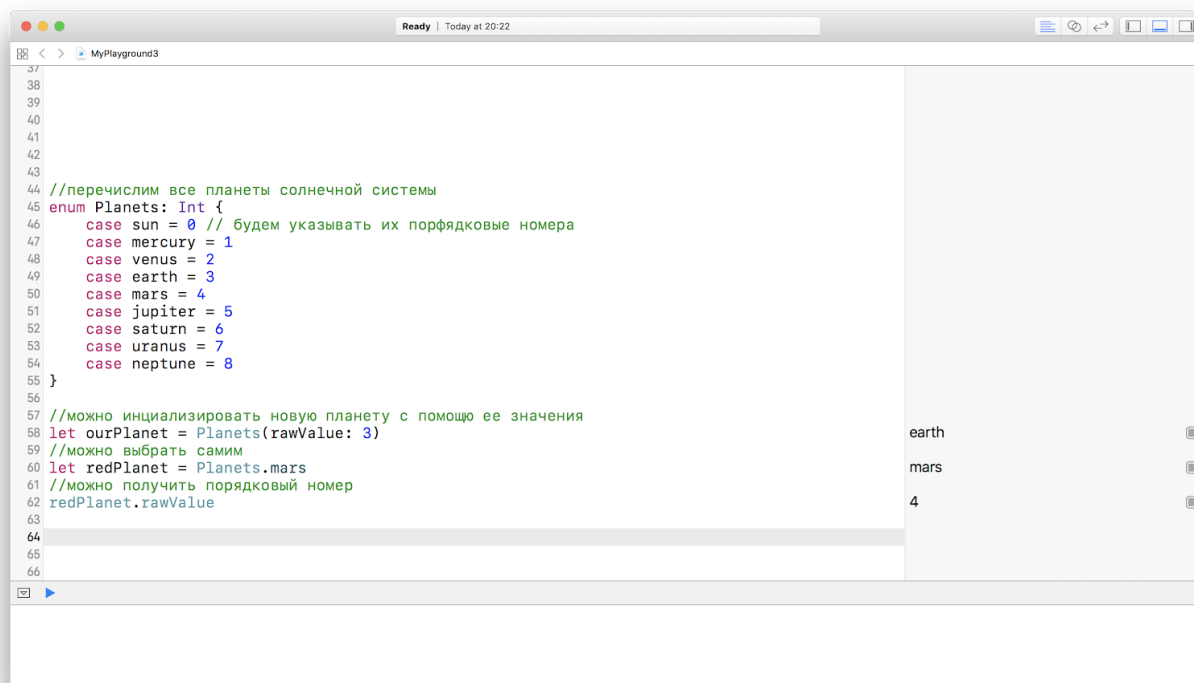
**Инициализация** – процесс создания нового значения какого-либо типа.

Если вы назначите вашему перечислению тип, но не укажете его, компилятор сделает это за вас.

```
enum Some: String { // мы указали тип
    case one // но не указали значение
}

Some.one.rawValue // оно автоматически стало равно имени
варианта
```

"one"



## Связанные значения

Мы уже разобрались, как создавать просто перечисления и ассоциировать их с некоторым значением, но самой мощной возможностью языка является связывать перечисление с некоторыми данными.

Помните, мы писали функцию, вычисляющую корни квадратного уравнения? Давайте перепишем ее, чтобы она возвращала результат работы. Правда, есть один нюанс. Функция может вернуть любое значение, но оно всегда одного типа и не может быть изменено. А наше уравнение имеет три варианта решения: с двумя корнями, с одним и вообще без корней.

Как же быть в такой ситуации? Можно, конечно, вернуть кортеж с двумя опциональными целыми числами, но это не очень красиво.

```

func solveQuadratic( a: Double, b: Double, c: Double) -> (Double?, Double?) {
    if(a != 0) {
        let discr: Double = pow(b, 2) - 4*a*c
        if(discr > 0 ) {
            let sqrOne = (-1) * b/(2 * a) + (sqrt(discr) / (2 * a))
            let sqrTwo = (-1) * b/(2 * a) - (sqrt(discr) / (2 * a))
            return (sqrOne, sqrTwo)
        }
        else if(discr == 0) {
            let sqrOne = (-1) * b/(2 * a)
            return (sqrOne, nil)
        } else {
            return (nil, nil)
        }
    }
    else {
        let sqrOne = (-1) * c / b
        return (sqrOne, nil)
    }
}

let result = solveQuadratic(a: 3, b: 9, c: -12)
if let sqr1 = result.0, let sqr2 = result.1 {
    print("два корня", sqr1, sqr2)
} else if let sqr1 = result.0 {
    print("один корень", sqr1)
} else{
    print("нет корней")
}

```

Согласитесь, выглядит не очень. А если представить, что вариантов не 3, а 5 или 15, проверка результата может стать намного сложнее его получения.

