Функции. Методы

Для того чтобы объявить функцию нужно использовать ключевое слово func. В скобках после названия - список параметров. После стрелки указывается возвращаемый тип. Если он Void, то его можно опустить. Значение по умолчанию указываются после знака равно. При этом необязательно чтобы такой параметр был в конце. Возврат нескольких объектов из функции можно реализовать через кортежи.

```
// --- Функции. Методы. CODE SNIPPET #1 --- //

func usefulFunc(firstParam: Int = 0, secondParam: String = "", thirdParam: String) ->
(firstResult: Int, secondResult: String) {

    return (firstParam, secondParam)
}
```

Для вызова функции нужно указать аргументы для всех параметров, не имеющих значений по умолчанию. При этом порядок аргументов должен сохраняться. Для того, чтобы проигнорировать возвращаемое значение его нужно присвоить _. Либо использовать при объявлении функции атрибут @discardableResult.

```
// --- Функции. Методы. CODE SNIPPET #2 --- //

_ = usefulFunc(firstParam: 1, thirdParam: "Simple string")
```

Еще в Swift есть такое понятие как имя аргумента и имя параметра. В предыдущем примере они совпадали. Но можно их сделать разными или вообще убрать имя аргумента. Внутри функции мы используем имя параметра.

```
// --- Функции. Методы. CODE SNIPPET #3 --- //

func usefulFunc2(_ parameterWithoutArgumentName: Int, argumentName parameterName:
String) {
    print("Received \(parameterWithoutArgumentName) and \(parameterName)")
}
```

При вызове указываем имя аргумента. Или просто значение если имя аргумента не используется. Используя такой подход убедитесь, что код остается понятным и читаемым.

```
// --- Функции. Методы. CODE SNIPPET #4 --- //

usefulFunc2(20, argumentName: "String argument")
```

Также в Swift можно использовать функции с переменным числом параметров. Для этого после типа нужно поставить три точки. Внутри функции эти параметры будут доступны как массив. Очевидно, что в функции может быть только один такой параметр. При этом не обязательно, чтобы этот параметр был последним.

```
// --- Функции. Методы. CODE SNIPPET #4 --- //

func sum(_ numbers: Int...) -> Int {
    var result: Int = 0
    for number in numbers {
        result += number
    }

    return result
}

sum(1, 2, 3, 4, 5) // 15
```

Аналогично передаче параметра по ссылке в C++ в Swift мы тоже имеем возможность изменять аргументы внутри функции. Для этого нужно пометить его ключевым словом inout.

```
// --- Функции. Методы. CODE SNIPPET #4 --- //

func swapInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

При вызове аргументы передаются через амперсанд. Нельзя передавать таким образом константу или литерал. inout параметры не могут иметь значений по умолчанию или быть параметром с переменным числом аргументов.

```
// --- Функции. Методы. CODE SNIPPET #6 --- //

var first = 1
var second = 2

swapInts(&first, &second)

print("first \(first) second \(second)") // first 2 second 1
```

Функции в Swift являются функциями высшего порядка. Они имеют свой тип. И как и другие типы данных могут храниться в переменных и быть переданы в другие

функции. Несмотря на разное название обе функции имеют тип (Int, Int) -> Int. Т.е. принимают два параметра типа Int и возвращают Int.

// --- Функции. Методы. CODE SNIPPET #7 --- //

```
func summation(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplication(_ a: Int, _ b: Int) -> Int {
    return a * b
}

type(of: summation) // (Int, Int) -> Int
type(of: multiplication) // (Int, Int) -> Int
```

Переменная для хранения функции должна иметь такой же тип. Указывать его в данном случае не обязательно. Это сделано для наглядности. Для вызова функции из переменной мы пишем название переменной и в скобках необходимые параметры.

// --- Функции. Методы. CODE SNIPPET #8 --- //

