12 Замыкания

Мы уже встречались с понятием замыканий во время изучения функций. Там они были представлены в виде безымянных функций. Как объясняет Apple в документации к языку Swift, *замыкания* (closures) — это организованные блоки с определенным функционалом, которые могут быть переданы и использованы в вашем коде.

Согласитесь, не очень доступное объяснение. Попробуем иначе.

Замыкания — это сгруппированный в контейнер код, который может быть передан в виде аргумента и многократно использован.

ПРИМЕЧАНИЕ Если вы ранее программировали на других языках, то аналогом замыканий для вас могут быть блоки (в C и Objective-C), лямбда-выражения и анонимные функции.

12.1. Функции как замыкания

Функции — это частный случай замыканий, так как они обладают следующими свойствами:

- группируют код для многократного использования;
- могут быть многократно вызваны посредством назначенного им имени;
- могут быть переданы в качестве аргументов.

Рассмотрим работу с замыканиями на примерах. Вернемся к примеру с электронным кошельком и купюрами различного достоинства в нем и напишем функцию, которая будет принимать на входе массив-кошелек и возвращать массив всех сторублевых купюр из этого кошелька (листинг 12.1).

Листинг 12.1

```
// функция отбора купюр
func handle100(wallet: [Int]) -> [Int] {
    var returnWallet = [Int]()
    for banknot in wallet {
        if banknot==100{
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
// электронный кошелек
var wallet = [10,50,100,100,5000,100,50,50,500,100]
handle100(wallet: wallet)
```

При каждом вызове функция `handle100(wallet:)` будет возвращать массив сторублевых купюр. Здесь `handle100(wallet:)` — это замыкание, так как оно обладает описанными ранее свойствами:

- группирует код;
- может быть многократно использовано;
- может быть передано в виде аргумента (с этим свойством мы знакомились, когда передавали функцию в виде входных параметров и возвращали в виде выходных значений).

Расширим функционал кода, написав дополнительную функцию для отбора купюр достоинством 1000 рублей и более (листинг 12.2).

Листинг 12.2

```
func handleMore1000(wallet: [Int]) -> [Int] {
    var returnWallet = [Int]()
    for banknot in wallet {
        if banknot>=1000{
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
var wallet = [10,50,100,100,5000,100,50,50,500,100]
handleMore1000(wallet: wallet)
```

В результате получается, что при написании двух функций в значительной мере происходит дублирование кода. Разница функций `handle100(wallet:)` и `handleMore1000(wallet:)` лишь в проверяемом условии. Остальной код в функциях один и тот же.

Для решения этой проблемы можно пойти двумя путями:

- реализовать весь функционал в пределах одной функции и передавать условие в виде аргумента;
- реализовать три функции. Первая будет группировать повторяющийся код и принимать в виде аргумента одну из оставшихся функций, которые, в свою очередь, будут производить операцию сравнения.

Если мы пойдем по первому пути, то при увеличении количества различных условий отбора единая функция будет разрастаться и в конце концов станет нечитабельной и слишком сложной. Поэтому воспользуемся вторым вариантом (листинг 12.3).

Листинг 12.3

```
// единая функция формирования результирующего массива
func handle(wallet: [Int], closure: (Int) -> Bool) -> [Int] {
    var returnWallet = [Int]()
    for banknot in wallet {
        if closure(banknot) {
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
// функция сравнения с числом 100
func compare100(banknot: Int) ->Bool {
    return banknot==100
}
// функция сравнения с числом 1000
func compareMore1000(banknot:Int) -> Bool {
    return banknot>=1000
}
var wallet = [10,50,100,100,5000,100,50,50,500,100]
handle(wallet: wallet, closure: compare100)
handle(wallet: wallet, closure: compareMore1000)
```

Функция `handle(wallet:closure:)` получает в качестве входного параметра `closure` одну из функций проверки условия и в операторе `if` вызывает переданную функцию. Функции проверки принимают на входе анализируемую купюру и возвращают `Bool` в зависимости от результата сравнения.

