





Содержание

- ★ Перечисления (Enum) 
- ★ Структуры 
- ★ Классы 
- ★ Контроль доступа 

Перечисления (Enums)

- ❑ **Перечисления** - определяет общий тип для группы связанных значений и позволяет вам работать с этими значениями безопасным для типов способом в вашем коде
- ❑ **Перечисления** в Swift принимают многие функции, традиционно поддерживаемые только классами, такие как *вычисляемые свойства для предоставления дополнительной информации о текущем значении перечисления* и *методы экземпляра для обеспечения функциональности*, связанной со значениями, которые представляет перечисление.
- ❑ **Перечисления** также могут *определять инициализаторы для предоставления начального значения case*; могут быть *расширены для расширения их функциональности за пределы их первоначальной реализации*; и может соответствовать протоколам для обеспечения стандартной функциональности

Синтаксис перечисления

- ❑ Вы создаете **перечисления** с помощью ключевого слова **enum** и заключаете определение (тело enum) в пару фигурных скобок
- ❑ Значения, определенные в **перечислении** (например, north, south, east и west), являются его **случаями (case)** перечисления
- ❑ Вы используете ключевое слово **case**, чтобы ввести новые **случаи перечисления**

```
1  enum CompassPoint {  
2      case north  
3      case south  
4      case east  
5      case west  
6  }
```

Синтаксис перечисления

В одной строке может отображаться несколько случаев, разделенных запятыми

```
1  enum Planet {  
2      case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune  
3  }
```

Синтаксис перечисления

- ❑ Каждое определение (создание и присвоение имени) перечисления определяет новый тип.
- ❑ Как и другие типы в Swift, их имена (например, `CompassPoint` и `Planet`) начинаются с заглавной буквы.
- ❑ Дайте типам перечисления имена в единственном, а не во множественном числе, чтобы они читались как само собой разумеющиеся

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

Тип `directionToHead` выводится, когда он инициализируется одним из возможных значений `CompassPoint`. Как только `directionToHead` объявлен как `CompassPoint`, вы можете установить для него другое значение `CompassPoint`, используя более короткий синтаксис точки

```
directionToHead = .east
```

Сопоставление значений перечисления с оператором Switch

Вы можете сопоставить отдельные значения **перечисления** с оператором **switch**:

```
1 directionToHead = .south
2 switch directionToHead {
3     case .north:
4         print("Lots of planets have a north")
5     case .south:
6         print("Watch out for penguins")
7     case .east:
8         print("Where the sun rises")
9     case .west:
10        print("Where the skies are blue")
11 }
12 // Prints "Watch out for penguins"
```

Сопоставление значений перечисления с оператором Switch

Когда нецелесообразно предоставлять случай для каждого случая перечисления, вы можете указать случай по умолчанию (**default**), чтобы охватить любые случаи, которые не рассматриваются явно

```
1  let somePlanet = Planet.earth
2  switch somePlanet {
3    case .earth:
4      print("Mostly harmless")
5    default:
6      print("Not a safe place for humans")
7  }
8  // Prints "Mostly harmless"
```

Итерация по случаям перечисления

- ❑ Для некоторых перечислений полезно иметь коллекцию всех случаев этого перечисления
- ❑ Вы включаете это, написав : `CaseIterable` после имени перечисления
- ❑ Swift предоставляет коллекцию всех случаев как свойство `allCases` типа перечисления

```
1 enum Beverage: CaseIterable {  
2     case coffee, tea, juice  
3 }  
4 let numberOfChoices = Beverage.allCases.count  
5 print("\(numberOfChoices) beverages available")  
6 // Prints "3 beverages available"
```

```
1 for beverage in Beverage.allCases {  
2     print(beverage)  
3 }  
4 // coffee  
5 // tea  
6 // juice
```


Связанные значения

- ❑ Примеры в предыдущем разделе показывают, как **случаи перечисления** являются определенными (и типизированными) значениями сами по себе
- ❑ Вы можете установить константу или переменную для **Planet.earth** и проверить это значение позже. Однако иногда полезно иметь возможность хранить значения других типов вместе с этими значениями **case**
- ❑ Эта дополнительная информация называется **ассоциированным значением** и меняется каждый раз, когда вы используете этот регистр в качестве значения в своем коде
- ❑ Вы можете определить **перечисления** Swift для *хранения связанных значений любого заданного типа, и типы значений могут быть разными для каждого случая перечисления*, если это необходимо

Связанные значения

1. Например, предположим, что системе отслеживания запасов необходимо отслеживать товары по двум разным типам штрих-кодов. Некоторые продукты маркируются одномерными штрих-кодами в формате UPC, в котором используются числа от 0 до 9. Каждый штрих-код имеет цифру системы счисления, за которой следуют пять цифр кода производителя и пять цифр кода продукта. За ними следует контрольная цифра, чтобы убедиться, что код был отсканирован правильно



