Урок 6

Урок 6: Кортежи, классы, структуры, введение в ООП

Кортежи

Одной из особенностью Swift является то, что одной переменной или константе можно присвоить сразу несколько значений, но в отличии от массива эти значения могут быть разных типов. Такие объекты называются **Tuples**, что в переводе значит **Кортежи**.

> *Кортежи - это группировка нескольких значений в одно составное значение*

У каждого значения в составе кортежа может быть свой тип данных, который не зависит от других.

Синтаксис

```
(valueOne, ValueTwo, ValueThree,…)
```

Для создания кортежа, значения, которые нужно сгруппировать, помещаются через запятую в круглые скобки. Количество значений в кортеже не ограничено.

```
var currentWeather = (22, "Sunny")
```

В данном примере **currentWeather** это кортеж, который хранит в себе число **22** и строку **Sunny** и описывает текущую погоду. Этот кортеж группирует в себе значения двух различных типов **Int** и **String**. Хотя кортеж и может хранить в себе значения различных типов, это не значит, что сам он не имеет типа данных. В данном конкретном случае работает вывод типа, так как компилятор может вывести тип по значениям, которые вы присвоили, поэтому тип нашего кортежа **(Int, String)**. При этом вы можете явно указать тип и создать кортеж не присваивая ему значений:

```
var currentWeather: (Int, String)
currentWeatherVerTwo = (16, "Cloudy")
```

Порядок указания значений должен соответствовать порядку указания типов, так как если поменять местами значения или типы, то это уже будет кортеж другого типа.

Получение доступа к элементам

```
currentWeather.0
currentWeather.1
```

```
currentWeather.0 = 24
currentWeather.1 = "Rainy"
```

Мы можем присвоить значения элементов из нашего кортежа новым переменным или константам. Переменные, которым присваиваются значения, записываются в скобках через запятую после ключевого слова `var` или `let`, при этом количество переменных или констант должно соответствовать количе- ству элементов в кортеже.

```
let (temperature, atmos) = currentWeather
```

Значения элементов кортежа можно присваивать другим константам и переменным выборочно. Для этого необходимо заменить элементы, которые мы не хотим использовать, нижними подчеркиваниями.

```
let (temperature, _) = currentWeather
```

Именование элементов кортежа

Так же как и параметрам функций, элементам кортежа можно давать имена. Это бывает удобно, если необходимо обратиться к какому то элементу кортежа, номер индекса которого мы не знаем. Доступ к элементам кортежа по имени удобнее и нагляднее, чем доступ через индексы.

```
var weather = (temp: 21, atmos: "Fog", windSpeed: 6)
```

Задать имена для отдельных элементов значений не получится. Если присваиваете имя одному элементу, то придется присвоить имя и для всех остальных.

Где используются кортежи?

Кортежи полезны в любых местах где нужно группировать значения. Первый и самый простой способ использования кортежей — это массовое присвоение. Например выражение:

```
var numberOne = 1
var numberTwo = 2
var numberThree = 3
```

мы можем записать в одну строку:

```
var (x, y, z) = (1, 2, 3)
```

Еще одно применение кортежей - в циклах for...in. Возьмем для примера словарь **cityTemp**, в котором содержатся значения температуры в нескольких городах :

```
var cityTemp = ["Moscow": 6, "NewYork": 4, "London": 2, "Los
Angeles": 22]
```

Далее в цикле **for in** можно разложить пары словаря на отдельные значения и присвоить их переменным

```
for (cityName, temp) in cityTemp {
    print("Температура в городе \(cityName) сейчас равняется \
(temp) градуса Цельсия")
}
```

Кроме этого, кортежи можно использовать как возвращаемое значение для функций.

Кортежи не предназначены для создания сложных структур данных, их предназначение заключается в том, чтобы группировать однотипные или разнотипные значения и передавать их в другое место.

ООП в Swift

Объектно-ориентированное программирование - это фундаментальная парадигма программирования, которое берет своё начало с 1950-х годов и является проверенным способом построения сложных систем.

Общее понятие ООП

В ООП все строится из объектов!

Любая объектно-ориентированная программа — это набор взаимодействующих друг с другом компонентов, которые называются объектами. В таких программах ответственность за работу всего приложения распределена между всеми объектами.