Перечисление со связанными параметрами решит проблему. Мы просто объявим новый тип перечислений с тремя возможными вариантами, каждый из которых может сохранять в себе данные.

```

enum SolveQuadraticResult{
    case twoSqr(one: Double, two: Double)
    case oneSqr(one: Double)
    case zeroSqr(error: String)
}

func solveQuadratic( a: Double, b: Double, c: Double) -> SolveQuadraticResult {
    if(a != 0) {
        let discr: Double = pow(b, 2) - 4*a*c
        if(discr > 0 ) {
            let sqrOne = (-1) * b/(2 * a) + (sqrt(discr) / (2 * a))
            let sqrTwo = (-1) * b/(2 * a) - (sqrt(discr) / (2 * a))
            return .twoSqr(one: sqrOne, two: sqrTwo)
        }
        else if(discr == 0) {
            let sqrOne = (-1) * b/(2 * a)
            return .oneSqr(one: sqrOne)
        } else {
            return .zeroSqr(error: "Корней нет")
        }
    }
    else {
        let sqrOne = (-1) * c / b
        return .oneSqr(one: sqrOne)
    }
}

let result = solveQuadratic(a: 3, b: 9, c: -12)
switch result {
case let .twoSqr(one, two):
    print("два корня", one, two)
case let .oneSqr(one):
    print("один корень", one)
case let .zeroSqr(error):
    print(error)
}

```

Новичкам обычно непросто начать использовать такие конструкции, но, поверьте, они сделают код намного лучше!

## Структуры

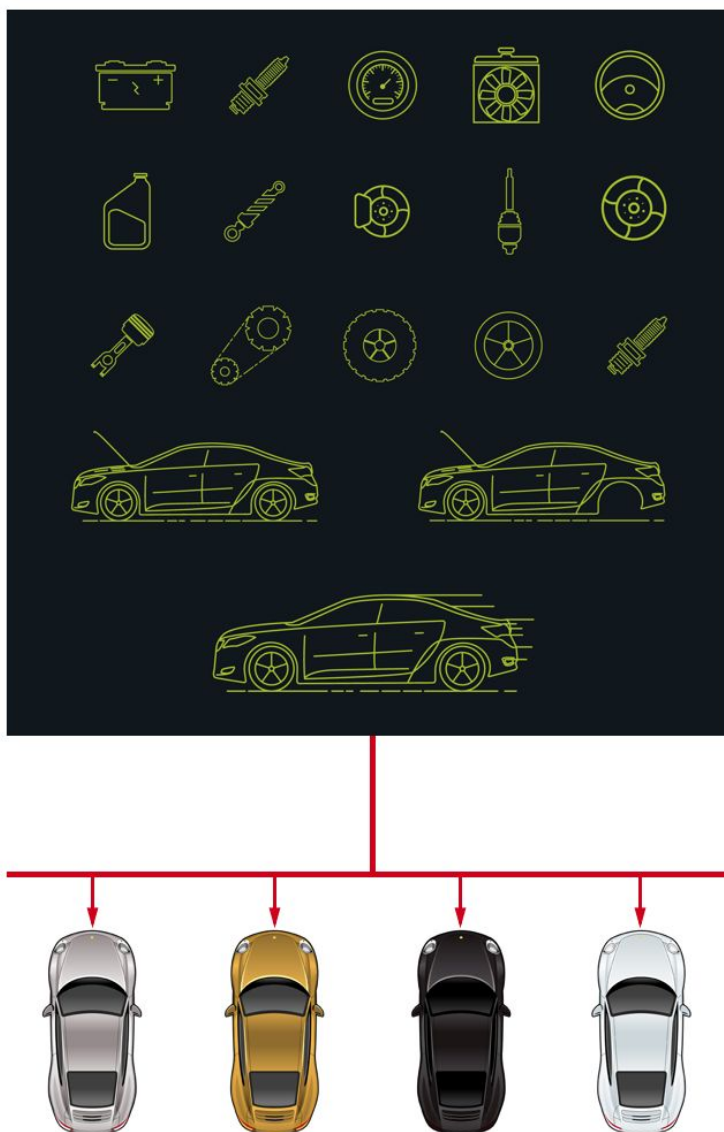
Мы с вами вплотную подошли к одной из самых сложных и фундаментальных областей – объектно-ориентированному программированию. О классах и объектах мы будем говорить на следующем уроке, но на самом деле в Swift структура мало чем отличается от класса.