Чтобы получить купюры определенного достоинства, необходимо вызвать функцию `handle(wallet:closure:)` и передать в нее имя одной из функций проверки.

В итоге мы получим очень качественный код, который достаточно легко расширять.

12.2. Замыкающие выражения

Представим, что возникла необходимость написать функции для отбора купюр по многим и многим условиям (найти все полтинники, все купюры достоинством менее 1000 рублей, все купюры, которые без остатка делятся на 100, и т. д.). Условий отбора может быть великое множество. В определенный момент писать отдельную функцию проверки для каждого из них станет довольно тяжелой задачей, так как для того, чтобы использовать единую функцию проверки, необходимо знать имя проверяющей функции, а их могут быть десятки.

В подобной ситуации куда более эффективным становится использование замыкающих выражений.

Замыкающие выражения — это безымянные замыкания, написанные в облегченном синтаксисе.

СИНТАКСИС

```
{ (входные_аргументы) -> ТипВозвращаемогоЗначения in
    тело_замыкающего_выражения
}
```

Замыкающее выражение пишется в фигурных скобках. После указания перечня входных аргументов и типа возвращаемого значения ставится ключевое слово in, после которого следует тело замыкания.

В соответствии с третьим свойством замыканий замыкающие выражения можно передавать в качестве аргументов. Давайте вызовем написанную ранее функцию `handle(wallet:closure:)`, передавая ей замыкающее выражение в качестве входного параметра (листинг 12.4).

Листинг 12.4

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet: wallet, closure: {(banknot: Int) -> Bool in
    return banknot>=1000
})
// отбор купюр достоинством 100 рублей
handle(wallet: wallet, closure: {(banknot: Int) -> Bool in
    return banknot==100
})
```

В результате необходимость в существовании функций `compare100(banknot:)` и `compareMore1000(banknot:)` отпадает, так как код

проверяющей функции передается напрямую в качестве аргумента `closure`.

Налицо увеличение гибкости и уменьшение объема кода.

Замыкающие выражения, которые мы встречали при изучении функций, не имели каких-либо входных параметров, да и их функциональный тип не указывался.

ПРИМЕЧАНИЕ Облегченный синтаксис замыкающих выражений упрощает работу и позволяет не писать лишний код, а оптимизированный код экономит время и приносит вам дополнительную выгоду.

Теперь приступим к оптимизации уже используемых замыкающих выражений. При объявлении входного параметра `closure` в функции `handle(wallet:closure:)` указывается его функциональный тип (он принимает функцию типа `(Int) -> Bool`), поэтому при передаче замыкающего выражения нет необходимости дублировать данную информацию (листинг 12.5).

Листинг 12.5

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet: wallet, closure: {banknot in
    return banknot>=1000
})
// отбор купюр достоинством 100 рублей
handle(wallet: wallet, closure: {banknot in
    return banknot==100
})
```

В замыкающем выражении перед ключевым словом `in` необходимо передать только имя, которое будет присвоено передаваемому в него значению очередного элемента массива `wallet`.

В коде функции `handle` при вызове функции `closure` ей передается параметр `banknot` — именно по этой причине мы указываем в качестве входного аргумента переменную с аналогичным названием.

12.3. Неявное возвращение значения

Замыкающие выражения позволяют в значительной мере оптимизировать программы. Это лишь одна из многих возможностей Swift, обеспечивающих красивый и понятный исходный код для ваших проектов.

Если тело замыкающего выражения содержит всего одно выражение, которое возвращает некоторое значение (с использованием оператора `return`), то такие замыкания могут неявно возвращать выходное

значение. Неявно — значит без использования оператора `return` (листинг 12.6).