2. Другие продукты маркируются двумерными штрих-кодами в формате QR-кода, который может использовать любой символ ISO 8859-1 и может кодировать строку длиной до 2953 символов



Связанные значения

- ❑ Для системы отслеживания запасов удобно хранить штрих-коды UPC в виде кортежа из четырех целых чисел, а штрих-коды QR-кода — в виде строки любой длины
- ❑ В Swift перечисление для определения штрих-кодов продуктов любого типа может выглядеть так:

```
1 enum Barcode {  
2     case upc(Int, Int, Int, Int)  
3     case qrCode(String)  
4 }
```

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

```
productBarcode = .qrCode("ABCDEFGHJKLMNOP")
```

Связанные значения - Switch

Вы извлекаете каждое связанное значение как константу (с префиксом **let**) или переменную (с префиксом **var**) для использования в теле **switch case**

```
1 switch productBarcode {
2   case .upc(let numberSystem, let manufacturer, let product, let check):
3     print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
4   case .qrCode(let productCode):
5     print("QR code: \(productCode).")
6   }
7   // Prints "QR code: ABCDEFGHIJKLMNOP."
```

```
1 switch productBarcode {
2   case let .upc(numberSystem, manufacturer, product, check):
3     print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")
4   case let .qrCode(productCode):
5     print("QR code: \(productCode).")
6   }
7   // Prints "QR code: ABCDEFGHIJKLMNOP."
```

Raw (default) значения

- ❑ **Raw значениями** могут быть строки (**string**), символы (**char**) или любые типы целых чисел (**int**) или чисел с плавающей запятой (**float, double**)
- ❑ Каждое необработанное значение должно быть уникальным в пределах объявления перечисления
- ❑ Здесь **raw значения** для перечисления (**enum**) с именем *ASCIIControlCharacter* определены как имеющие тип **Character** и заданы для некоторых из наиболее распространенных управляющих символов ASCII

```
1  enum ASCIIControlCharacter: Character {  
2      case tab = "\t"  
3      case lineFeed = "\n"  
4      case carriageReturn = "\r"  
5  }
```

Неявно (implicit) присвоенные raw значения

- ❑ Когда вы работаете с **перечислениями (enum)**, в которых хранятся целые (**int**) или строковые (**string**) **raw значения**, вам не нужно явно назначать **raw значение** для каждого случая
- ❑ Если вы этого не сделаете, Swift автоматически присвоит вам значения
- ❑ Например, когда для raw значений используются целые числа (**int**), **неявное значение (implicit value)** для каждого случая на единицу больше, чем для предыдущего случая. Если в первом случае не задано значение, его значение равно **0**
- ❑ Приведенное ниже **перечисление (enum)** является уточнением предыдущего перечисления **Planet** с целочисленными raw значениями для представления порядка каждой планеты от солнца

```
1 enum Planet: Int {  
2     case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune  
3 }
```

Неявно (implicit) присвоенные raw значения

Когда строки (`string`) используются для **raw значений**, неявным значением для *каждого случая* (`case`) *является текст имени этого случая* (`case`)

В примере `CompassPoint.south` имеет неявное raw значение «`south`» и так далее.

```
1  enum CompassPoint: String {  
2      case north, south, east, west  
3  }
```

Доступ к raw значениям

Вы получаете доступ к **raw значению** случая (**case**) перечисления с помощью его свойства ***rawValue***

```
1  let earthsOrder = Planet.earth.rawValue
2  // earthsOrder is 3
3
4  let sunsetDirection = CompassPoint.west.rawValue
5  // sunsetDirection is "west"
```


Инициализация из raw значения

- ❑ Если вы определяете **перечисление (enum)** с типом **raw значения**, перечисление автоматически получает инициализатор, который принимает значение типа raw значения (в качестве параметра с именем **rawValue**) и возвращает либо case **перечисления**, либо **nil**
- ❑ Вы можете использовать этот инициализатор, чтобы попытаться создать новый экземпляр перечисления

```
1 let possiblePlanet = Planet(rawValue: 7)
2 // possiblePlanet is of type Planet? and equals Planet.uranus
```

Структуры и Классы

- ❑ **Структуры и классы** — это универсальные гибкие конструкции, которые становятся строительными блоками кода вашей программы
- ❑ Вы определяете **свойства(attributes)** и **методы(methods)** для добавления функциональности вашим структурам и классам, используя тот же синтаксис, который вы используете для определения **констант, переменных** и **функций**

Сравнение Структур и Классов

Структуры и классы в Swift имеют много общего. Оба могут:

- Определять свойства(attributes) для хранения значений
- Определять методы(methods) для обеспечения функциональности
- Определять инициализаторы(constructor), чтобы настроить их начальное состояние
- Быть расширенным(extensions), чтобы расширить их функциональность за пределы реализации по умолчанию
- Соответствовать протоколам(protocols) для обеспечения стандартной функциональности определенного типа

Сравнение Структур и Классов

У **классов** есть дополнительные возможности, которых нет у **структур**:

- **Наследование(inheritance)** позволяет одному классу наследовать характеристики другого.
- **Приведение типов(type casting)** позволяет проверять и интерпретировать тип экземпляра класса во время выполнения.
- **Деинициализаторы(deinit)** позволяют экземпляру класса освободить любые назначенные ему ресурсы.
- **Подсчет ссылок(reference counting)** допускает более одной ссылки на экземпляр класса.

Сравнение Структур и Классов

- ❑ Дополнительные возможности, поддерживаемые **классами**, *достигаются за счет увеличения сложности*
- ❑ Как правило, отдавайте предпочтение **структурам**, потому что они проще в использовании(безопаснее), и используйте классы, когда они уместны и необходимы
- ❑ На практике это означает, что большинство пользовательских типов данных, которые вы определяете, будут **структурами и перечислениями(enums)**

Синтаксис определения

- ❑ **Структуры и классы** имеют схожий синтаксис определения
- ❑ Вы вводите структуры с ключевым словом *struct* и классы с ключевым словом *class*
- ❑ Оба помещают свое полное определение в пару фигурных скобок `{ }`

```
1 struct SomeStructure {  
2     // structure definition goes here  
3 }  
4 class SomeClass {  
5     // class definition goes here  
6 }
```

```
1 struct Resolution {  
2     var width = 0  
3     var height = 0  
4 }  
5 class VideoMode {  
6     var resolution = Resolution()  
7     var interlaced = false  
8     var frameRate = 0.0  
9     var name: String?  
10 }
```

Объекты Класса и Структуры

Структуры и Классы используют синтаксис *инициализатора (конструктора)* для новых объектов

В простейшей форме синтаксиса инициализатора используется *имя типа класса или структуры, за которым следуют пустые круглые скобки*, например *Resolution()* или *VideoMode()*

Это создает новый *объект класса или структуры* со всеми *свойствами(attributes)*, инициализированными их значениями по умолчанию

```
1 let someResolution = Resolution()  
2 let someVideoMode = VideoMode()
```

Доступ к свойствам (attributes)

Вы можете получить доступ к свойствам объекта, используя **dot syntax**

В **dot syntax** вы пишете **имя свойства** сразу после **имени объекта**, разделяя его точкой (**.**), без пробелов

```
1 print("The width of someResolution is \someResolution.width")
2 // Prints "The width of someResolution is 0"
```

```
1 print("The width of someVideoMode is \someVideoMode.resolution.width")
2 // Prints "The width of someVideoMode is 0"
```


Доступ к свойствам (attributes)

Вы также можете использовать **dot syntax**, чтобы присвоить новое значение свойству переменной

```
1 someVideoMode.resolution.width = 1280
2 print("The width of someVideoMode is now \
   (someVideoMode.resolution.width)")
3 // Prints "The width of someVideoMode is now 1280"
```

Инициализаторы с атрибутами для Структур

- ❑ Все **структуры** имеют **автоматически сгенерированный инициализатор**(конструктор по умолчанию) для отдельных атрибутов, который можно использовать для инициализации свойств элементов (атрибутов) новых объектов структуры
- ❑ Начальные значения атрибутов нового объекта, могут быть переданы **конструктору** по имени атрибутов
- ❑ В отличие от **структур**, *объекты класса не получают инициализатор по умолчанию* (его нужно прописать вручную)

```
let vga = Resolution(width: 640, height: 480)
```

Инициализация (Конструктор)

Инициализация — это процесс подготовки объекта класса, структуры или перечисления к использованию

Этот процесс включает в себя *установку начального значения для каждого сохраненного атрибута в этом объекте* и выполнение любой другой настройки или *инициализации, которые требуются до того, как новый объект будет готов к использованию*

Вы реализуете этот **процесс инициализации**, определяя инициализаторы(конструкторы), похожие на специальные методы, которые можно вызывать для создания нового объекта определенного типа

Инициализация (Конструктор)

Инициализаторы вызываются для *создания нового объекта определенного типа*

В своей простейшей форме инициализатор похож на метод объекта без параметров, написанный с использованием ключевого слова **init**

```
1  init() {  
2      // perform some initialization here  
3  }
```

```
1  struct Fahrenheit {  
2      var temperature: Double  
3      init() {  
4          temperature = 32.0  
5      }  
6  }  
7  var f = Fahrenheit()  
8  print("The default temperature is \(f.temperature)° Fahrenheit")  
9  // Prints "The default temperature is 32.0° Fahrenheit"
```