До сих пор мы с вами использовали только простые типы: числа, строки, логические значения. Но при написании настоящих программ все эти типы не используются сами по себе, они являются частью объектов или структур. Любое iOS-приложение – это объект, который содержит в себе другие объекты и простые типы. Экраны в приложении – это объекты, даже кнопка – объект.



Объект – это экземпляр класса или структуры. Так что же такое структура?

**Структура** – абстрактный тип данных, который может содержать другие данные, в том числе другие структуры, а также набор методов по изменению этих данных.



Давайте представим, что нам надо описать в программе три машины. Каждая из машин может иметь различные цвет, магнитола, коробку передач, пробег и состояние дверей. Вот как мы сделали бы это, не зная о структурах:

```

enum HondaDoorState {
    case open, close
}
enum Transmission {
    case manual, auto
}
let car1Color = UIColor.white
let car1Mp3 = true
let car1Transmission = Transmission.auto
var car1Km = 0.0
var car1DoorState = HondaDoorState.open
let car2Color = UIColor.black
let car2Mp3 = false
let car2Transmission = Transmission.auto
var car2Km = 0.0
var car2DoorState = HondaDoorState.open
let car3Color = UIColor.red
let car3Mp3 = true
let car3Transmission = Transmission.manual
var car3Km = 12.0
var car3DoorState = HondaDoorState.open

```

Выглядит не очень хорошо. У нас всего три машины, а мы уже не можем с одного взгляда разобраться, что к чему. Представьте, что вам необходимо описать каждую деталь автомобиля и таких автомобилей создать тысячи. Но давайте перепишем этот код с использованием структуры:

```

enum HondaDoorState {
    case open, close
}
enum Transmission {
    case manual, auto
}
struct Honda {
    let color: UIColor
    let mp3: Bool
    let transmission: Transmission
    var km: Double
    var doorState: HondaDoorState
}
let car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
let car2 = Honda(color: .black, mp3: true, transmission: .manual, km: 120.0,
doorState: .close)
let car3 = Honda(color: .red, mp3: false, transmission: .manual, km: 0.0,
doorState: .open)
let car4 = Honda(color: .green, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)

```

Что произошло? Мы объявили структуру – другими словами, добавили в нашу программу новый тип данных «Honda». Теперь можно создавать переменные этого типа, массивы машин, передавать их в функции.

