Практическая работа № 10

**Замыкания**

Цель работы:

1. Изучить работу замыканий, научиться применять их на практике.
2. Получить навыки использования замыканий.
3. Создать консольное приложение, работа которого будет построена на использовании замыканий.

Краткие теоретические сведения

**Замыкания**

Замыкания - это самодостаточные блоки с определенным функционалом, которые могут быть переданы и использованы в вашем коде. Замыкания в Swift похожи на блоки в C и Objective-C, и лямбды в других языках программирования.

Замыкания могут захватывать и хранить ссылки на любые константы и переменные из контекста, в котором они объявлены. Эта процедура известна как заключение этих констант и переменных, отсюда и название "замыкание". Swift выполняет всю работу с управлением памятью при захвате за вас.

Глобальные и вложенные функцииявляются частным случаем замыканий. Замыкания принимают одну из трех форм:

* Глобальные функции являются замыканиями, у которых есть имя и которые не захватывают никакие значения.
* Вложенные функции являются замыканиями, у которых есть имя и которые могут захватывать значения из включающей их функции.
* Выражения замыкания являются безымянными замыканиями, написанные в облегченном синтаксисе, которые могут захватывать значения из их окружающего контекста.

Выражения замыкания в Swift имеют четкий, ясный, оптимизированный синтаксис в распространенных сценариях. Эти оптимизации включают:

* Вывод типа параметра и возврат типа значения из контекста
* Неявные возвращающиеся значения однострочных замыканий
* Сокращенные имена параметров
* Синтаксис последующих замыканий

# Захват значений

Замыкания могут захватывать константы и переменные из окружающего контекста, в котором они объявлены. После захвата замыкание может ссылаться или модифицировать значения этих констант и переменных внутри своего тела, даже если область, в которой были объявлены эти константы и переменные уже больше не существует.

В Swift самая простая форма замыкания может захватывать значения из вложенных функций, написанных внутри тела других функций. Вложенная функция может захватить любые значения из аргументов окружающей ее функции, а так же константы и переменные, объявленные внутри тела внешней функции.

Вот пример функции makeIncrementer, которая содержит вложенную функцию incrementer. Вложенная функция incrementer() захватывает два значения runningTotal и amount из окружающего контекста. После захвата этих значений incrementer возвращается функцией makeIncrementer как замыкание, которое увеличивает runningTotal на amount каждый раз как вызывается.

func makeIncrementer(forIncrement amount: Int) -> () ->Int {

var runningTotal = 0

func incrementer() ->Int {

runningTotal += amount

return runningTotal

}

return incrementer

}

Возвращаемый тип makeIncrementer Void -> Int. Это значит, что он возвращает функцию, а не простое значение. Возвращенная функция не имеет параметров и возвращает Int каждый раз как ее вызывают.

Функция incrementer() не имеет ни одного параметра и она ссылается на runningTotal и amount внутри тела функции. Она делает это, захватывая существующие значения от runningTotal и amount из окружающей функции и используя их внутри. Захват ссылки дает гарантию того, что runningTotal не исчезнет при окончании вызова makeIncrementer и гарантирует, что runningTotal останется переменной в следующий раз, когда будет вызвана функция incrementer().

let incrementByTen = makeIncrementer(forIncrement: 10)

Этот пример заставляет константу incrementByTen ссылаться на функцию инкрементора, которая добавляет 10 к значению переменной runningTotal каждый раз как вызывается. Многократный вызов функции показывает ее в действии:

incrementByTen()

// возвращает 10

incrementByTen()

// возвращает 20

incrementByTen()

// возвращает 30

Если вы создаете второй инкрементор, он будет иметь свою собственную ссылку на новую отдельную переменную runningTotal :

let incrementBySeven = makeIncrementer(forIncrement: 7)

incrementBySeven()

//возвращает значение 7

Повторный вызов первоначального инкрементора( incrementByTen ) заставит увеличиваться его собственную переменную runningTotal и никак не повлияет на переменную, захваченную в incrementBySeven :

incrementByTen()

//возвращает 40

#### Заметка

Если вы присваиваете замыкание свойству экземпляра класса, и замыкание захватывает этот экземпляр по ссылке на него или его члены, вы создаете сильные обратные связи между экземпляром и замыканием. Swift использует списки захвата, для разрыва этих сильных обратных связей.