Листинг 12.6

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet: wallet,
closure: {banknot in banknot>=1000})
// отбор купюр достоинством 100 рублей
handle(wallet: wallet,
closure: {banknot in banknot==100})
```

В результате мы добились того, что замыкающее выражение записывается всего в одну короткую строку и при этом код становится понятнее.

12.4. Сокращенные имена параметров

Продолжим оптимизацию используемых нами замыканий. Для однострочных замыкающих выражений Swift автоматически предоставляет доступ к входным аргументам с помощью сокращенных имен доступа. Сокращенные имена доступа к входным аргументам пишутся в форме `$номер_параметра`. Номера входных параметров начинаются с нуля.

ПРИМЕЧАНИЕ В сокращенной форме записи имен входных параметров обозначение $0 указывает на первый передаваемый аргумент. Для доступа ко второму аргументу необходимо использовать обозначение $1, к третьему — $2 и т. д.

Перепишем вызов функции `handle(wallet:closure:)` с использованием сокращенных имен (листинг 12.7).

Листинг 12.7

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet: wallet,
closure: {$0>=1000})
// отбор купюр достоинством 100 рублей
handle(wallet: wallet,
closure: {$0==100})
```

Здесь `$0` — это входной аргумент `banknot` входного аргумента-замыкания `closure` в функции `handle(wallet:closure:)`.

В тех случаях, когда входной параметр-функция состоит всего из одного выражения, использование замыкающих выражений делает код более понятным.

Если входной параметр-функция расположен последним в списке входных параметров функции (как в данном случае в функции

handle(wallet:closure:), где параметр closure является последним), Swift позволяет вынести его значение (замыкающее выражение) за круглые скобки (листинг 12.8).

Листинг 12.8

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet: wallet)
{$0>=1000}
// отбор купюр достоинством 100 рублей
handle(wallet: wallet)
{$0==100}
```

Данный пример, возможно, не в полной мере демонстрирует необходимость выноса замыкания за круглые скобки, но в случае, когда замыкающее выражение многострочное, данный прием делает код значительно понятнее. В листинге 12.9 помимо массива wallet, который содержит купюры, находящиеся в кошельке, мы создадим массив successbanknots, содержащий в себе только разрешенные для использования купюры. Задача состоит в том, чтобы из wallet отобрать все купюры, которые представлены в successbanknots.

Листинг 12.9

```
let successbanknots = [100, 500]
handle(wallet: wallet)
{banknot in
    for number in successbanknots {
        if number == banknot {
            return true
        }
    }
    return false
}
```

Внутри замыкающего выражения мы обращаемся к массиву successbanknot без его непосредственной передачи через входные аргументы, но работа внутри производится не с самим массивом successbanknot, а с его копией. В связи с этой особенностью Swift любые изменения массива внутри замыкающего выражения не привнесут никаких изменений в исходный массив.

12.5. Переменные-замыкания

Ранее, когда мы рассматривали работу с функциями, мы передавали в качестве значения переменной безымянную функцию. Так мы узна-

ли, что переменные могут хранить в себе замыкающие выражения, но при этом ограничились лишь присвоением значения, упустив вопросы указания функционального типа.

Рассмотрим пример из листинга 12.10, в котором определяется константа `closure`, содержащая в виде значения замыкающее выражение.

Листинг 12.10

```
1  let closure : () -> () = {
2      print("Замыкающее выражение")
3  }
4  closure()
```

Консоль:

```
Замыкающее выражение
```

Так как данное замыкающее выражение не имеет входных параметров и возвращаемого значения, то его функциональный тип равен `() -> ()`.

Для вызова записанного в константу замыкающего выражения необходимо написать имя константы с круглыми скобками, то есть точно так же, как мы вызываем функции.

Для того чтобы передать в замыкание некоторые значения (в качестве входящих параметров), необходимо описать их в функциональном типе данной константы.

ПРИМЕЧАНИЕ Обратите внимание, что при сохранении замыкания в переменную или константу входные аргументы не должны иметь внешних имен. Имена можно либо вообще опустить, либо указать внешние, такие как знак подчеркивания (_).

Для доступа к значениям входных аргументов внутри замыкающего выражения необходимо использовать сокращенные имена доступа (`$0`, `$1` и т. д.), как показано в листинге 12.11.

Листинг 12.11

