GeekBrains

Основы языка Swift

Методичка к уроку 2

Функции, опционалы, перечисления

Оглавление

Введение

На прошлой лекции мы познакомились с тем, что такое песочница, рассмотрели базовые операторы, ветвления и циклы.

На этой лекции вы:

- Узнаете про функции
- Научитесь объявлять различные функции, с параметрами и без, с возвращаемым значением и без
- Научитесь вызывать функции
- Узнаете про перечисления
- Научитесь использовать перечисления
- Узнаете про ассоциативные значения
- Узнаете про опционалы
- Научитесь распаковывать опционалы

Термины, используемые в лекции

Функция - это именованный кусок кода, который выполняет какую-либо задачу.

Тело функции - это действия, которые выполняет функция.

Функции

Функции - это именованные куски кода, которые выполняют какую-либо задачу. Для объявления функций в языке Swift используется ключевое слово “func”.

Функции без параметров и возвращаемого значения

Функции без параметров и возвращаемого значения имеют следующую конструкцию:

```
func <имя функции> {
    <дейтсвия>
}
```

Например, ниже представлен код функции, которая будет увеличивать в два раза переменную, объявленную ранее.

```
var a = 1
func multiplicationA() {
    a *= 2
}
```

Также действий может быть несколько, то есть здесь и a увеличивается в 2 раза, и b уменьшается на 1:

```
var a = 1
var b = 3
func multiplicationA() {
    a *= 2
    b -= 1
}
```

Вызов функции

Для вызова функции необходимо указать ее имя.

```
var a = 1
func multiplicationA() {
    a *= 2
}
```

```
multiplicationA() // Вызов функции
```

Функции с возвращаемым значением

Функции могут возвращать значения после выполнения действия, для этого нужно указать тип возвращаемого значения и после ключевого слова “return” само возвращаемое значение.

```
func <имя функции> -> <тип возвращаемого значения> {
    <действия>
    return <возвращаемое значение>
}
```

Полученное после выполнения функции значение можно, например, присвоить какой-нибудь переменной. Ниже представлена функция, которая должна вернуть целочисленное значение, в ходе выполнения этой функции a увеличивается в 2 раза и результат возвращается, как результат выполнения функции

```
var a = 1
func multiplicationA() -> Int {
    a *= 2
    return a
}
var c = multiplicationA() // Присваивание переменной с результата выполнения функции, то есть с будет равно 2 (a * 2 = 1 * 2 = 2)
```

🔥 Указание ключевого значения “return” для функций с возвращаемым значением обязательно, иначе выскочит ошибка

```
1 var a = 1
2 var b = 2
3 func multiplicationA() -> Int {
4     a *= 2
5     b -= 3
6 }
```

Missing return in a function expected to return 'Int'

💡 Если в теле функции выполняется только одно действие, то указывать “return” не нужно, будет возвращен результат действия.

```
var a = 1
func multiplicationA() -> Int {
    a * 2
}
let c = multiplicationA() // c будет равно 2
```

Функции с параметрами

В функции могут быть переданы разные параметры, для этого необходимо указать имя параметра и его тип:

```
func <имя функции>(<имя параметра>: <тип параметра>) {
    <тело функции>
}
```

В функцию может быть передано несколько параметров, в этом случае они перечисляются через запятую:

```
func <имя функции>(<имя параметра 1>: <тип параметра 1>, <имя параметра 1>: <тип параметра 2>, <имя параметра 3>: <тип параметра 3>) {
    <тело функции>
}
```

Также функция может одновременно и принимать параметры и возвращать значение.

```
func <имя функции>(<имя параметра>: <тип параметра>) -> <тип возвращаемого значения> {
    <тело функции>
    return <возвращаемое значение>
}
```

Функция ниже принимает на вход целочисленное значение и возвращает некое целочисленное значение. В ходе выполнения функции возвращается результат перемножения переданного значения и числа 2

```
func multiplication(value: Int) -> Int {
    value * 2 // Функция возвращает результат умножения переданного параметра на 2
}
```

Для вызова функции необходимо вызвать функцию и передать необходимые параметры:

```
func multiplication(value: Int) -> Int {
    value * 2 // Функция возвращает результат умножения переданного параметра на 2
}
let c = multiplication(value: 6) // c будет присвоен результат выполнения функции, а в качестве параметра будет передано число 6, таким образом c будет равно 6 * 2 = 12
let d = multiplication(value: c) // d будет присвоен результат выполнения функции, а в качестве параметра будет передана переменная c, равная 12, таким образом d будет равно 12 * 2 = 24
```