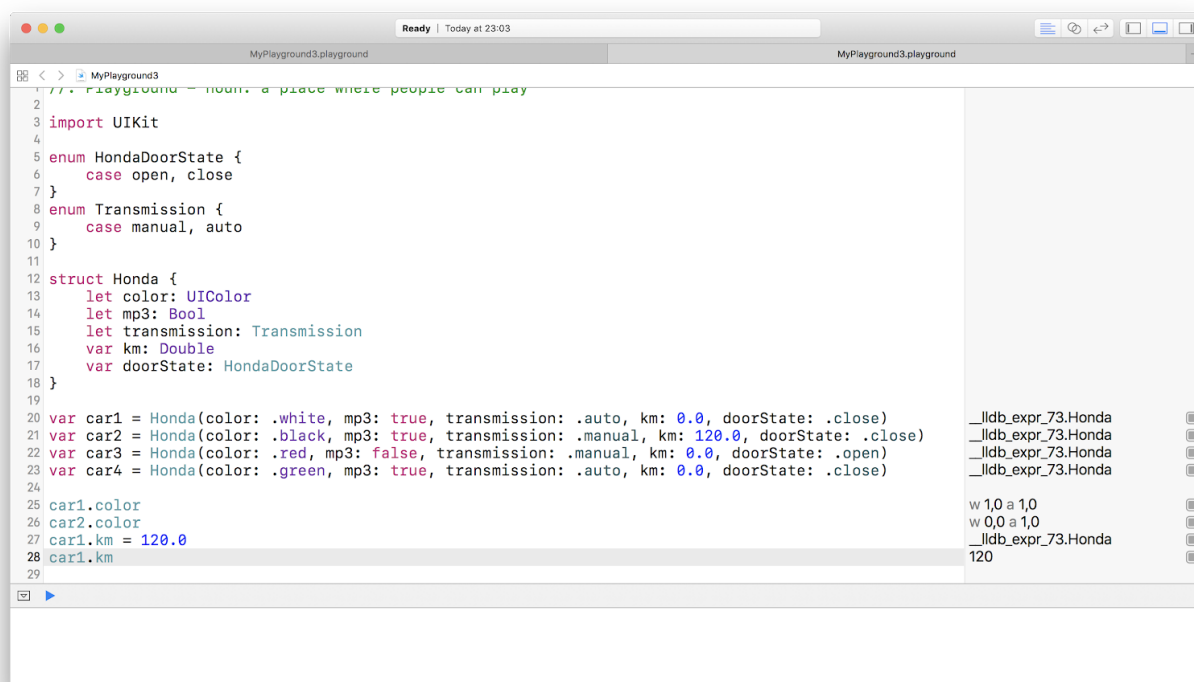
Возвращаясь к картинке с машинами, можно сказать, что структура – это чертеж нового типа данных. Он содержит полное описание: что будет содержать тип и какие действия может выполнять. Как только вы инициализируете экземпляр этой структуры, вы получите объект, настоящую машину, сделанную по чертежу.

Вы можете иметь только одну структуру одного типа, но создавать бесконечно много объектов этого типа. Другими словами, если вам необходимо сделать две одинаковых машины просто с разными значениями свойств, вам достаточно одной структуры типа «Honda».

## Свойства

Свойства похожи на переменные. Но в отличие от переменных, они не существуют сами по себе, а являются частью структуры. Любая структура может хранить сколько угодно любых свойств.

Например, у структуры «Honda» мы объявили 5 свойств: color, mp3, transmission, km и doorState. А когда мы инициализировали машины, мы указали значения для них. К любым свойствам можно получить доступ, но переустановить значение можно, только если свойство и сам объект не объявлены как константа.



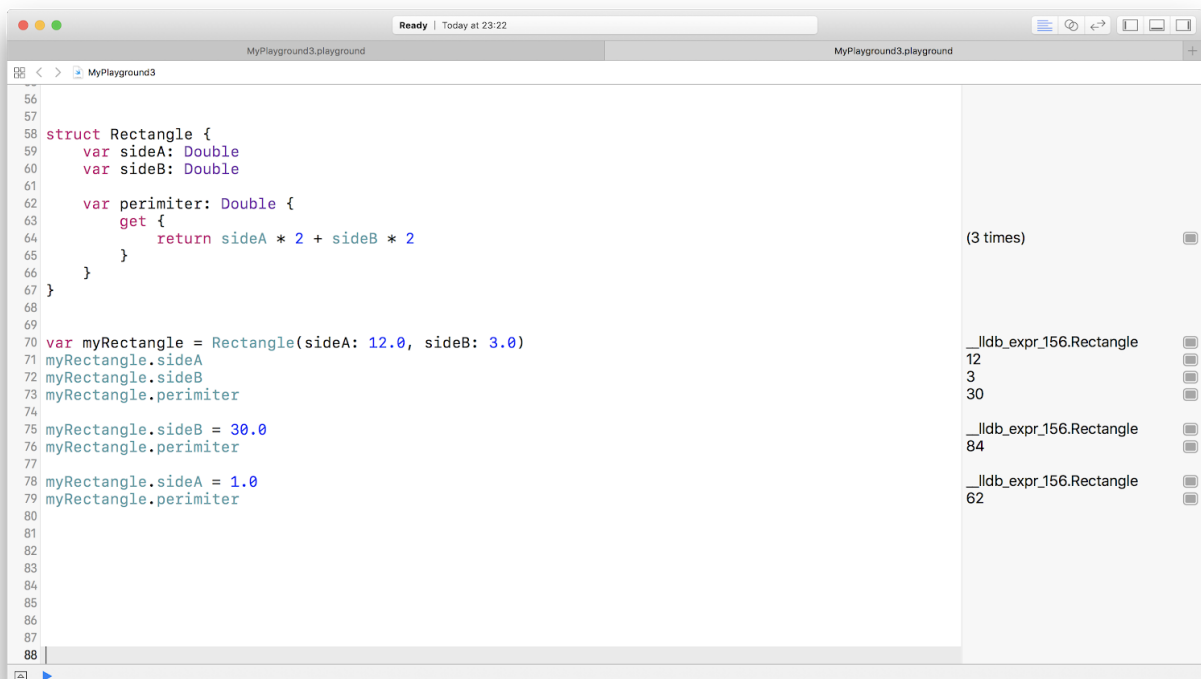
Хоть свойства и похожи на переменные, но у них есть несколько «козырей» в кармане. Это вычисляемые свойства и наблюдатели изменений.