```
var function: (Int, Int) -> Int = summation
function = multiplication

function(3, 4) // 12
```

Для того чтобы передать функцию нужно указать ее тип как и любой другой. Обратите внимание, что squareTransformator объявлена внутри другой функции. В предыдущих примерах функции были объявлены в глобальной области видимости. Поэтому их называют глобальными. squareTransformator является вложенной. Такие функции доступны только в своей области видимости. Но их все равно можно передавать во внешний скоуп.

// --- Функции. Методы. CODE SNIPPET #9 --- //

```
func provideTransformator() -> (Int) -> Int {
    func squareTransformator(transform value: Int) -> Int {
        return value * value
    }

    return squareTransformator
}
```

Для того, чтобы передать функцию нужно просто указать ее название без круглых скобок. Их мы используем в момент вызова. Обратите внимание, что имена аргументов уже не указываются при вызове переданной функции.

```
// --- Функции. Методы. CODE SNIPPET #10 --- //
func transform(_ value: Int, using transformator: (Int) -> Int) -> Int {
    return transformator(value)
}

let transformator = provideTransformator()
let result = transform(6, using: transformator) // 36
```

Методы - это функции ассоциированные с каким-либо типом. В Swift методы могут быть у классов, структур и перечислений (enum).

Синтаксис объявления метода аналогичен объявлению функции. Внутри метода имеется свойство self. Через него можно обратиться к самому объекту у которого вызван метод. Однако в большинстве случаев Swift сам понимает, что вы имеете в виду и self можно опустить. Он может понадобиться, например, если имя свойства у объекта и имя параметра совпадают.

```
// --- Функции. Методы. CODE SNIPPET #11 --- //

class Counter {
    private var count = 0
    func increment() {
        self.count += 1
    }

    func isCountGraterThan(_ count: Int) -> Bool {
        return self.count > count
    }
}
```

По умолчанию методы value type не могут модифицировать свой объект. Чтобы дать им такую возможность нужно добавить ключевое слово mutating перед объявлением метода. Более того, вы можете присвоить в self новый объект и он заменит собой предыдущий, в содержащей его переменной.

```
// --- Функции. Методы. CODE SNIPPET #12 --- //

struct Point {
    var x = 0.0
    var y = 0.0

    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
```

```
    mutating func reset() {
        self = Point()
    }
}

var point = Point(x: 3, y: 4)
point.moveBy(x: 4, y: -2)
point.reset()

print("X:\(point.x) Y:\(point.y)") // X:0.0 Y:0.0
```

Для добавления методов самому типу нужно использовать ключевое слово static. Однако в Swift есть небольшая особенность. Статические методы нельзя переопределять в сабклассе. Чтобы показать, что необходимо использовать динамическую диспетчеризацию нужно вместо static указать слово class.

Внутри классовых методов self указывает на сам тип. Это дает доступ к классовым свойствам и методам. Но как и с обычными методами для доступа self можно не указывать. Не обращайтесь из одного классового метода к другому через имя класса. Если вы переопределите такой метод в дочернем классе, то вызовется все равно родительский. Т.к. вы явно указали тип.

```
// --- Функции. Методы. CODE SNIPPET #13 --- //

class MyClass {
    class func generateText() -> String {
        return "MyClass"
    }

    class func printer1() {
        print("\(self.generateText())") // тут self не обязательно
    }
    class func printer2() {
        // Не вызывайте методы так если не уверены на 100%, что понимаете к чему это
приведет
        print("\(MyClass.generateText())")
    }
}

class MySubClass: MyClass {
    override class func generateText() -> String {
        return "MySubClass"
    }
}
```

```
MySubClass.printer1() // MySubClass
MySubClass.printer2() // MyClass
```

К методам типа можно обратиться и из экземпляра класса или структуры. Для этого лучше использовать функцию type(of:). Она возвращает тип переданного экземпляра. Обращаться к ним по имени класса не стОит из-за наследования.

```
// --- Функции. Методы. CODE SNIPPET #14 --- //

struct MyStruct {
    static var sharedValue = 0

    func increment() {
        type(of: self).sharedValue += 1
    }
}

let myStruct = MyStruct()

MyStruct.sharedValue // 0

myStruct.increment()

MyStruct.sharedValue // 1
```