При вызове функции можно использовать ярлыки аргументов, в этом случае при вызове функции будет использоваться ярлык аргумента, а в самой функции имя параметра, в таком случае объявляемая функция имеет следующую конструкцию:

```
func <имя функции>(<ярлык аргумента> <имя параметра>: <тип параметра>) {
    <тело функции>
}
```

А при объявлении функции используется следующий синтаксис:

```
<имя функции>(<ярлык аргумента>: <передаваемое значение>)
```

```
func multiplication(with value: Int) -> Int { // with - ярлык аргумента, value - имя параметра
    value * 2 // в теле используется имя параметра
}

multiplication(with: 5) // при вызове используется ярлык аргумента
```

Если при вызове функции нет необходимости при вызове функции ни в имени переменной, ни в ярлыке аргумента, а из названия и так все понятно, то при объявлении функции вместо ярлыка аргумента необходимо указать “_”.

```
func multiplication(_ value: Int) -> Int {
    value * 2 // Функция возвращает результат умножения переданного параметра на 2
}
let c = multiplication(6) // c будет присвоен результат выполнения функции, а в качестве параметра будет передано число 6, таким образом c будет равно 6 * 2 = 12
let d = multiplication(c) // d будет присвоен результат выполнения функции, а в качестве параметра будет передана переменная c, равная 12, таким образом d будет равно 12 * 2 = 24
```

Функции с параметрами по умолчанию

При объявлении функции можно задать для параметра значение по умолчанию, в этом случае, если значение не будет передано, то будет использоваться дефолтное. Имеет следующую конструкцию:

```
func <имя функции>(<имя параметра>: <тип параметра> = <дефолтное значение>) {
    <тело функции>
}
```

```
func multiplication(value: Int = 5) -> Int {
    value * 2 // Функция возвращает результат умножения переданного параметра на 2
```

```
}

var a = multiplication(value: 10) // В качестве параметра передано 10, а будет равно 10 * 2 = 20
var b = multiplication() // В качестве параметра не передано ничего, значит будет использоваться значение по умолчанию, b будет равно 5 * 2 = 10
```

Inout

Если попытаться изменить переданный параметр внутри функции, то выскочит ошибка:

```
1 var a = 5
2 func multiplication(_ value: Int) {
3     value *= a    Left side of mutating operator isn't mutable: 'value' is a 'let' constant
4 }
```

Это потому, что переданное значение не может быть изменено. Чтобы изменять входное значение необходимо указать ключевое слово “inout”, имеет следующую конструкцию:

```
func <имя функции>(<имя параметра>: inout <тип параметра>) {
    <тело функции>
}
```

Вызов функции имеет следующую конструкцию:

```
<имя функциии>(<имя параметра>: &<параметр>)
```

Пример функции:

```
var a = 5
func multiplication(value: inout Int) {
    value *= a
}

var c = 7
multiplication(value: &c) // c станет равно 35 (7 * 5)
multiplication(value: &c) // c станет равно 175 (35 * 5)
```

Перегрузка функций

Перегрузка функций подразумевает, что может быть несколько функций с одинаковыми именами, но разными параметрами (их названием, количеством) или разным типом возвращаемого значения.

В коде ниже представлены 4 функции с одинаковыми именами “multiplication”:

```
// Функция 1
func multiplication(value: Int) -> Int {
    value * 2
}

// Функция 2
func multiplication(value: Int, newValue: Int) -> Int {
    value * 2
}

// Функция 3
func multiplication(value: Int) -> Double {
    Double(value) * 2
}

// Функция 4
func multiplication(value: Double) -> Double {
    value * 2
}
```

1. Функция 1 имеет имя “multiplication” и представляет собой функцию с одним входным значением и возвращаемым значением типа Int
2. Функция 2 так же имеет имя “multiplication”, но на вход принимаются уже два параметра
3. Функция 3 так же имеет имя “multiplication” и один параметр, но возвращает значение типа Double
4. Функция 4 так же имеет имя “multiplication”, но параметр и возвращаемое значение имеют тип Double

Вызвать функции можно следующим образом:

```
let c: Int = multiplication(value: 5)
let d = multiplication(value: 5, newValue: 4)
let e: Double = multiplication(value: 5)
let f = multiplication(value: 5.6)
```

Для c и e необходимо прописать их ожидаемый тип, иначе компилятор не поймет, какую из функций надо вызвать, функцию 1 или функцию 3.