```
1  var sum: (_ numOne: Int, _ numTwo: Int) -> Int = {
2      return $0 + $1
3  }
4  sum(10, 34)
```

Здесь замыкающее выражение, хранящееся в константе `sum`, принимает два входных аргумента типа `Int` и возвращает их сумму.

ПРИМЕЧАНИЕ Замыкающие выражения могут храниться как в константах, так и в переменных. Выбирайте правильный тип параметра в зависимости от того, будет ли перезаписано его значение.

12.6. Метод сортировки массивов

Swift предлагает большое количество функций и методов, позволяющих значительно упростить разработку приложений. Одним из таких методов является `sorted(by:)`, предназначенный для сортировки массивов, как строковых, так и числовых. Он принимает на входе массив, который необходимо отсортировать, и условие сортировки.

Принимаемое условие сортировки — это обыкновенное замыкающее выражение, которое вызывается внутри метода `sorted(by:)`, принимает на входе два очередных элемента сортируемого массива и возвращает значение `Bool` в зависимости от результата их сравнения. Для того чтобы получить отсортированный массив, необходимо передать соответствующее замыкание. В листинге 12.12 мы отсортируем массив `myArray` таким образом, чтобы элементы были расположены по возрастанию. Для этого в метод `sorted(by:)` передадим такое замыкающее выражение, которое возвращает `true`, когда второе из сравниваемых чисел больше.

Листинг 12.12

```
1  var array = [1,44,81,4,277,50,101,51,8]
2  array.sorted(by: { (first: Int, second: Int) -> Bool in
3  return first < second
4  })
```

Теперь применим все рассмотренные ранее способы оптимизации замыкающих выражений:

- уберем функциональный тип замыкания;
- заменим имена переменных именами в сокращенной форме.

В результате получится выражение, приведенное в листинге 12.13. Как и в предыдущем примере, здесь тоже необходимо отсортировать массив `myArray` таким образом, чтобы элементы были расположены по возрастанию. Для этого в метод `sorted(by:)` передается такое замыкающее выражение, которое возвращает `true`, когда второе из сравниваемых чисел больше.

Листинг 12.13

```
1  var array = [1,44,81,4,277,50,101,51,8]
2  var sortedArray = array.sorted(by: {$0<$1})
```

В результате код получается более читабельным и красивым.

В Swift существует особая реализация замыкания сравнения переменных, которая принимает на входе два сравниваемых элемента, а воз-

вращает значения типа `Bool`. Она называется бинарным оператором сравнения. С ним мы уже давно знакомы.

То есть в качестве замыкающего выражения можно написать бинарный оператор сравнения без указания имен сравниваемых параметров, так как данный оператор является бинарным — совершающим операцию с двумя операндами (листинг 12.14).

Листинг 12.14