Настройка инициализации

Вы можете **настроить процесс инициализации** с помощью **входных параметров** и **необязательных типов свойств (optional)** или путем **назначения постоянных свойств (constant)** во время инициализации

```
1 struct Celsius {  
2     var temperatureInCelsius: Double  
3     init(fromFahrenheit fahrenheit: Double) {  
4         temperatureInCelsius = (fahrenheit - 32.0) / 1.8  
5     }  
6     init(fromKelvin kelvin: Double) {  
7         temperatureInCelsius = kelvin - 273.15  
8     }  
9 }  
10 let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)  
11 // boilingPointOfWater.temperatureInCelsius is 100.0  
12 let freezingPointOfWater = Celsius(fromKelvin: 273.15)  
13 // freezingPointOfWater.temperatureInCelsius is 0.0
```

Настройка инициализации

Как и в случае параметров функций и методов, параметры инициализации могут иметь как имя параметра для использования в теле инициализатора, так и название аргумента для использования при вызове инициализатора

```
1 struct Color {  
2     let red, green, blue: Double  
3     init(red: Double, green: Double, blue: Double) {  
4         self.red = red  
5         self.green = green  
6         self.blue = blue  
7     }  
8     init(white: Double) {  
9         red = white  
10        green = white  
11        blue = white  
12    }  
13 }
```

```
1 let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)  
2 let halfGray = Color(white: 0.5)
```

Структуры и Enums являются типами Значений

- ❑ Тип Значения (Value type) — это тип, значение которого копируется, когда оно присваивается переменной или константе или когда оно передается функции
- ❑ Фактически, все основные типы в Swift — целые числа, числа с плавающей запятой, логические значения, строки, массивы и словари — являются типами значений и реализуются как структуры за кулисами
- ❑ Все структуры и перечисления являются типами значений в Swift
- ❑ Это означает, что любые созданные вами экземпляры структуры и перечисления — и любые типы значений, которые они имеют в качестве свойств — всегда копируются, когда они передаются в вашем коде

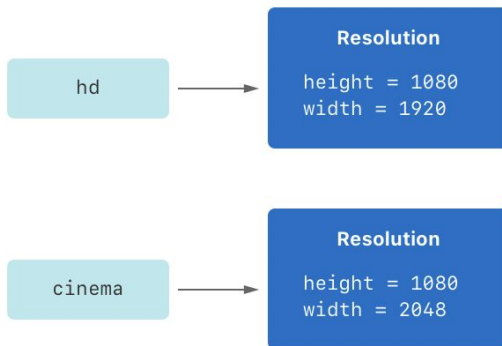
Структуры и Enums являются типами Значений

```
8 let hd = Resolution(width: 1920, height: 1080)
9 var cinema = hd
10
11 cinema.width = 2048
12
13 print("cinema is now \(cinema.width) pixel wide")
14 // Prints "cinema is now 2048 pixel wide"
15 print("hd is still \(hd.width) pixel wide")
16 // Prints "hd is still 1920 pixel wide"
```

Before



After



Структуры и Enums являются типами Значений

- ❑ Когда *RememberDirection* присваивается значение *currentDirection*, фактически устанавливается копия этого значения.
- ❑ Изменение значения *currentDirection* после этого не влияет на копию исходного значения, которое было сохранено в *RememberDirection*

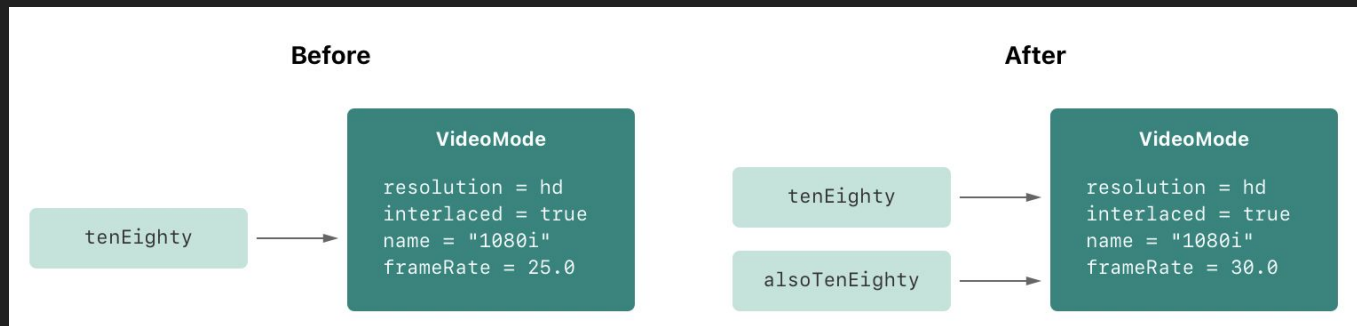
```
1  enum CompassPoint {
2      case north, south, east, west
3      mutating func turnNorth() {
4          self = .north
5      }
6  }
7  var currentDirection = CompassPoint.west
8  let rememberedDirection = currentDirection
9  currentDirection.turnNorth()
10
11  print("The current direction is \(currentDirection)")
12  print("The remembered direction is \(rememberedDirection)")
13  // Prints "The current direction is north"
14  // Prints "The remembered direction is west"
```