🔥 Если существуют две функции с одинаковыми параметры, отличающиеся только типом возвращаемого значения - необходимо явно указать, какой тип ожидается.

```
1  func multiplication(value: Int) -> Int {
2      value * 2
3  }
4
5  func multiplication(value: Int) -> Double {
6      Double(value) * 2
7  }
8
9  let c = multiplication(value: 5)      ⊗ Ambiguous use of 'multiplication(value:)'
10 let d = multiplication(value: 5)      ⊗ Ambiguous use of 'multiplication(value:)'
```

В этом случае код должен выглядеть следующим образом:

```
func multiplication(value: Int) -> Int {
    value * 2
}

func multiplication(value: Int) -> Double {
    Double(value) * 2
}

let c: Int = multiplication(value: 5)
let d: Double = multiplication(value: 5)
```

Прекращение выполнения функции

Бывают случаи, когда необходимо прекратить выполнение функции после какого-то события, в этом случае на помощь приходит “return”. Например, есть функция, которая на вход принимает какое либо значение и если оно меньше 3, то ничего не делать и прекратить выполнение функции, а в противном случае напечатать в консоль число, которое было передано.

```
func multiplication(with value: Int) {
    if value < 3 {
        return
    } else {
        print(value)
    }
}

multiplication(with: 2) // Так как 2 меньше 3, то ничего не выполнится, выполнение функции будет прекращено
multiplication(with: 10) // // Так как 10 больше 3, то в консоль будет выведено 10
```

Несколько “return” в функции

Функция при разных условиях может вернуть разные значения, например, если параметр меньше 3, то вернуть параметр, умноженный на 2, если меньше 10, то умноженный на 4, во всех остальных случаях возвращает 0.

```
func multiplication(with value: Int) -> Int {
    if value < 3 {
        return value * 2
    } else if value < 10 {
        return value * 4
    } else {
        return 0
    }
}

let a = multiplication(with: 2) // Так как 2 меньше 3, функция вернет 2 * 2, а именно 4
let b = multiplication(with: 5) // Так как 5 меньше 10 и больше
```

```
3, функция вернет 5 * 4, а именно 20
let c = multiplication(with: 15) // // Так как 15 больше 10, функция вернет 0
```

Область видимости

Если написать функцию и попытаться вызвать переменную, объявленную в этой функции вне самой функции, то выскочит ошибка, что переменная не найдена.

```
1  func add(value: Int) -> Int{
2      let a = 7
3      return value + a
4  }
5
6  a = 8                                   Cannot find 'a' in scope
```

Это связано с областью видимости. По сути область видимости - это код, написанный между двух фигурных скобок. “Глобальная” область видимости - файлы до каких-либо скобок.

Областью видимости регулируется время жизни переменных и их доступность. Нельзя вызвать переменную вне ее области видимости.

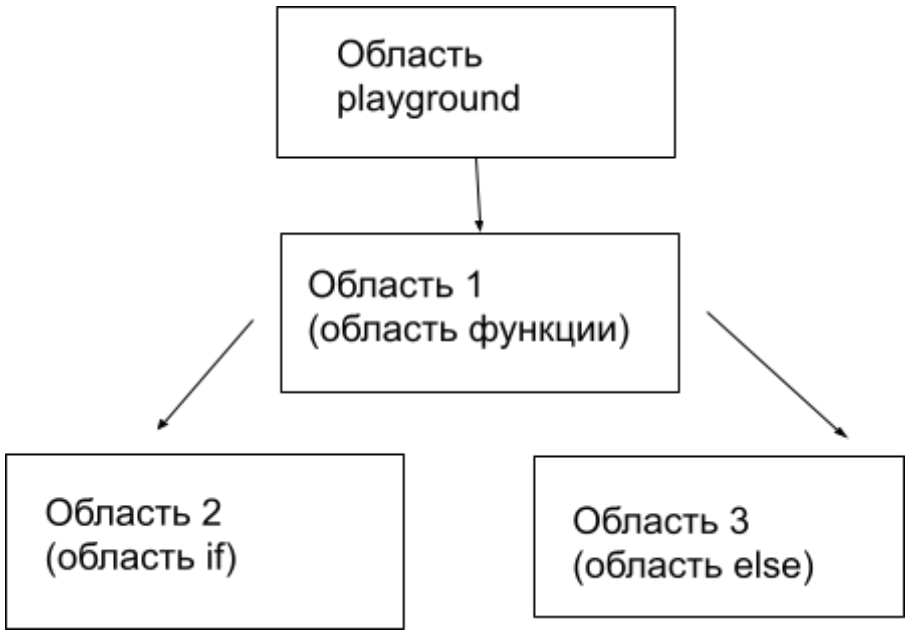
```
1  func add(value: Int) -> Int{
2      var a = 7                          1
3      if value < 5 {
4          let b = 4                      2
5          a += b
6      } else {
7          let c = 7                      3
8          a += c
9      }
10     return value + a
11 }
```