Вычисляемое свойство выглядит как обычное, но ничего не хранит. При чтении оно вычисляет значение, а при установке значения – обрабатывает его. Яркий пример – диаметр круга.

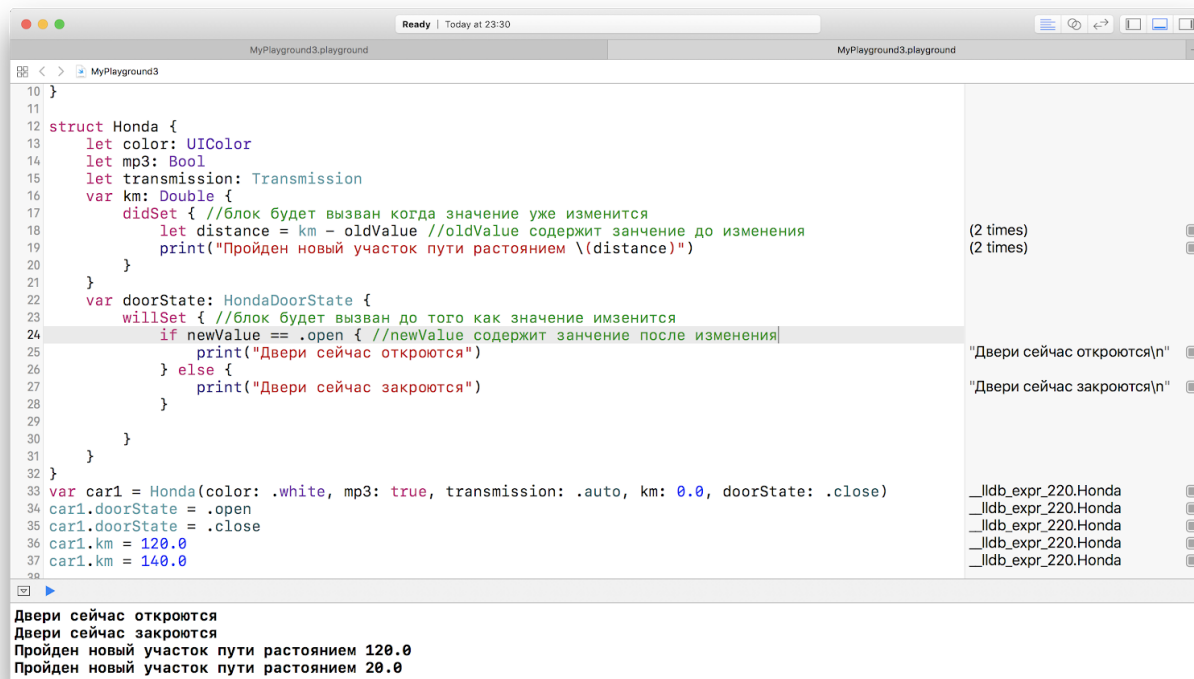


Мы объявили новую структуру «Circle» со свойствами «радиус» и «диаметр». Но так как эти свойства связаны, нет смысла хранить их оба. Мы храним только радиус, а диаметр вычисляем. Также при изменении значения диаметра мы на самом деле меняем радиус.

Блок «set» можно опустить — тогда вычисляемое значение будет доступно только для чтения, но не для записи. Например, у прямоугольника можно вычислить периметр на основе сторон, но нельзя вычислить стороны на основе периметра.



**Наблюдатели свойств** — это блоки кода, которые вызываются, когда свойство меняет значение. Это может быть полезно, чтобы следить за объектом. Давайте добавим наблюдение за нашим автомобилем.