Классы являются Ссылочными типами (Reference Type)

- ❑ В отличие от **типов значений**(value type), **ссылочные типы**(reference type) не копируются, когда они присваиваются переменной или константе или когда они передаются функции
- ❑ Вместо копии используется ссылка на тот же существующий экземпляр

```
17 let tenEighty = VideoMode()
18 tenEighty.resolution = hd
19 tenEighty.interlaced = true
20 tenEighty.name = "1080i"
21 tenEighty.frameRate = 25.0
22
23 let alsoTenEighty = tenEighty
24 alsoTenEighty.frameRate = 30.0
25
26 print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
27 // Prints "The frameRate property of tenEighty is now 30.0"
```

Классы являются Ссылочными типами (Reference Type)



Атрибуты (свойства) класса/структуры

- ❑ **Свойства** связывают значения с определенным классом, структурой или перечислением
- ❑ **Хранимые свойства** хранят постоянные и переменные значения как часть объекта, тогда как **вычисляемые свойства** вычисляют (а не сохраняют) значение.
- ❑ **Вычисляемые свойства** предоставляются классами, структурами и перечислениями. **Хранимые свойства** предоставляются только классами и структурами.
- ❑ Кроме того, вы можете определить **наблюдателей свойств** для отслеживания изменений значения свойства, на которые вы можете реагировать с помощью настраиваемых действий
- ❑ **Наблюдатели свойств** могут быть добавлены к **хранимым свойствам**, которые вы определяете сами, а также к **свойствам, которые подкласс наследует от своего суперкласса**

Хранимые свойства

- ❑ В своей простейшей форме **хранимое свойство** — это константа или переменная, которая хранится как часть объекта определенного класса или структуры
- ❑ **Хранимые свойства** могут быть либо переменными (**var**), либо константой (**let**)
- ❑ Объекты *FixedLengthRange* имеют **хранимые свойства** как, переменную *firstValue*, и константу *length*. В примере *length* инициализируется при создании нового объекта и не может быть изменена после этого, поскольку является константой (**let**)

```
1 struct FixedLengthRange {  
2     var firstValue: Int  
3     let length: Int  
4 }  
5 var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)  
6 // the range represents integer values 0, 1, and 2  
7 rangeOfThreeItems.firstValue = 6  
8 // the range now represents integer values 6, 7, and 8
```

Ленивые хранимые свойства (lazy stored)

- ❑ Ленивое хранимое свойство — это свойство, начальное значение которого не вычисляется до первого использования
- ❑ Вы указываете ленивое хранимое свойство, записывая модификатор **lazy** перед его объявлением
- ❑ Вы всегда должны объявлять ленивое(lazy) свойство как переменную (с ключевым словом **var**), потому что его начальное значение может быть получено только после завершения инициализации экземпляра
- ❑ Ленивые свойства полезны, когда начальное значение свойства зависит от внешних факторов, значения которых неизвестны до завершения инициализации экземпляра
- ❑ Ленивые свойства также полезны, когда начальное значение свойства требует сложной или ресурсоемкой настройки, которую не следует выполнять до тех пор, пока она не потребуется

Ленивые хранимые свойства (lazy stored)

```
1  class DataImporter {
2      /*
3       DataImporter is a class to import data from an external file.
4       The class is assumed to take a nontrivial amount of time to initialize.
5       */
6      var filename = "data.txt"
7      // the DataImporter class would provide data importing functionality
      here
8  }
9
10 class DataManager {
11     lazy var importer = DataImporter()
12     var data: [String] = []
13     // the DataManager class would provide data management functionality
      here
14 }
15
16 let manager = DataManager()
17 manager.data.append("Some data")
18 manager.data.append("Some more data")
19 // the DataImporter instance for the importer property hasn't yet been
    created
```