# Замыкающие выражения

Вложенные функции являются удобным способом для обозначения и объявления самоорганизованных блоков кода, которые являются частью более крупной функции. Тем не менее, иногда полезно писать короткие версии функциональных конструкций, без полного объявления и указания имени. Это особенно верно, когда вы работаете с функциями, которые принимают другие функции в виде одного из своих параметров.

Замыкающие выражения, являются способом написания встроенных замыканий через краткий и специализированный синтаксис. Замыкающие выражения обеспечивают несколько синтаксических оптимизаций для написания замыканий в краткой форме, без потери ясности и намерений. Примеры замыкающих выражений ниже, показывают эти оптимизации путем рассмотрения метода sorted(by:) при нескольких итерациях, каждая из которых изображает ту же функциональность в более сжатой форме.

## Метод sorted

В стандартной библиотеке Swift есть метод sorted(by:), который сортирует массив значений определенного типа, основываясь на результате сортирующего замыкания, которые вы ему передадите. После завершения процесса сортировки, метод sorted(by:) возвращает новый массив того же типа и размера как старый, с элементами в правильном порядке сортировки. Исходный массив не изменяется методом sorted(by:).

Примеры замыкающих выражений ниже используют метод sorted(by:)для сортировки массива из String значений в обратном алфавитном порядке. Вот исходный массив для сортировки:

let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

Замыкание метода sorted(by:) принимает два аргумента одного и того же типа, что и содержимое массива, и возвращает Bool значение, которое решает поставить ли первое значение перед вторым, или после второго. Замыкание сортировки должно вернуть true, если первое значение должно быть до второго значения, и false в противном случае.

Этот пример сортирует массив из String значений, так что сортирующее замыкание должно быть функцией с типом

(String, String) ->Bool.

Один из способов обеспечить сортирующее замыкание, это написать нормальную функцию нужного типа, и передать его в качестве аргумента метода sorted(by:):

func backward(\_ s1: String, \_ s2: String) ->Bool {

return s1 > s2

}

var reversedNames = names.sorted(by: backward)

// reversedNamesравен ["Ewa", "Daniella", "Chris", "Barry", "Alex"]

Если первая строка (s1) больше чем вторая строка (s2), функция backward(\_:\_:) возвращает true, что указывает, что s1 должна быть перед s2 в сортированном массиве. Для символов в строках, "больше чем" означает "появляется в алфавите позже, чем". Это означает что буква "B" "больше чем" буква "А", а строка "Tom" больше чем строка "Tim". Это делает обратную алфавитную сортировку, с "Barry" поставленным перед "Alex", и так далее.

### Синтаксис

Синтаксис замыкающего выражения имеет следующую общую форму:

1. { (параметры) -> тип результата in
2. выражения
3. }

Синтаксис замыкающего выражения может использовать сквозные параметры. Значения по умолчанию не могут быть переданы. Вариативные параметры могут быть использованы в любом месте в списке параметров. Кортежи также могут быть использованы как типы параметров и как типы возвращаемого значения.

Пример ниже показывает версию функции backward(\_:\_:) с использованием замыкающего выражения:

reversedNames = names.sorted(by: { (s1: String, s2: String) ->Boolin

return s1 > s2

})

Обратите внимание, что объявление типов параметров и типа возвращаемого значения для этого однострочного замыкания идентично объявлению из функции backward(\_:\_:). В обоих случаях, оно пишется в виде (s1:String, s2: String) ->Bool. Тем не менее, для однострочных замыкающих выражений, параметры и тип возвращаемого значения пишутся внутри фигурных скобок, а не вне их.

Начало тела замыкания содержит ключевое слово in. Это ключевое слово указывает, что объявление параметров и возвращаемого значения замыкания закончено, и тело замыкания вот-вот начнется.