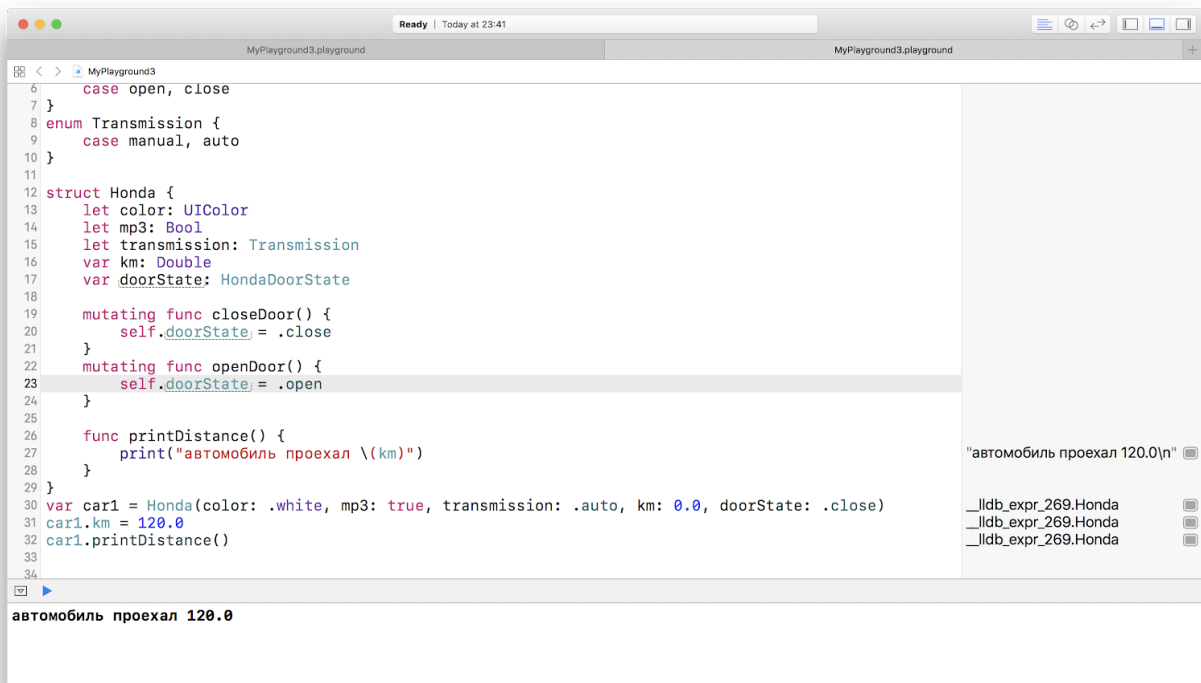


Теперь мы узнаем, когда у него открываются/закрываются двери или меняется километраж. Важно понимать, что «willSet» вызывается до того, как новое значение будет присвоено переменной, а «didSet» — уже после этого.

## Методы

Методы похожи на функции, пожалуй, даже больше, чем свойства на переменные. Но они тоже являются частью структуры, и им доступны все свойства структуры.

Метод может выполнять задачи, не связанные со свойствами, или же целиком опираться на свой объект. Метод даже может менять свойства своего объекта, но для этого его нужно пометить ключевым словом «mutating».



Мы добавили автомобилю три метода: метод открытия дверей, метод закрытия и метод вывода в консоль информации о пробеге.

Внутри метода доступна переменная «self». Это ссылка на объект структуры изнутри. В большинстве случаев его можно опустить, компилятор и так поймет, что вы обращаетесь к свойствам текущего объекта. Это как сказать: «Я вчера потянул свою руку» или «Я вчера потянул руку» – с большой долей вероятности вы потянули свою, а не чужую конечность.

## Конструкторы

**Конструкторы** – это особые методы, которые позволяют создать экземпляр структуры. В этом методе вы обязаны присвоить значение всем переменным, если это не было сделано ранее. Конструктор может содержать некую логику. Но основное его назначение – создать экземпляр и настроить его начальное состояние.

```
struct Rectangle {
    var sideA: Double
    var sideB: Double
    var perimeter: Double {
        get {
            return sideA * 2 + sideB * 2
        }
    }
    init() { // объявим простой конструктор, создающий прямоугольник со
        сторонами 5 и 10
        sideA = 5
        sideB = 10
    }
}
let rectangle = Rectangle()
```

