Замыкания

Замыкания похожи на блоки в Objective-C или на лямбда функции из других языков программирования. Т.е. замыкания это просто блоки кода, которые можно передавать и использовать. Назвали их так потому, что они могут захватывать\завязываться на переменные и константы объявленные в их контексте.

Рассмотрим пример. Если бы внутренняя функция никак не обращалась к counter, то после выхода из внешней переменная была бы удалена из памяти. Однако в этом случае Swift сохраняет ее в памяти для того, чтобы увеличивать ее значение при вызове внутренней функции. Это и есть замыкание. При этом вам ничего не нужно дополнительно объявлять. Swift все управление памятью берет на себя.

```
// --- Замыкания CODE SNIPPET #1 --- //

func counterFunc() -> (Int) -> String {
    var counter = 0
    func innerFunc(i: Int) -> String {
        counter += i // counter захвачен в замыкании
        return "running total: \(counter)"
    }
    return innerFunc
}

let closureFromFunc = counterFunc()

print("\(closureFromFunc(3))") // running total: 3
print("\(closureFromFunc(3))") // running total: 6
print("\(closureFromFunc(3))") // running total: 9
```

Рассмотрим еще один пример замыкания. В этот раз оно будет анонимным. В Swift такие замыкания называются closure expression. Для его объявления в фигурных скобках указываются: параметры, перечисленные внутри круглых скобок, стрелка, тип возвращаемого значения и ключевое слово in. Так же как в обычной функции, но значения по умолчанию в этом случае указывать нельзя. Далее следует сам код замыкания. В нем мы можем обращаться к переданным параметрам по соответствующим именам. Также нам доступны переменные и константы объявленные во внешнем скоупе.

```
// --- Замыкания CODE SNIPPET #2 --- //

let multiplier = 3

let anonymousClusure = {
    (anotherItemInArray: Int) -> Int in
```

```
    return anotherItemInArray * multiplier
}

func transform(_ value: Int, using transformator: (Int) -> Int) -> Int {
    return transformator(value)
}

transform(6, using: anonymousClusure) // 18
```

Если вы смотрели предыдущую лекцию о функциях и методах, то наверняка вспомнили, что в ней мы объявляли точно такую же функцию transform(_: using:), но передавали в нее функцию. В чем тогда разница между closure expression и функциями? Ответ в том, что разницы нет. Функции это и есть замыкания.

Глобальные функции - это замыкания, которые ничего не захватывают. Вложенные функции - это замыкания, имеющие имя и захватывающие контекст в котором объявлены. Анонимные closure expression тоже захватывают контекст, но не имеют имени.

В Swift closure expressions очень активно используются. Поэтому для них было придумано много оптимизаций в синтаксисе, чтобы сделать его более коротким.

Для начала уберем ненужную переменную. Замыкание можно объявлять прямо при вызове функции. Ключевое слово in означает окончание объявления параметров и возвращаемого значения. Поэтому можно писать код замыкания в той же строке. Естественно не нужно это делать если в вашем замыкании больше одного выражения.

```
// --- Замыкания CODE SNIPPET #3 --- //

transform(6, using: { (anotherItemInArray: Int) -> Int in return anotherItemInArray * multiplier
})
```

Т.к. мы передаем замыкание в функцию Swift может сам вывести ее тип. Достаточно просто перечислить имена параметров.

```
transform(6, using: { anotherItemInArray in return anotherItemInArray * multiplier })
```

Если в вашем замыкании всего одно выражение, то ключевое слово return можно опустить. Результат его выполнения сам вернется из замыкания.

```
transform(6, using: { anotherItemInArray in anotherItemInArray * multiplier })
```

Также Swift неявно объявляет параметры в виде $0, $1, $2 и т.д. если вы не объявите их сами. Они соответствуют параметрам замыкания по порядку.

```
transform(6, using: { $0 * multiplier })
```

Часто замыкание передается в функцию в качестве последнего или единственного аргумента. В этом случае его можно вынести за скобки. А если оно единственный аргумент даже не указывать их вообще.

```
transform(6) { $0 * multiplier }
```

Это особенно удобно использовать если в замыкании много кода и в одну строку его не записать.

```
// --- Замыкания CODE SNIPPET #4 --- //

transform(6) {
    let temp = $0 * multiplier

    /*

    Много полезных вычислений

     */

    return temp
}
```

Несмотря на то, что Swift берет на себя управление памятью все же нужно понимать как работают замыкания, чтобы не допустить ошибки. Замыкание захватывает именно тот объект, который ему доступен при его создании. Например, если мы создадим еще одно замыкание, вызвав counterFunc(), то во время выполнения этой функции будет создана новая переменная counter. И новое замыкание захватит именно ее, а не ту, что была захвачена в прошлый раз.

```
// --- Замыкания CODE SNIPPET #5 --- //

let anotherCounter = counterFunc()
print("\(anotherCounter(3))") // running total: 3
print("\(closureFromFunc(3))") // running total: 12
```

При этом переменную оно захватывает по ссылке. Ее изменения внутри замыкания будут видны и снаружи. В качестве оптимизации может быть сделана копия, но только если Swift уверен, что внутри замыкания переменная не будет изменяться.

Однако у нас есть возможность изменить это поведение. Для этого используется список захвата. Он располагается перед списком параметров замыкания. Если его нет, то перед ключевым словом in. В квадратных скобках через запятую перечисляются имена переменных и констант. Все они будут скопированы в замыкание как константы.

В качестве бонуса есть возможность дать им другое имя. Но это только для удобства и ни на что не влияет.

```
// --- Замыкания CODE SNIPPET #6 --- //
var a = 0
var b = 0
let closure = {
    [newNameForA = a] in

    print(newNameForA, b)
}

a = 10
b = 10

closure() // 0 10
```

Помните, что замыкания сами по себе reference type. Так же как и обычные функции. Поэтому несмотря на то, что мы объявили anotherCounter как константу значение переменных, находящихся в ней, может меняться. Неизменно лишь то, на какую область памяти ссылается константа. И если мы присвоим значение anotherCounter другой константе или переменной, то они будут ссылаться на одно и тоже замыкание.

```
// --- Замыкания CODE SNIPPET #7 --- //

let secondReferenceToAnotherCounter = anotherCounter
print("\(secondReferenceToAnotherCounter(3))") // running total: 6
print("\(secondReferenceToAnotherCounter(3))") // running total: 9
print("\(anotherCounter(3))") // running total: 12
```

В предыдущем примере замыкания переданные в функцию transform выполнялись во время выполнения этой функции. Т.е. синхронно. К тому моменту когда управление возвращалось назад замыкание уже было не нужно и Swift мог удалить его из памяти. Однако это не всегда так.

Рассмотрим пример. На слайде вы видите класс у которого инициализатор принимает замыкание. Однако это замыкание не используется сразу, а сохраняется в свойство класса. Для этого его нужно пометить как @escaping. Замыкания, которые не помечены @escaping могут быть более агрессивно оптимизированы.

```
// --- Замыкания CODE SNIPPET #8 --- //
class ClosureRunner {
    private var closure: () -> Void

    init(closure: @escaping () -> Void) {
        self.closure = closure
```

```
    }

    func runClosure() {
        closure()
    }
}

let closureRunner = ClosureRunner() {
    print("Closure runned")
}

closureRunner.runClosure() // Closure runned
```

До версии Swift 3.0 логика была обратная. По умолчанию замыкания были escaping, а если это было не нужно, то необходимо было пометить его как @noescape для оптимизации. Но потом было решено, что безопасное и быстрое поведение по умолчанию лучше и логику изменили.