Поскольку тело замыкания настолько короткое, оно может быть записано в одну строку:

reversedNames = names.sorted(by: { (s1: String, s2: String) ->Boolinreturn s1 > s2 })

## Определение типа из контекста

Поскольку сортирующее замыкание передается как аргумент метода, Swift может вывести типы его параметров и тип возвращаемого значения, через тип параметра метода sorted(by:). Этот параметр ожидает функцию имеющую тип (String, String) ->Bool. Это означает что типы (String, String) и Bool не нужно писать в объявлении замыкающего выражения. Поскольку все типы могут быть выведены, стрелка результата ( -> ) и скобки вокруг имен параметров также могут быть опущены:

reversedNames = names.sorted(by: { s1, s2 inreturn s1 > s2 })

Тем не менее, вы всё равно можете явно указать типы, если хотите. И делать это предполагается, если это поможет избежать двусмысленности для читателей вашего кода.

## Неявные возвращаемые значения из замыканий с одним выражением

Замыкания с одним выражением могут неявно возвращать результат своего выражения через опускание ключевого слова return из их объявления, как показано в этой версии предыдущего примера:

reversedNames = names.sorted(by: { s1, s2 in s1 > s2 })

Здесь, функциональный тип аргумента метода sorted(by:)дает понять, что замыкание вернет Bool значение. Поскольку тело замыкания содержит одно выражение (s1 > s2), которое возвращает Bool значение, то нет никакой двусмысленности, и ключевое слово return можно опустить.

## Сокращенные имена параметров

Swift автоматически предоставляет сокращённые имена для однострочных замыканий, которые могут быть использованы для обращения к значениям параметров замыкания через имена $0, $1, $2, и так далее.

Если вы используете эти сокращенные имена параметров с вашим замыкающим выражением, вы можете пропустить список параметров замыкания из его объявления, а количество и тип сокращенных имен параметров будет выведено из ожидаемого типа метода. Ключевое слово in также может быть опущено, поскольку замыкающее выражение полностью состоит из его тела:

reversedNames = names.sorted(by: { $0> $1 })

Здесь, $0 и $1 обращаются к первому и второму String параметру замыкания.

## Операторные функции

Здесь есть на самом деле более короткий способ написать замыкающее выражение выше. Тип String в Swift определяет свою специфичную для строк реализацию оператора больше ( > ) как функции, имеющей два строковых параметра и возвращающей значение типа Bool. Это точно соответствует типу метода, для параметра метода sorted(by:). Таким образом, вы можете просто написать оператор больше, а Swift будет считать, что вы хотите использовать специфичную для строк реализацию:

reversedNames = names.sorted(by: >)

# Автозамыкания (autoclosures)

Автозамыкания - замыкания, которые автоматически создаются для заключения выражения, которое было передано в качестве аргумента функции. Такие замыкания не принимают никаких аргументов при вызове и возвращают значение выражения, которое заключено внутри нее. Синтаксически вы можете опустить круглые скобки функции вокруг параметров функции, просто записав обычное выражение вместо явного замыкания.

Нет ничего необычного в вызове функций, которые принимают автозамыкания, но необычным является реализовывать такие функции. Например, функция assert(condition:message:file:line:) принимает автозамыкания на место condition и message параметров. Ее параметр condition вычисляется только в сборке дебаггера, а параметр message вычисляется, если только condition равен false.

Автозамыкания позволяют вам откладывать вычисления, потому как код внутри них не исполняется, пока вы сами его не запустите. Это полезно для кода, который может иметь сторонние эффекты или просто является дорогим в вычислительном отношении, потому что вы можете контролировать время исполнения этого кода. Пример ниже отображает как замыкания откладывают вычисления:

var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

print(customersInLine.count)

// Выведет "5"

let customerProvider = { customersInLine.remove(at: 0) }

print(customersInLine.count)

// Выведет "5"

print("Now serving \(customerProvider())!")

// Выведет "Now serving Chris!"

print(customersInLine.count)

// Выведет "4"

Даже если первый элемент массива customersInLine удаляется кодом внутри замыкания, элемент массива фактически не удаляется до тех пор пока само замыкание не будет вызвано. Если замыкание так и не вызывается, то выражение внутри него никогда не выполнится и, соответственно, элемент не будет удален из массива. Обратите внимание, что customerProvider является не String, а () ->String, то есть функция не принимает аргументов, но возвращает строку. Вы получите то же самое поведение, когда сделаете это внутри функции:

// customersInLineравен ["Alex", "Ewa", "Barry", "Daniella"]

func serve(customer customerProvider: () ->String) {

print("Now serving \(customerProvider())!")

}

serve(customer: { customersInLine.remove(at: 0) } )

// Выведет "NowservingAlex!"