Вы уже работали с конструктором, сами того не подозревая: вызывали его при создании машин из структуры «Honda». Вы можете удивиться, откуда взялся этот конструктор, ведь вы его не создавали. Дело в том, что всем структурам по умолчанию предоставляется конструктор, устанавливающий все свойства, не имеющие значения по умолчанию и не имеющие опциональный тип. Фактически вам был предоставлен конструктор, который вы могли бы написать сами.

```
// добавляем аргументы для каждого значения
init(color: UIColor, mp3: Bool, transmission: Transmission, km: Double,
doorState: HondaDoorState) {
    self.color = color // устанавливаем свойству color значение из
переменной color
    self.mp3 = mp3
    self.transmission = transmission
    self.km = km
    self.doorState = doorState
}
```

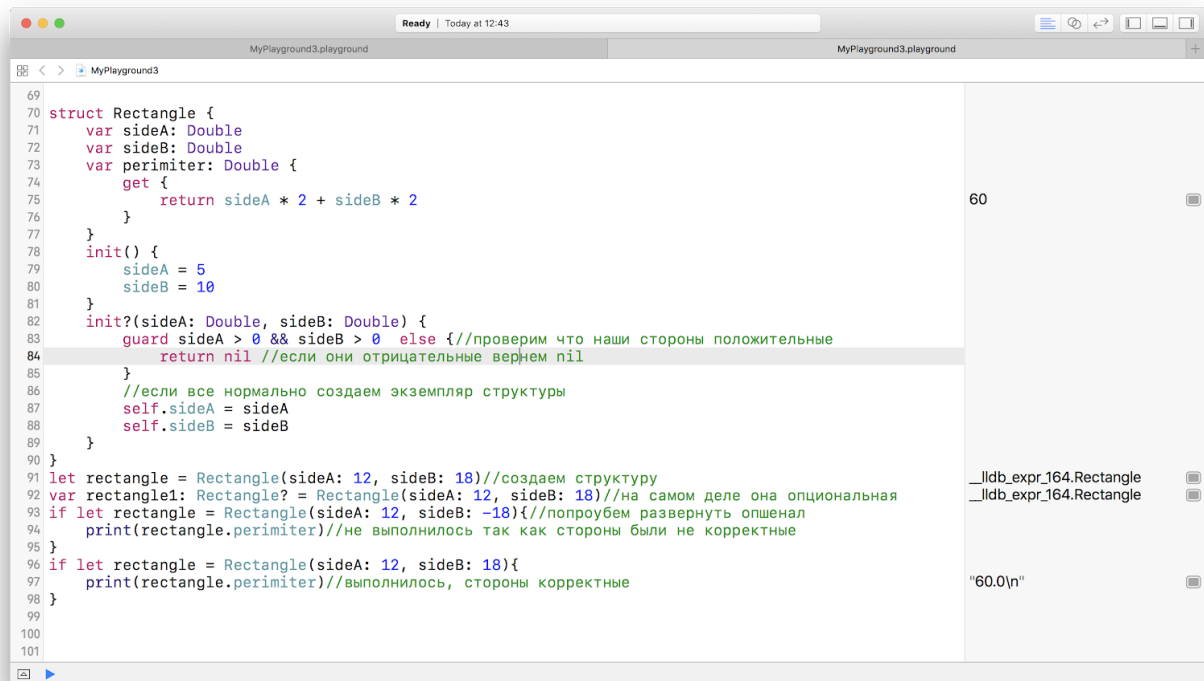
Ваш конструктор может принимать или не принимать параметры. Имена аргументов конструктора не обязаны совпадать с именами свойств. Вы можете даже принять больше аргументов, чем имеется свойств. Это не связанные сущности – связываете их вы сами в тот момент, когда присваиваете значение из переменной в свойство.