```
1  var array = [1,44,81,4,277,50,101,51,8]
2  var sortedArray = array.sorted(by: <)
```

Надеюсь, вы приятно удивлены потрясающими возможностями Swift!

12.7. Каррирование функций

Одной из непростых для понимания, но в то же время очень важных для разработки тем является каррирование (от *англ.* currying). Несмотря на всю его сложность, знать и использовать этот механизм обязательно, не зря при подготовке Swift 3 в Apple обратили на него особое внимание и внесли некоторые правки, направленные на повышение читабельности кода при использовании каррирования.

Каррирование — это процесс, при котором функция от нескольких аргументов преобразуется в функцию (или набор функций) от одного аргумента. Это становится возможным благодаря тому, что в качестве выходного значения функции может выступать другая функция.

НАЧИНАЮЩИМ Предположим, что у нас есть функция, которая принимает на вход три параметра. При каррировании мы получим функцию, которая принимает на вход один аргумент, а возвращает функцию, которая также принимает на вход один аргумент и в свою очередь также возвращает функцию, которая принимает на вход один аргумент и возвращает некоторый результат. То есть получается своеобразная линейка функций.

Разложим все по полочкам.

- У нас есть функция с типом (Int, Int, Int)->Int, которая зависит от трех входных аргументов типа Int и возвращает значение типа Int.
- При каррировании мы получим функцию типа (Int)->(Int)->(Int)->Int. Для этого выполним следующие шаги:
    - обозначим функциональный тип (Int)->Int последней функции в цепочке как A. Тогда каррированная функция будет выглядеть как (Int)->(Int)->A;
    - обозначим функциональный тип (Int)->A как B. Тогда каррированная функция будет выглядеть как (Int)->B.

- Получается, что наша функция принимает на вход одно целое число и возвращает значение типа B.
- Значение типа B, в свою очередь, также является функцией, которая принимает на вход одно целое число и возвращает значение типа A.
- Значение типа A также является функцией, которая принимает на вход одно целое число и возвращает одно целое число.
- В результате одну функцию, зависящую от трех параметров, мы реорганизовали (каррировали) к трем взаимозависимым функциям, каждая из которых зависит всего от одного значения.

Рассмотрим пример. Существует функция с типом `(Int, Int)->Int`, которая получает на вход два целочисленных значения, производит некоторую операцию и возвращает ответ в виде целого числа (листинг 12.15).

Листинг 12.15

```
1  func sum(x: Int, y: Int) -> Int {
2      return x + y
3  }
4  sum(x: 1,y: 4) // вернет 5
```

С целью каррирования напишем новую функцию, которая принимает на вход всего один целочисленный параметр, а возвращает замыкание типа `(Int)->Int` (листинг 12.16).

Листинг 12.16

```
1  func sum2(_ x: Int) -> (Int) -> Int {
2      return {  return $0+x }
3  }
4  var closure = sum2(1)  (Int)->Int
5  closure(12)  13
```

Переменная `closure` получает в качестве значения замыкание, которому мы можем передать входной параметр. Прелесть каррирования в том, что мы можем объединить вызов функции `sum2(x:)` и передачу значения в возвращаемое ею замыкание (листинг 12.17).

Листинг 12.17

```
1  sum2(5)(12) // вернет 17
```

В результате мы получаем прекрасно читаемую и удобную в использовании функцию.

Если говорить о пользе, то каррирование хорошо именно тем, что устраняет зависимость функции от нескольких параметров. Мы мо-

жем вызвать функцию, как только получим первое нужное значение, и далее при необходимости многократно вызывать возвращенное ею замыкание при поступлении второго требуемого параметра (листинг 12.18).

Листинг 12.18

```
var closure = sum2( 1)
closure(12) // вернет 13
closure(19) // вернет 20
```

12.8. Захват переменных

Swift позволяет зафиксировать значения внешних по отношению к замыканию параметров, которые они имели на момент его определения.

Синтаксис захвата переменных

Обратимся к примеру в листинге 12.19. Существуют два параметра, `a` и `b`, которые не передаются в качестве входных аргументов в замыкание, но используются им в вычислениях. При каждом вызове такого замыкания оно будет определять значения данных параметров, прежде чем приступить к выполнению операции с их участием.

Листинг 12.19

```
var a = 1
var b = 2
let closureSum : () -> Int = {
    return a+b
}
closureSum() // 3
a = 3
b = 4
closureSum() // 7
```

Замыкание, хранящееся в константе `closureSum`, складывает значения перемененных `a` и `b`. При изменении их значений возвращаемое замыканием значение меняется.

Существует способ «захватить» значения параметров, то есть зафиксировать те значения, которые имеют эти параметры на момент написания замыкающего выражения. Для этого в начале замыкания в квадратных скобках необходимо перечислить захватываемые переменные, разделив их запятой, после чего указать ключевое слово `in`. Перепишем инициализированное переменной `closureSum` замыкание

таким образом, чтобы оно захватывало первоначальные значения переменных a и b (листинг 12.20).

Листинг 12.20