Функция serve(customer:) описанная выше принимает явное замыкание, которое возвращает имя клиента. Версия функции serve(customer:) ниже выполняет ту же самую операцию, но вместо использования явного замыкания, она использует автозамыкание, поставив маркировку при помощи атрибута @autoclosure. Теперь вы можете вызывать функцию, как будто бы она принимает аргумент String вместо замыкания. Аргумент автоматически преобразуется в замыкание, потому что тип параметра customerProvider имеет атрибут @autoclosure.

// customersInLineравен ["Ewa", "Barry", "Daniella"]

func serve(customer customerProvider: @autoclosure () ->String) {

print("Now serving \(customerProvider())!")

}

serve(customer: customersInLine.remove(at: 0))

// Выведет "NowservingEwa!"

#### Заметка

Слишком частое использование автозамыканий может сделать ваш код сложным для чтения. Контекст и имя функции должны обеспечивать ясность отложенности исполнения кода.

Если вы хотите чтобы автозамыкание могло сбежать, то вам нужно использовать оба атрибута и @autoclosure, и @escaping.

// customersInLine равен ["Barry", "Daniella"]

var customerProviders: [() ->String] = []

func collectCustomerProviders(\_customerProvider: @autoclosure @escaping () ->String) {

customerProviders.append(customerProvider)

}

collectCustomerProviders(customersInLine.remove(at: 0))

collectCustomerProviders(customersInLine.remove(at: 0))

print("Collected \(customerProviders.count) closures.")

// Выведет "Collected 2 closures."

for customerProviderincustomerProviders {

print("Now serving \(customerProvider())!")

}

// Выведет "Now serving Barry!"

// Выведет "Now serving Daniella!"

Вкодевыше, вместотого, чтобывызыватьпереданноезамыканиевкачествеаргумента customer, функция collectCustomerProviders(\_:) добавляетзамыканиекмассиву customerProviders. Массив объявлен за пределами функции, что означает, что замыкание в массиве может быть исполнено после того, как функция вернет значение. В результате значение аргумента customerProvider должен иметь “разрешение” на “побег” из зоны видимости функции.

**Резюме**

#### Объекты первого класса

Для начала укрепимся с мыслью, что в Swift функции являются носителями гордого статуса объектов первого класса. Это значит, что функцию можно хранить в переменной, передавать как параметр, возвращать в качестве результата работы другой функции. Вводится понятие «типа функции». Этот тип описывает не только тип возвращаемого значения, но и типы входных аргументов.  
Допустим, у нас есть две похожие функции, которые описывают две математические операции сложения и вычитания:

func add(op1: Double, op2: Double) ->Double {

return op1 + op2

}

func subtract(op1: Double, op2: Double) ->Double {

return op1 - op2

}

Их тип будет описываться следующим образом:

(Double, Double) ->Double

Прочесть это можно так: «Перед нами тип функции с двумя входными параметрами типа *Double* и возвращаемым значением типа *Double*.»  
Мы можем создать переменную такого типа:

*// Описываем переменную*

varoperation: (Double, Double) ->Double

*// Смело присваиваем этой переменной значение*

*// нужной нам функции, в зависимости от каких-либо условий:*

foriin0..<2 {

ifi == 0 {

operation = add

} else {

operation = subtract

}

let result = operation(1.0, 2.0) *// "Вызываем" переменную*

println(result)

}

Код, описанный выше, выведет в консоли:  
**3.0  
-1.0**

#### 1.2. Замыкания

Используем еще одну привилегию объекта первого класса. Возвращаясь к предыдущему примеру, мы могли бы создать такую новую функцию, которая бы принимала одну из наших старых функций типа *(Double, Double) ->Double* в качестве последнего параметра. Воттаконабудетвыглядеть:

*// (1)*

funcperformOperation(op1: Double, op2: Double, operation: (Double, Double) ->Double) ->Double { *// (2)*

returnoperation(op1, op2) *// (3)*

}

Разберем запутанный синтаксис на составляющие. Функция *performOperation* принимает три параметра:

* *op1* типа *Double* (op — сокращенное от «операнд»)
* *op2* типа *Double*
* *operation* типа *(Double, Double) ->Double*

В своем теле *performOperation* просто возвращает результат выполнения функции, хранимой в параметре *operation*, передавая в него первых два своих параметра.  
Пока что выглядит запутанно, и, возможно, даже не понятно. Немного терпения, господа.  
  