Вычисляемые свойства

- ❑ В дополнение к сохраненным свойствам классы, структуры и перечисления могут определять **вычисляемые свойства**, которые фактически не хранят значение
- ❑ Вместо этого они предоставляют **getter** и необязательный **setter** для извлечения и установки нового значения свойства объекта

```
1 struct Point {
2     var x = 0.0, y = 0.0
3 }
4 struct Size {
5     var width = 0.0, height = 0.0
6 }
7 struct Rect {
8     var origin = Point()
9     var size = Size()
10    var center: Point {
11        get {
12            let centerX = origin.x + (size.width / 2)
13            let centerY = origin.y + (size.height / 2)
14            return Point(x: centerX, y: centerY)
15        }
16        set(newCenter) {
17            origin.x = newCenter.x - (size.width / 2)
18            origin.y = newCenter.y - (size.height / 2)
19        }
20    }
21 }
```

```
22 var square = Rect(origin: Point(x: 0.0, y: 0.0),
23                    size: Size(width: 10.0, height: 10.0))
24 let initialSquareCenter = square.center
25 // initialSquareCenter is at (5.0, 5.0)
26 square.center = Point(x: 15.0, y: 15.0)
27 print("square.origin is now at \(square.origin.x), \(square.origin.y)")
28 // Prints "square.origin is now at (10.0, 10.0)"
```


Короткая запись Setter

- ❑ Если **setter** **вычисляемого свойства** не определяет имя для нового устанавливаемого значения, используется имя по умолчанию *newValue*
- ❑ Вот альтернативная версия структуры Rect, использующая преимущества этой сокращенной записи

```
1  struct AlternativeRect {  
2      var origin = Point()  
3      var size = Size()  
4      var center: Point {  
5          get {  
6              let centerX = origin.x + (size.width / 2)  
7              let centerY = origin.y + (size.height / 2)  
8              return Point(x: centerX, y: centerY)  
9          }  
10         set {  
11             origin.x = newValue.x - (size.width / 2)  
12             origin.y = newValue.y - (size.height / 2)  
13         }  
14     }  
15 }
```

Read-only Вычисляемые свойства

- ❑ Вычисляемое свойство с `getter`, но без `setter` называется **вычисляемым свойством только для чтения (read-only)**
- ❑ Вычисляемое свойство, доступное только для чтения, всегда возвращает значение, и к нему можно получить доступ с помощью точечного синтаксиса, но нельзя установить другое значение
- ❑ Вы можете упростить объявление вычисляемого свойства только для чтения, удалив ключевое слово `get` и его фигурные скобки

```
1  struct Cuboid {  
2      var width = 0.0, height = 0.0, depth = 0.0  
3      var volume: Double {  
4          return width * height * depth  
5      }  
6  }  
7  let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)  
8  print("the volume of fourByFiveByTwo is \$(fourByFiveByTwo.volume)")  
9  // Prints "the volume of fourByFiveByTwo is 40.0"
```

Наблюдатели за свойствами (property observer)

- ❑ Наблюдатели за свойствами (property observer) наблюдают и реагируют на изменения в значении свойства
- ❑ Наблюдатели свойств вызываются каждый раз, когда устанавливается значение свойства, даже если новое значение совпадает с текущим значением свойства

Наблюдатели за свойствами (property observer)

Вы можете добавить наблюдателей свойств в следующих местах:

- Сохраненные свойства, которые вы определяете
- Сохраненные свойства, которые вы наследуете
- Вычисляемые свойства, которые вы наследуете

Наблюдатели за свойствами (property observer)

У вас есть возможность определить один или оба этих **наблюдателя для свойства**:

- *willSet* вызывается непосредственно перед сохранением значения
- *didSet* вызывается сразу после сохранения нового значения

Observer willSet

- ❑ Если вы реализуете наблюдатель **willSet**, ему передается новое значение свойства в качестве постоянного параметра
- ❑ Вы можете указать **имя для этого параметра** как часть реализации **willSet**
- ❑ Если вы не напишете имя параметра и круглые скобки в своей реализации, параметр будет доступен с именем параметра по умолчанию *newValue*

Observer didSet

- ❑ Точно так же, если вы реализуете наблюдатель `didSet`, ему *передается постоянный параметр, содержащий старое значение свойства*
- ❑ Вы можете назвать параметр или использовать имя параметра по умолчанию `oldValue`
- ❑ Если вы присваиваете значение свойству в его собственном обозревателе `didSet`, новое значение, которое вы присваиваете, заменяет только что установленное