```
var a = 1
var b = 2
let closureSum : () -> Int = {
    [a,b] in
    return a+b
}
closureSum() // 3
a = 3
b = 4
 closureSum() // 3
```

Замыкание, хранящееся в константе closureSum, складывает значения перемененных a и b. При изменении этих значений возвращаемое замыканием значение не меняется.

Захват вложенной функцией

Другим способом захвата значения внешнего параметра является вложенная функция, написанная в теле другой функции. Вложенная функция может захватывать произвольные переменные, константы и даже входные аргументы, объявленные в родительской функции.

Рассмотрим пример из листинга 12.21.

Листинг 12.21

```
func makeIncrement(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func increment() -> Int {
        runningTotal += amount
        return runningTotal
    }
return incrementer
}
```

Функция makeIncrement(forIncrement:) возвращает значение с функциональным типом ()->Int. Это значит, что вернется замыкание, не имеющее входных аргументов и возвращающее целочисленное значение.

Функция makeIncrement(forIncrement:) определяет два параметра:

- runningTotal — переменную типа Int, объявляемую в теле функции. Именно ее значение является результатом работы всей конструкции;

- `amount` — входной аргумент, имеющий тип `Int`. Он определяет, насколько увеличится значение `runningTotal` при очередном обращении.

Вложенная функция `increment()` не имеет входных или объявляемых параметров, но при этом обращается к `runningTotal` и `amount` внутри своей реализации. Она делает это в автоматическом режиме путем захвата значений обоих параметров по ссылке. Захват значений по ссылке гарантирует, что измененные значения параметров не исчезнут после окончания работы функции `makeIncrement(forIncrement:)` и будут доступны при повторном вызове функции `increment()`.

Теперь обратимся к листингу 12.22.

Листинг 12.22

```
var incrementByTen = makeIncrementer(forIncrement: 10)
var incrementBySeven = makeIncrementer(forIncrement: 7)
incrementByTen()   // вернется 10
incrementByTen()   // вернется 20
incrementByTen()   // вернется 30
incrementBySeven() // вернется 7
incrementBySeven() // вернется 14
incrementBySeven() // вернется 21
```

В переменных `incrementByTen` и `incrementBySeven` хранятся возвращаемые функцией `makeIncrement(forIncrement:)` замыкания. В первом случае значение `runningTotal` увеличивается на 10, а во втором — на 7. Каждая из переменных хранит свою копию захваченного значения `runningTotal`, именно поэтому при их использовании увеличиваемые значения не пересекаются и увеличиваются независимо друг от друга.

ВНИМАНИЕ Так как в переменных incrementByTen и incrementBySeven хранятся замыкания, то при доступе к ним после их имени необходимо использовать скобки (по аналогии с доступом к функциям).

12.9. Замыкания — это тип-ссылка

Функциональный тип данных — это тип-ссылка (reference type). Это значит, что замыкания передаются не копированием, а с помощью ссылки на область памяти, где хранится это замыкание.

Рассмотрим пример, описанный в листинге 12.23.

Листинг 12.23

```
var incrementByFive = makeIncrement(forIncrement: 5)
var copyIncrementByFive = incrementByFive
```

В данном примере используется функция `makeIncrement(forIncrement:)`, объявленная ранее. Напомню, она возвращает замыкание типа `()->Int`, которое в данном случае предназначено для увеличения значения на 5. Возвращаемое замыкание записывается в переменную `incrementByFive`, после чего копируется в переменную `copyIncrementByFive`. В результате можно обратиться к одному и тому же замыканию, используя как `copyIncrementByFive`, так и `incrementByFive` (листинг 12.24).

Листинг 12.24