Давайте теперь передадим в качестве третьего аргумента не переменную, а анонимную функцию, заключив ее в фигурные *{}* скобки. Переданный таким образом параметр и будет называться **замыканием**:

letresult = performOperation(1.0, 2.0, {(op1:Double, op2: Double) ->Doublein

return op1 + op2 *// (5)*

}) *// (4)*

println(result) *// Выводит 3.0 вконсоли*

Отрывок кода *(op1:Double, op2: Double) ->Doublein* — это, так сказать, «заголовок» замыкания. Состоит он из:

* псевдонимов *op1*, *op2* типа *Double* для использования внутри замыкания
* возвращаемого значения замыкания *->Double*
* ключевого слова *in*

Еще раз о том, что сейчас произошло, по пунктам:  
**(1)** Объявлена функция *performOperation*  
**(2)** Эта функция принимает три параметра. Два первых — операнды. Последний — функция, которая будет выполнена над этими операндами.  
**(3)** *performOperation* возвращает результат выполнения операции.  
**(4)** В качестве последнего параметра в *performOperation* была передана функция, описанная замыканием.  
**(5)** В теле замыкания указывается, какая операция будет выполняться над операндами.

# Синтаксический сахар и неожиданные «плюшки»

Авторы Swift приложили немало усилий, чтобы пользователи языка могли писать как можно меньше кода и как можно больше тратить свое драгоценное время на  размышления об архитектуре проекта. Взяв за основу наш пример с арифметическими операциями, посмотрим, до какого состояния мы сможем его «раскрутить».

##### **2.1. Избавляемся от типов при вызове.**

Во-первых, можно не указывать типы входных параметров в замыкании явно, так как компилятор уже знает о них. Вызовфункциитеперьвыглядиттак:

performOperation(1.0, 2.0, {(op1, op2) ->Doublein

return op1 + op2

})

##### **2.2. Используем синтаксис «хвостового замыкания».**

Во-вторых, если замыкание передается в качестве последнего параметра в функцию, то синтаксис позволяет сократить запись, и код замыкания просто прикрепляется к хвосту вызова:

performOperation(1.0, 2.0) {(op1, op2) ->Doublein

return op1 + op2

}

##### **2.3. Не используем ключевое слово «return».**

Приятная (в некоторых случаях) особенность языка заключается в том, что если код замыкания умещается в одну строку, то результат выполнения этой строки автоматичеси будет возвращен. Таким образом ключевое слово «return» можно не писать:

performOperation(1.0, 2.0) {(op1, op2) ->Doublein

op1 + op2

}

##### **2.4. Используем стенографические имена для параметров.**

Идем дальше. Интересно, что Swift позволяет использовать так называемые стенографические (англ. shorthand) имена для входных параметров в замыкании. Т.е. каждому параметру по умолчанию присваивается псевдоним в формате **$n**, где *n* — порядковый номер параметра, начиная с нуля. Таким образом, нам, оказывается, даже не нужно придумывать имена для аргументов. В таком случае весь «заголовок» замыкания уже не несет в себе никакой смысловой нагрузки, и его можно опустить:

performOperation(1.0, 2.0) { $0 + $1 }

Согласитесь, эта запись уже совсем не похожа на ту, которая была в самом начале.

##### **2.5. операторные функции.**

Давайте посмотрим на предыдущую запись и зададимся вопросом, что уже знает компилятор о замыкании? Он знает количество параметров (*2*) и их типы (*Double* и *Double*). Знает тип возвращаемого значения (*Double*). Так как в коде замыкания выполняется всего одна строка, он знает, что ему нужно возвращать в качестве результата его выполнения. Можно ли упростить эту запись как-то еще?  
Оказывается, можно. Если замыкание работает только с двумя входными аргументами, в качестве замыкания разрешается передать операторную функцию, которая будет выполняться над этими аргументами (операндами). Теперь наш вызов будет выглядеть следующим образом:

performOperation(1.0, 2.0, +)

Теперь можно производить элементарные операции над нашими операндами в зависимости от некоторых условий, написав при этом минимум кода.  
  