На рисунке приведены 3 области видимости, под номерами 1, 2 и 3. Области 2 и 3 являются вложенными по отношению к области 1, поэтому переменная a, объявленная в области 1 видима и может быть изменена в областях 2 и 3.

В области 2 объявлена переменная b, она видима и может быть использована только в области 2, так как область область 2 вложена в область 2 и на одинаковом уровне с областью 3.

Такая же ситуация и с переменной c в области 3.

Таким образом, если построить иерархию областей видимости из рисунка 3, то она будет выглядеть следующим образом:



Вложенные области могут видеть переменные из верхнеуровневых областей, но не наоборот.

Перечисления

Перечисление определяет общий тип для группы связанных значений и позволяет работать с этими значениями.

В языке Swift перечисления объявляются при помощи ключевого слова “enum” и представляют собой некий список значений. Имеют следующий синтаксис:

```
enum <Имя для перечислений> {
    case <первый кейс>
    case <второй кейс>
    case <третий кейс>
}
```

В коде задано перечисление под именем TestEnum, которое имеет три кейса: a, b, c

```
enum TestEnum {
    case a
    case b
    case c
}
```

Объявить переменную с кейсом из перечисления можно двумя способами

```
let latte = Coffee.latte
let espresso: Coffee = .espresso
```

Изменить значение в переменной можно также двумя способами

```
var coffee = Coffee.cappuccino
coffee = .latte
coffee = .espresso
```

Для примера будет рассмотрена следующая задача: необходимо написать функцию, которая в зависимости от вида кофе возвращает его стоимость. Виды кофе есть следующие: латте, капучино, эспрессо.

Для начала необходимо создать перечисление с возможными видами кофе

```
enum Coffee {
    case latte
    case cappuccino
    case espresso
}
```

Затем нужно объявить функцию, которая на вход принимает одно из значений перечисления, а возвращает стоимость кофе.

```
func getCost (coffee: Coffee) -> Double {
    switch coffee {
    case .latte: return 150
    case .cappuccino: return 200
    case .espresso: return 100
    }
}
```

В теле функции используется switch, чтобы описать действие для каждого типа кофе.

При вызове функции необходимо передать определенный тип кофе, передача значения происходит при помощи конструкции: .<кейс> или <имя перечисления>.<кейс>

```
print(getCost(coffee: Coffee.latte))
print(getCost(coffee: .cappuccino))
print(getCost(coffee: .espresso))
```

Перечисления с исходным значением

В перечислениях можно объявить кейсы с исходным(хранимым) значением. Такое перечисление имеет следующий синтаксис:

```
enum <имя перечисления>: <тип исходного значения> {
    case <первый кейс> = <исходное значение>
    case <второй кейс> = <исходное значение>
    case <третий кейс> = <исходное значение>
}
```

Например, перечислению с видами кофе будет присвоены названия по умолчанию с названиями кофе

```
enum Coffee: String {
    case latte = "Latte"
    case cappuccino = "Cappuccino"
    case espresso = "Espresso"
}
```

Таким образом, кейс “latte” будет хранить название “Latte”, кейс “espresso” название “Espresso” и так далее. Получить название можно следующим способом:

```
let coffeName = Coffee.latte.rawValue
```

Также можно написать функцию, которая будет возвращать названия переданного ей вида кофе, пример функции представлен ниже:

```
func getName(coffee: Coffee) -> String {
    coffee.rawValue
}
```

При вызове функции будет возвращено название напитка

```
print(getName(coffee: .latte)) // выведет в консоль "Latte"
print(getName(coffee: .cappuccino)) // выведет в консоль
"Cappuccino"
print(getName(coffee: .espresso)) // выведет в консоль
"Espresso"
```