Отдельный объект в программе является полностью самостоятельным и отвечает только сам за себя, а не за всю логику программы в целом. У каждого объекта своё назначение и своя роль. Объекты могут представлять из себя, как объекты реального мира, так и быть полностью абстрактными.

Хорошо спроектированные и продуманные программы позволяют работать над отдельными её частями, не охватывая логику всей программы целиком.

ООП так же позволяет упростить большие и сложные объекты, составляя их из более маленьких. Это в свою очередь позволяет упростить сложность всего приложения.

ООП позволяет создавать новые объекты на базе уже существующих. Это позволяет не переписывать каждый раз код заново, а дополнять уже существующий код новыми свойствами и методами. Этот механизм называется наследованием.

Классы и объекты

Класс — это фрагмент кода, у которого есть имя, класс(как и структура) определяет шаблон для построения объектов.

```
class Name {

}
```

После ключевого слова **class** идет имя класса, которое выступает типом для экземпляров класса. Имя класса пишется с прописной буквы. Далее между фигурными скобками располагаются элементы класса (свойства и методы).

На основе одного класса мы можем создавать любое количество экземпляров класса (объектов), отличающихся между собой уникальным именем.

Создание объекта в программировании называется инициализацией.

- Класс — это чертеж, по которому создается объект
- Объект — это переменная
- Из одного класса можно создать множество подобных объектов

Описание класса

Любой класс может состоять только из двух составляющих. Это методы и свойства класса. Методы класса отвечают за поведение объекта, а свойства за его состояние.

Свойства класса

Свойства класса это константы и переменные, которые описывают объект.

Виды методов

Метод — это функция, которая отвечает за выполнение одного определенного действия в классе и соответственно во всех его экземплярах. Методы не могут существовать вне классов, поэтому они и называются методы класса. Функции в отличии от методов, существуют вне классов и вообще к ним не привязаны.

Все методы можно разделить на две большие группы:

- метод экземпляра
- метод класса

Методы экземпляра

Для того что бы вызвать такой метод, нам необходимо сначала создать экземпляр класса (объект) и обратившись к экземпляру класса, вызвать метод.

Методы класса

В отличии от метода экземпляра, метод класса не возможно вызвать из экземпляра класса. Для того, что бы его вызвать, необходимо обратиться напрямую к классу, а не к его объекту.

Инициализаторы

Хорошим примером метода класса является инициализатор. Задача этого метода заключается в создании объекта класса или можно сказать в инициализации экземпляра класса. Соответственно если бы инициализатор был методом экземпляра, то мы бы просто не смогли его вызвать.

Getter / Setter

К методам класса также относятся геттеры и сеттеры. Эти методы устанавливают и получают значения переменных. У каждой закрытой переменной есть свой гетер и

сеттер. Гетеры — это публичные методы которые позволяют получать значения переменных, которые мы не видим. А сеттеры в свою очередь позволяют установить новые значения для переменных.

Структуры

Синтаксис

```
struct StructureName {
   var nameOfVarioable: DataType
   func someFunction {
        some code
   }
}
```

Структуры объявляются с помощью ключевого слова **stuct**, после которого следует имя создаваемой структуры. Так же как имена классов и перечислений, имена структур записываются в стиле камелкейс с прописной буквы. В теле структуры, между фигурными скобками, описываются свойства и методы.

Создавая структуру, вы определяете новый тип данных.

Описание структуры

Так же как и в классах, в структурах можно:

- Объявлять свойства для хранения значений
- Объявлять методы, что бы обеспечить функциональность
- Объявлять инициализаторы, что бы обеспечить первоначальное значение свойств

Но в отличии от классов структуры не могут наследовать характеристики от других структур. Так же у структур, в отличии от классов нет деинициализаторов. И пожалуй самым главным отличием структуры от класса является то, что структуры не являются ссылочным типом. В отличии от классов, при создании экземпляра структуры, данные копируются в новую переменную или константу, а не передаются в нее по ссылке!