```
1  incrementByFive() // вернет 5
2  copyIncrementByFive() // вернет 10
3  incrementByFive() // вернет 15
```

Как вы можете видеть, какую бы функцию мы ни использовали, происходит модификация одного и того же значения (каждое последующее значение больше предыдущего на 5). Это обусловлено тем, что замыкания — ссылочный тип данных, или тип-ссылка, или reference type.

12.10. Автозамыкания

Автозамыкания — это замыкания, которые автоматически создаются из переданного выражения. Иными словами, может существовать функция, имеющая один или несколько входных параметров, которые при ее вызове передаются как значения, но во внутренней реализации функции используются как самостоятельные замыкания. Рассмотрим пример из листинга 12.25.

Листинг 12.25

```
1  var arrayOfNames = ["Helga", "Bazil", "Alex"]
2  func printName(nextName: String ) {
3      // какой-либо код
4      print(nextName)
5  }
6  printName(nextName: arrayOfNames.remove(at: 0))
```

Консоль:

```
Helga
```

При каждом вызове функции `printName(nextName:)` в качестве входного значения ей передается результат вызова метода `remove(at:)` массива `arrayOfNames`.

Независимо от того, в какой части функции будет использоваться переданный параметр (или не будет использоваться вовсе), значение,

возвращаемое методом `remove(at:)`, будет вычислено в момент вызова функции `printName(nextName:)`. Получается, что передаваемое значение вычисляется независимо от того, нужно ли оно в ходе выполнения функции.

Отличным решением данной проблемы станет использование ленивых вычислений, то есть таких вычислений, которые будут выполняться лишь в тот момент, когда это понадобится. Для того чтобы реализовать этот подход, можно передавать в функцию `printName(nextName:)` замыкание, которое будет вычисляться в тот момент, когда к нему обратятся (листинг 12.26).

Листинг 12.26

```
1  var arrayOfNames = ["Helga", "Bazil", "Alex"]
2  func printName(nextName: ()->String) {
3      // какой-либо код
4      print(nextName())
5  }
6  printName(nextName: {arrayOfNames.remove(at: 0)})
```

Консоль:

```
Helga
```

Для решения этой задачи потребовалось изменить тип входного параметра `nextName` на `()->String` и заключить передаваемый метод `remove(at:)` в фигурные скобки. Теперь внутри реализации функции `printName(nextName:)` к входному аргументу `nextName` необходимо обращаться как к самостоятельной функции (с использованием круглых скобок после имени параметра). Таким образом, значение метода `remove(at:)` будет вычислено именно в тот момент, когда оно понадобится, а не в тот момент, когда оно будет передано. Единственным недостатком данного подхода является то, что входной параметр должен быть заключен в фигурные скобки, а это несколько усложняет использование функции и чтение кода.

С помощью автозамыканий можно использовать положительные функции обоих рассмотренных примеров: отложить вычисление переданного значения и передавать значение в виде значения (без фигурных скобок).

Для того чтобы реализовать автозамыкание, требуется, чтобы выполнялись следующие требования.

- Входной аргумент должен иметь функциональный тип.

  В примере, приведенном ранее, аргумент `nextName` уже имеет функциональный тип `()->String`.

- Функциональный тип не должен определять типы входных параметров.

  В примере типы входных параметров не определены (пустые скобки).

- Функциональный тип должен определять тип возвращаемого значения.

  В примере тип возвращаемого значения определен как `String`.

- Переданное выражение должно возвращать значение того же типа, которое определено в функциональном типе замыкания.

  В примере передаваемая в качестве входного аргумента функция возвращает значение типа `String` точно так же, как определено функциональным типом входного аргумента.

- Перед функциональным типом необходимо использовать атрибут `@autoclosure`.

- Передаваемое значение должно указываться без фигурных скобок.

Перепишем код из предыдущего листинга в соответствии с указанными требованиями (листинг 12.27).

Листинг 12.27