Наблюдатели за свойствами (property observer)

```
1 class StepCounter {  
2     var totalSteps: Int = 0 {  
3         didSet(newTotalSteps) {  
4             print("About to set totalSteps to \$(newTotalSteps)")  
5         }  
6         didSet {  
7             if totalSteps > oldValue {  
8                 print("Added \$(totalSteps - oldValue) steps")  
9             }  
10        }  
11    }  
12 }
```

```
13 let stepCounter = StepCounter()  
14 stepCounter.totalSteps = 200  
15 // About to set totalSteps to 200  
16 // Added 200 steps  
17 stepCounter.totalSteps = 360  
18 // About to set totalSteps to 360  
19 // Added 160 steps  
20 stepCounter.totalSteps = 896  
21 // About to set totalSteps to 896  
22 // Added 536 steps
```


Методы

Методы — это функции, связанные с определенным типом

Классы, структуры и перечисления могут определять **методы** объекта, которые инкапсулируют определенные задачи и функции для работы с экземпляром данного типа

Методы объекта

- ❑ **Методы объекта** — это функции, принадлежащие объектам определенного класса, структуры или перечисления
- ❑ Они поддерживают функциональность этих объектов, либо предоставляя способы доступа и изменения свойств объекта, либо предоставляя функциональные возможности, связанные с назначением объекта
- ❑ **Методы объекта** *имеют точно такой же синтаксис, что и функции*
- ❑ Вы пишете **метод объекта** *внутри открывающей и закрывающей фигурных скобок типа, к которому он принадлежит* (класс, структура, перечисление)
- ❑ **Метод объекта** *имеет неявный доступ ко всем другим методам и свойствам объекта этого типа*
- ❑ **Метод объекта** можно вызывать *только для конкретного объекта* того типа, к которому он принадлежит. Его нельзя вызвать изолированно без существующего объекта

Методы объекта

- ❑ Вот пример, определяющий простой класс **Counter**, который можно использовать для подсчета количества повторений действия
- ❑ Вы вызываете методы экземпляра с тем же **точечным синтаксисом**, что и свойства:

```
1  class Counter {  
2      var count = 0  
3      func increment() {  
4          count += 1  
5      }  
6      func increment(by amount: Int) {  
7          count += amount  
8      }  
9      func reset() {  
10         count = 0  
11     }  
12 }
```

```
1  let counter = Counter()  
2  // the initial counter value is 0  
3  counter.increment()  
4  // the counter's value is now 1  
5  counter.increment(by: 5)  
6  // the counter's value is now 6  
7  counter.reset()  
8  // the counter's value is now 0
```

Атрибут self

- ❑ Каждый объект типа имеет неявное свойство *self*, которое точно эквивалентно самому объекту
- ❑ Вы используете свойство *self* для ссылки на текущий объект в его собственных методах объекта

```
1 func increment() {  
2     self.count += 1  
3 }
```

Атрибут self

На практике вам не нужно часто писать `self` в коде. Если вы явно не пишете `self`, Swift предполагает, что вы ссылаетесь на свойство или метод текущего объекта всякий раз, когда вы используете известное имя свойства или метода внутри метода

Основное исключение из этого правила возникает, когда **имя параметра для метода экземпляра** совпадает с **именем свойства этого экземпляра**. В этой ситуации имя параметра имеет приоритет, и возникает необходимость обращаться к свойству более квалифицированным образом. Вы используете свойство `self`, чтобы различать имя параметра и имя свойства

```
1 struct Point {
2     var x = 0.0, y = 0.0
3     func isToTheRightOf(x: Double) -> Bool {
4         return self.x > x
5     }
6 }
7 let somePoint = Point(x: 4.0, y: 5.0)
8 if somePoint.isToTheRightOf(x: 1.0) {
9     print("This point is to the right of the line where x == 1.0")
10 }
11 // Prints "This point is to the right of the line where x == 1.0"
```

Изменение типов значений из методов объекта (mutating)

- ❑ Структуры и перечисления являются типами значений. По умолчанию свойства типа значения не могут быть изменены из методов его экземпляра
- ❑ Однако, если вам нужно изменить свойства вашей структуры или перечисления в определенном методе, вы можете выбрать изменение поведения (**mutating**) для этого метода
- ❑ Затем метод может видоизменять (то есть изменять) свои свойства внутри метода, и любые изменения, которые он делает, записываются обратно в исходную структуру, когда метод завершается
- ❑ Метод также может назначить совершенно новый объект своему неявному свойству **self**, и этот новый объект заменит существующий, когда метод завершится

Изменение типов значений из методов объекта (mutating)

Вы можете включить это поведение, поместив ключевое слово **mutating** перед ключевым словом **func** для этого метода

```
1  struct Point {  
2      var x = 0.0, y = 0.0  
3      mutating func moveBy(x deltaX: Double, y deltaY: Double) {  
4          x += deltaX  
5          y += deltaY  
6      }  
7  }  
8  var somePoint = Point(x: 1.0, y: 1.0)  
9  somePoint.moveBy(x: 2.0, y: 3.0)  
10 print("The point is now at \(somePoint.x), \(somePoint.y)")  
11 // Prints "The point is now at (3.0, 4.0)"
```