Сравнение классов и структур

Изучая классы, мы поняли, что класс является ссылочным типом:

```
class MacBook {
    var name: String
    var year: Int
    var color: String

    init(name: String, year: Int, color: String) {
        self.name = name
        self.year = year
        self.color = color
    }

    func specs() {
        print("\(name) \(year) \(color)")
    }
}

let myMac = MacBook(name: "MacBook Pro", year: 2015, color:
"Silver")

myMac.specs()

let myWifesMac = myMac
myWifesMac.name = "MacBook Air"

myWifesMac.specs()
myMac.specs()
```

Из этого примера видно, что наши константы это всего лишь ссылки на экземпляр класса.

Когда вы работаете с экземплярами структуры, это можно сравнить с работой в Microsoft Office. В этом случае работая например с Excel вы создаете локальный файл у себя на компьютере. Поделившись этим файлом с другим пользователем, например по электронной почте, вы создаете копию своего

файла. Это значит, что все внесенные в новый файл изменения ни как не отобразятся на вашем оригинальном файле. Так же как и внесенные изменения в оригинальный файл, не найдут отражения в копии.

```
struct SmartPhone {
    var name: String
    var color: String
    var capacity: Int

    func specs() {
        print("\(name) \(color) \(capacity)Gb")
    }
}

let myPhone = SmartPhone() //Ошибка
```

Обратите внимание, что в отличии от класса, я не могу создать экземпляр структуры без заданных значений. При этом мне не обязательно реализовывать инициализатор для вызова значений по умолчанию. Структуры, так же как в перечисления, имеют встроенный инициализатор, который не требуется объявлять. Данный инициализатор принимает на входе значения всех свойств структуры, производит их инициализацию и создает экземпляр структуры

```
let myPhone = SmartPhone(name: "iPone 6", color: "Space Gray",
capacity: 64)
```

В результате создается новый экземпляр структуры, обладающий значениями всех свойств. Значения всех свойств должны быть определены. Пропустить какое либо из них не получится.

Для свойств можно задать значения по умолчанию непосредственно в теле структуры. В этом случае можно будет создавать экземпляр структуры без указания значения свойств. При этом если вы решите задать значение по умолчанию хотя бы для одного свойства, то вам придется задать значения и для всех остальных, т.к. Swift не позволяет определять значения по умолчанию лишь для некоторых свойств.

```
var myWifesPhone = myPhone
myWifesPhone.name = "iPhone SE"

myWifesPhone.specs()
myPhone.specs()
```

Как видите, в отличии от примера с классом, поменяв значения второго экземпляра мы не изменили значения первого. В отличии от классов структуры - это типы-значения, т.е. при создании экземпляра структуры этот экземпляр всегда будет уникальным и при передаче параметров от одного экземпляра структуры в другой происходит копирование. Так же обратите внимание на то, что создавая второй экземпляр структуры я присвоил ему ключевое слово **var**, в отличии от первого примера с классом, т.к. присвоив экземпляр кон-станте мы уже не сможем поменять значения свойств.

Методы в структурах

Создадим новую структуру **Rectangle**

```
struct Rectangle {
    var width: Int
    var height: Int

    func aria() -> Int {
        return width * height
    }
}

let myRectangle = Rectangle(width: 10, height: 5)
myRectangle.aria()
```

Итак наша структура позволяет рассчитать площадь прямоугольника. Давайте добавим метод, который будет позволять нам увеличивать стороны прямоугольника в нужное число раз.

```
struct Rectangle {
    var width: Int
```

```
    var height: Int

    func aria() -> Int {
        return width * height
    }

    func scale(w: Int, h: Int) {
        width *= w // Error
        height *= h // Error
    }
}
```

По умолчанию методы структур, кроме инициализаторов, не могут изменять значения свойств. Для того что бы обойти это ограничение, перед именем объявляемого метода необходимо указать модификатор *mutating*.

```
struct Rectangle {
    var width: Int
    var height: Int

    func aria() -> Int {
        return width * height
    }

    mutating func scale(w: Int, h: Int) {
        width *= w
        height *= h
    }
}

let myRectangle = Rectangle(width: 10, height: 5)
myRectangle.aria()
myRectangle.scale(w: 2, h: 3) // Error
```

Структура может изменять значения свойств только в том случае, если экземпляр структуры хранится в переменной

```
var myRectangle = Rectangle(width: 10, height: 5)
myRectangle.aria()
myRectangle.scale(w: 2, h: 3)
myRectangle.aria()
```