```
1  var arrayOfNames = ["Helga", "Bazil", "Alex"]
2  func printName(nextName: @autoclosure ()->String) {
3      // какой-либо код
4      print(nextName())
5  }
6 printName(arrayOfNames.remove(at: 0))
```

Консоль:

```
Helga
```

Теперь метод `remove(at:)` передается в функцию `printName(nextName:)` как обычный аргумент, без использования фигурных скобок, но внутри тела используется как самостоятельная функция.

Ярким примером глобальной функции, входящей в стандартные возможности Swift и использующей механизм автозамыканий, является функция `assert(condition:message:file:line:)`. Аргументы `condition` и `message` — это автозамыкания, первое из которых вычисляется только в случае активного debug-режима, а второе — только в случае, когда `condition` соответствует `false`.

11.12. Выходящие замыкания

Как вы уже неоднократно убеждались и убедитесь еще не раз, Swift — очень умный язык программирования. Он старается экономить ваши ресурсы там, где вы можете об этом даже не догадываться.

По умолчанию все переданные в функцию замыкания имеют ограниченную этой функцией область видимости, то есть если вы решите сохранить замыкание для дальнейшего использования, то получите сообщение об ошибке. Другими словами, все переданные в функцию замыкания называются не выходящими за пределы ее тела. Если Swift видит, что область, где замыкание доступно, ограничена, он при первой же возможности удалит его, чтобы освободить и не расходовать оперативную память.

Для того чтобы позволить замыканию выйти за пределы области видимости функции, необходимо указать атрибут `@escaping` перед функциональным типом при описании входных параметров функции.

ПРИМЕЧАНИЕ

В Swift 3.0 был изменен подход к передаче замыканий в качестве параметра в функции. Ранее для того, чтобы запретить замыканию выход за пределы области видимости функции, требовалось указать атрибут @noescaping, теперь же все передаваемые в функцию замыкания автоматически помечаются как не выходящие.

Рассмотрим пример. Предположим что в программе есть специальная переменная, предназначенная для хранения замыканий, то есть являющаяся коллекцией замыканий (листинг 12.28).

Листинг 12.28

```
1  var arrayOfClosures: [()->Int] = []
```

Пока еще пустой массив `arrayOfClosures` может хранить в себе замыкания с функциональным типом `()->Int`. Реализуем функцию, добавляющую в этот массив переданные ей в качестве аргументов замыкания (листинг 12.29).

Листинг 12.29

```
1  func addNewClosureInArray(_ newClosure: ()->Int){
2    arrayOfClosures.append(newClosure) // ошибка
3  }
```

Xcode сообщит вам об ошибке. И на то есть две причины:

- Замыкание — это тип-ссылка, то есть оно передается по ссылке, но не копированием.

- Замыкание, которое будет храниться в параметре `newClosure`, будет иметь ограниченную телом функции область видимости, а значит, не может быть добавлено в глобальную (по отношению к телу функции) переменную `arrayOfClosures`.

Для решения этой проблемы необходимо указать, что замыкание, хранящееся в переменной `newClosure`, является выходящим. Для этого перед описанием функционального типа данного аргумента укажите атрибут `@escaping`, после чего вы сможете передать в функцию `addNewClosureInArray(_:)` произвольное замыкание (листинг 12.30).

Листинг 12.30

```
1  func addNewClosureInArray(_ newClosure: @escaping ()->Int){
2    arrayOfClosures.append(newClosure)
3  }
4  addNewClosureInArray({return 100})
5  addNewClosureInArray{return 1000}
6  arrayOfClosures[0]() // 100
7  arrayOfClosures[1]() // 1000
```

Обратите внимание на то, что в одном случае замыкание передается с круглыми скобками, а в другом — без них. Так как функция `addNewClosureInArray(_:)` имеет один входной аргумент, то допускаются оба варианта.

ПРИМЕЧАНИЕ

Если вы передаете замыкание в виде параметра, то можете использовать модификатор inout вместо @escaping.