Перечисления с исходным значением по умолчанию

Исходное значение для кейса можно задавать не всегда, например, если значение будет типа String, то есть не задать какое-либо значение, то по умолчанию будет хранится само название кейса

```
enum Coffee: String {
    case latte
    case cappuccino
    case espresso
}

func getName(coffee: Coffee) -> String {
    coffee.rawValue
}

print(getName(coffee: .latte)) // выведет в консоль "latte"
print(getName(coffee: .cappuccino)) // выведет в консоль
"cappuccino"
print(getName(coffee: .espresso)) // выведет в консоль
"espresso"
```

Если же тип задан Int, а само значения не задано, то по умолчанию будут возвращаться 0, 1, 2 и так далее в зависимости от количества кейсов и их порядка

```
enum TestCase: Int {
    case first
    case second
    case third
}

func getNumber(num: TestCase) -> Int {
    num.rawValue
}

print(getNumber(num: .first)) // Выведет в консоль 0
print(getNumber(num: .second)) // Выведет в консоль 1
print(getNumber(num: .third)) // Выведет в консоль 2
```

В одном перечислении могут быть для каких-то кейсов заданы значения, а для каких-то быть значения по умолчанию

```
enum Coffee: String {
    case latte
    case cappuccino = "Coffee"
    case espresso
}

func getName(coffee: Coffee) -> String {
    coffee.rawValue
}

print(getName(coffee: .latte)) // выведет в консоль "latte"
print(getName(coffee: .cappuccino)) // выведет в консоль
"Coffee"
print(getName(coffee: .espresso)) // выведет в консоль
"espresso"
```

Ассоциативные значения

Можно объявить перечисление, в котором каждый кейс будет хранить ассоциативное значение. В каждом кейсе может быть свой тип ассоциативного значения.

```
enum TestCase {
    case first(Int) // ассоциативное значение с типом Int
    case second(String) // ассоциативное значение с типом
String
}

var first = TestCase.first(7) // Инициализация с ассоциативным
значением 7
var second = TestCase.second("second") // Инициализация с
ассоциативным значением "second"

var third: TestCase = .first(3) // Инициализация с
ассоциативным значением 3
var fourth: TestCase = .second("fourth") // Инициализация с
ассоциативным значением "fourth"
```

```
func get(test: TestCase) { // Функция, которая в зависимости от полученного кейса получает ассоциативное значение и выполняет с ним действие
    switch test {
    case .first(let a): // в a хранится ассоциативное значение с типом Int для кейса first
        print(a * 5)
    case .second(let b): // в b хранится ассоциативное значение с типом String для кейса second
        print(b)
    }
}

get(test: first)
get(test: second)
```

Инициализация через исходное значение

Проинициализировать переменную в качестве кейса перечисления можно при помощи исходного значения, в этом случае используется следующая конструкция:

```
let/var <имя переменной> = <имя перечисления>(rawValue: <исходное значение по которому необходимо найти кейс>)
```

```
let latte = Coffee(rawValue: "latte") // Таким образом latte будет хранить значение .latte.
```

🔥 Если не будет найдено кейса по исходному значению, то в переменной будет хранится nil, так как такой инициализатор возвращает опционал, про которые будет рассказано ниже.

Опционалы

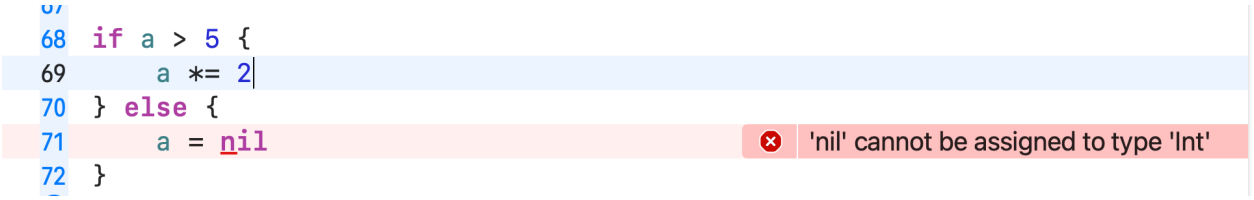
Опционал - тип данных, который может иметь значение, а может быть nil. То есть, например, в переменной может хранится значение 5, а может быть nil. “Под капотом” у опционалов перечисления с двумя кейсами: .some и .none

Optional используются, когда неизвестно будет значение или нет. Для объявления опционала используется “?”.

```
var a: Int? = 5 // Объявление опционала типа Int
// type(of:) позволяет узнать тип переменной
print(type(of: a)) // Выведет в консоль Optional<Int>
```

Например, необходимо изменить значение a. Если оно больше 5, то умножить его на 2, а если меньше - сделать nil.