Кстати, Swift также позволяет использовать операции сравнения в качестве операторной фуниции. Выглядетьэтобудетпримернотак:

funcperformComparisonOperation(op1: Double, op2: Double, operation: (Double, Double) ->Bool) ->Bool {

returnoperation(op1, op2)

}

println(performComparisonOperation(1.0, 1.0, >=)) *// Выведет "true"*

println(performComparisonOperation(1.0, 1.0, <)) *// Выведет "false"*

Илибитовыеоперации:

funcperformBitwiseOperation(op1: Bool, op2: Bool, operation: (Bool, Bool) ->Bool) ->Bool {

returnoperation(op1, op2)

}

println(performBitwiseOperation(true, true, ^)) *// Выведет "false"*

println(performBitwiseOperation(true, false, |)) *// Выведет "true"*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

##### **UPD.: Реальное применение**

Если вам нужно создать очередь с приоритетом, можно использовать двоичную кучу (binaryheap). Как известно, это может быть как MinHeap, так и MaxHeap, т.е. кучи, где в корне дерева находится минимальный или максимальный элемент соотвественно. Базовая реализация MinHeap от MaxHeap будет отличаться по сути только проверочными сравнениями при восстановлении инварианта двоичной кучи после добавления/удаления элемента.  
  
Таким образом, мы могли создать базовый класс *BinaryHeap*, который будет содержать свойство *comparison* типа *(T, T) ->Bool*. А конструктор этого класса будет принимать способ сравнения и затем использовать его в методах *heapify*. Прототип базового класса выглядел бы так:

classBinaryHeap<T: Comparable>: DebugPrintable{

privatevar array: Array<T?>

privatevar comparison: (T, T) ->Bool

privatevar used: Int = 0

*// Бла-бла-бла*

*// Internal Methods*

internalfuncremoveTop() ->T? { *//... }*

internalfuncgetTop() ->T? { *//... }*

*// Public Methods:*

funcaddValue(value: T) {

if used == self.array.count {

self.grow()

}

self.array[used] = value

heapifyToTop(used, comparison) *// Одно из мест, где используется функция сравнения*

self.used++

}

init(size newSize: Int, comparison newComparison: (T, T) ->Bool) {

array = [T?](count: newSize, repeatedValue: nil)

comparison = newComparison

}

}

Теперь для того, чтобы создать классы *MinHeap* и *MaxHeap* нам достаточно унаследоваться от *BinaryHeap*, а в их конструкторах просто явно указать, какое сравнение применять. Вот так будет выглядеть наши классы:

classMaxHeap<T: Comparable>: BinaryHeap<T>{

funcgetMax() ->T? {

returnself.getTop()

}

funcremoveMax() ->T? {

returnself.removeTop()

}

init(sizenewSize:Int) {

super.init(size: newSize, {$0> $1})

}

}

classMinHeap<T: Comparable>: BinaryHeap<T>{

funcgetMin() ->T? {

returnself.getTop()

}

funcremoveMin() ->T?{

returnself.removeTop()

}

init(size newSize: Int) {

super.init(size: newSize, {$0<= $1})

}

}

Задание для самостоятельного выполнения

Выполнить с учетом варианта

1. Создать массив из 5-ти элементов «бонусные акции», содержащий названия акций точки продаж(магазина, кафе и т.п.), отсортировать его, используя замыкания
2. Создать функцию с замыканием «увеличить бонус на», для каждой бонусной акции задать свое значение инкремента.
3. Создать класс «бонусные акции» с полями название и бонус, описать функции замыкания для начисления бонусов клиенту.
4. Изменить свой код так, чтобы при выборе любой из 5-ти акций изменялся накопленный бонус клиента в соответствие с выбранной акцией, акции и бонусы хранятся и обрабатываются с использованием класса и его методов, примените замыкание для захвата значения бонуса клиента.

Варианты

1. Магазин «Магнит»
2. Магазин «Перекресток»
3. Магазин «Marmalato»
4. Магазин «1000 и 1 бутылка»
5. Авиалинии, бонусы за километры
6. Служба «Такси»
7. Служба ремонта «Почини», скидки при последующих заказах
8. Кафе «У друзей»
9. Турфирма «Пегас»
10. Кинотеатр «Рассвет»