Важно понимать: как только вы добавите в структуру инициализатор, вы лишитесь инициализатора по умолчанию. Посмотрите еще раз на пример с прямоугольником. Мы объявили конструктор без аргументов, после этого стандартный конструктор с аргументами исчез. Мы лишились возможности задавать начальные значения сторон. Чтобы решить эту проблему, мы можем создать еще один конструктор. Таким образом, одна структура может иметь несколько инициализаторов.

```
struct Rectangle {
    var sideA: Double
    var sideB: Double
    var perimeter: Double {
        get {
            return sideA * 2 + sideB * 2
        }
    }
    init() {
        sideA = 5
        sideB = 10
    }
    init(sideA: Double, sideB: Double) {
        self.sideA = sideA
        self.sideB = sideB
    }
}
let rectangle = Rectangle()
let rectangle2 = Rectangle(sideA: 12, sideB: 23)
```

Конструктор может иметь опциональный тип. Это означает, что он может и не создать экземпляра структуры. Вернемся к прямоугольнику. Его конструктор с размерами сторон принимает два числа. Но

что, если мы переведем туда отрицательные числа? Прямоугольник инициализируется, но он будет математической ошибкой. Давайте предотвратим эту ситуацию, добавив проверку значений.



```
69 struct Rectangle {
70     var sideA: Double
71     var sideB: Double
72     var perimeter: Double {
73         get {
74             return sideA * 2 + sideB * 2
75         }
76     }
77 }
78 init() {
79     sideA = 5
80     sideB = 10
81 }
82 init?(sideA: Double, sideB: Double) {
83     guard sideA > 0 && sideB > 0 else { //проверим что наши стороны положительные
84         return nil //если они отрицательные вернем nil
85     }
86     //если все нормально создаем экземпляр структуры
87     self.sideA = sideA
88     self.sideB = sideB
89 }
90 }
91 let rectangle = Rectangle(sideA: 12, sideB: 18) //создаем структуру
92 var rectangle1: Rectangle? = Rectangle(sideA: 12, sideB: 18) //на самом деле она опциональная
93 if let rectangle = Rectangle(sideA: 12, sideB: -18) { //попробуем развернуть опшенал
94     print(rectangle.perimeter) //не выполнилось так как стороны были не корректные
95 }
96 if let rectangle = Rectangle(sideA: 12, sideB: 18) {
97     print(rectangle.perimeter) //выполнилось, стороны корректные
98 }
99
100
101
```

Теперь, если вы укажете отрицательные длины сторон, экземпляр структуры не будет создан. Такой конструктор возвращает опциональный тип, и его необходимо разворачивать, прежде чем использовать.

## Уровень доступа

Для свойств, методов, да и самих структур можно указывать области видимости. Уровень доступа определяет, из какой части проекта вы можете взаимодействовать с теми или иными свойствами. Уровень доступа указывается перед определением свойства, метода или структуры.



```

fileprivate struct Rectangle {                                // эта структура доступна
теперь только внутри файла
    private var sideA: Double                                  // стороны доступны теперь
только внутри структуры
    private var sideB: Double
    var perimeter: Double {
        get {
            return sideA * 2 + sideB * 2
        }
    }
    init() {
        sideA = 5
        sideB = 10
    }
    init?(sideA: Double, sideB: Double) {                     // проверим, что стороны
положительные
        guard sideA > 0 && sideB > 0 else {                    // если они отрицательные, вернем
nil
            return nil
        }
                                                                    // если все нормально, создаем
экземпляр структуры
        self.sideA = sideA
        self.sideB = sideB
    }
}

```

В данном примере вы не сможете воспользоваться структурой прямоугольника за пределами файла, в котором он описан. Компилятор просто не будет видеть ее, а попытки обратиться к ней будут вызывать ошибку. То же касается свойств «sideA» и «sideB». Вы можете работать с ними внутри структуры, но за ее пределами вы их уже не увидите и компилятор будет выдавать ошибку при попытке обратиться к ним. Все, что помечено как «private», не будет доступно даже у наследников.

## Домашнее задание

1. Описать несколько структур – любой легковой автомобиль и любой грузовик.
2. Структуры должны содержать марку авто, год выпуска, объем багажника/кузова, запущен ли двигатель, открыты ли окна, заполненный объем багажника.
3. Описать перечисление с возможными действиями с автомобилем: запустить/заглушить двигатель, открыть/закрыть окна, погрузить/выгрузить из кузова/багажника груз определенного объема.
4. Добавить в структуры метод с одним аргументом типа перечисления, который будет менять свойства структуры в зависимости от действия.
5. Инициализировать несколько экземпляров структур. Применить к ним различные действия.
6. Вывести значения свойств экземпляров в консоль.

# Дополнительные материалы

1. [Официальная документация.](#)
2. Брюс Эккель «Философия Java».

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift\\_Programming\\_Language/TheBasics.html#//apple\\_ref/doc/uid/TP40014097-CH5-ID309](https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309).