Изменение типов значений из методов объекта (mutating)

Mutating методы могут присваивать совершенно новый объект неявному свойству **self**

```
1 struct Point {  
2     var x = 0.0, y = 0.0  
3     mutating func moveBy(x deltaX: Double, y deltaY: Double) {  
4         self = Point(x: x + deltaX, y: y + deltaY)  
5     }  
6 }
```


Методы Типа (Type Methods)

- ❑ Методы экземпляра, как описано выше, — это методы, которые вы вызываете для экземпляра определенного типа
- ❑ Вы также можете определить методы, которые вызываются для самого типа. Такие методы называются **методами типа**
- ❑ Вы указываете **методы типа**, написав ключевое слово **static** перед ключевым словом **func** метода
- ❑ Вместо этого классы могут использовать ключевое слово **class**, чтобы *позволить подклассам переопределять реализацию этого метода в суперклассе*

```
1 class SomeClass {
2     class func someTypeMethod() {
3         // type method implementation goes here
4     }
5 }
6 SomeClass.someTypeMethod()
```

```
3 struct SomeStruct {
4
5     static func greet() {
6         print("Hello World!")
7     }
8 }
9
10 SomeStruct.greet()
11 // Prints "Hello World!"
```

Контроль доступа

- ❑ **Контроль доступа** ограничивает доступ к частям вашего кода из других исходных файлов и модулей
- ❑ Эта функция позволяет вам *скрыть детали реализации вашего кода и указать предпочтительный интерфейс*, через который можно получить доступ к этому коду и использовать его
- ❑ Вы можете *назначать определенные уровни доступа* отдельным типам (классам, структурам и перечислениям), а также свойствам, методам, инициализаторам и индексам, принадлежащим этим типам

Уровни доступа

Swift предоставляет пять различных уровней доступа для сущностей в вашем коде. Эти уровни доступа относятся к исходному файлу, в котором определен объект, а также к модулю, которому принадлежит исходный файл

1. Открытый доступ (**Open**)
2. Публичный доступ (**Public**)
3. Внутренний доступ (**Internal**)
4. Файловый доступ (**Fileprivate**)
5. Приватный доступ (**Private**)

```
1 public class SomePublicClass {}
2 internal class SomeInternalClass {}
3 fileprivate class SomeFilePrivateClass {}
4 private class SomePrivateClass {}
5
6 public var somePublicVariable = 0
7 internal let someInternalConstant = 0
8 fileprivate func someFilePrivateFunction() {}
9 private func somePrivateFunction() {}
```

Открытый и публичный доступ (Open & Public)

- ❑ **Открытый доступ и публичный доступ** позволяют использовать объекты в любом исходном файле из их определяющего модуля, а также в исходном файле из другого модуля, который импортирует определяющий модуль. Обычно вы используете **открытый** или **общедоступный** доступ при указании общедоступного интерфейса для фреймворка
- ❑ **Открытый доступ** применяется *только к классам и объектам класса*, и он отличается от **публичного доступа** тем, что позволяет коду вне модуля создавать подклассы и переопределять их
- ❑ Пометка класса как **открытого** явно указывает на то, что вы учли влияние кода из других модулей, использующих этот класс в качестве суперкласса, и что вы соответствующим образом разработали код своего класса.

Внутренний доступ (internal)

- ❑ **Внутренний доступ (internal)** позволяет использовать объект в любом исходном файле из определяющего их модуля, но не в любом исходном файле за пределами этого модуля
- ❑ Обычно вы используете **внутренний доступ** при **определении внутренней структуры приложения или фреймворка**

Файловый доступ (fileprivate)

- ❑ **Файловый доступ (fileprivate)** ограничивает использование объекта его собственным определяющим исходным файлом
- ❑ Используйте **файловый доступ**, чтобы *скрыть детали реализации определенной части функциональности, когда эти детали используются во всем файле*

Приватный доступ (private)

- ❑ **Приватный доступ (private)** ограничивает использование объекта вложенным объявлением (*тело текущего класса/структуры*) и расширениями этого объявления, которые находятся в том же файле
- ❑ Используйте закрытый доступ, чтобы скрыть детали реализации определенной части функциональности, когда эти детали используются только в одном объявлении

Уровень доступа по умолчанию

Все сущности в вашем коде имеют **уровень доступа по умолчанию** — **внутренний(internal)**, если вы сами явно не укажете уровень доступа

В результате во многих случаях вам не нужно явно указывать уровень доступа в коде

```
1 class SomeInternalClass {}           // implicitly internal
2 let someInternalConstant = 0         // implicitly internal
```


Thanks for your attention!