Если значение будет не optional, подобное условие будет нельзя написать



А с опционалом такое условие написать можно

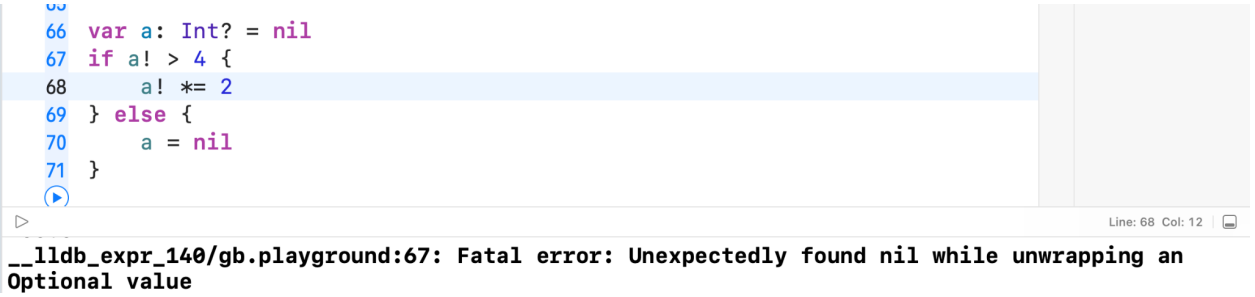
```
var a: Int? = 5
if a! > 5 {
    a! *= 2
} else {
    a = nil
}
```

Force unwrapping

Force unwrapping используется для распаковки опционалов, для распаковки таким способом используется “!”. Такой механизм стоит использовать только если есть 100% уверенность, что значение в переменной есть

```
var a: Int? = 5
if a! > 4 { // Принудительно распаковывается a, так как нельзя использовать арифметические операторы с optional, так как значения может не быть
    a! *= 2 // a станет равно 10
} else {
    a = nil
}
```

🔥 Необходимо использовать Force unwrapping с большой осторожностью, так как в процессе компиляции никаких ошибок не будет, а вот если в переменной, которая распаковывается при помощи такого механизма будет nil, то произойдет краш в рантайме



Implicity unwrapping

Implicity unwrapping - это механизм для распаковки опционалов, когда в процессе написания кода заранее известно, что значение будет, тогда можно заранее указать “!” для типа

```
var a: Int! = 5
print(type(of: a)) // Optional<Int>
```

Nil coalescing

Nil coalescing - это механизм для распаковки опционалов, когда переменной можно задать альтернативное значение, если она nil. Механизм имеет следующий синтаксис:

```
var/let <имя переменной> = <опционал> ?? <значение, которое будет присвоено, если опционал nil>
```

```
var a: Int? = 5
var b = a ?? 10 // Так как а имеет значение 5, то b станет равно 5
```

```
var a: Int? = nil
var b = a ?? 10 // Так как а не имеет значения, то b станет равно 10
```

```
var a: Int? = 5
if a ?? 0 > 4 { // если в а нет значения, то оно станет равно 0, 0 меньше 4, значит условие будет false
    a! *= 2
} else {
    a = nil
}
```

Optional binding

Optional binding - это механизм для распаковки опционалов с проверкой на наличие значения при помощи if или guard

If let

If let используется для распаковки опционала, имеет следующую конструкцию:

```
if let <имя для переменной, в которой будет распакованное значение, не опционал> = <опционал, который нужно распаковать>
{
   <действия с распакованным опционалом>
}
```

```
var a: Int? = 5
if let b = a { // Если в значении a есть значение, то оно будет присвоено b и будут выполнены действия после if
    print(b) // Так как в переменной a есть значение - функция будет выполнена и в консоль будет выведено 5
}
```

Также для переменной, в которой будет хранится распакованное значение можно использовать то же имя переменной, что и для опционал. В этом случае внутри if let переменная будет распакована, а вне - все еще будет опционалом.

```
if let a = a {
    print(a) // В консоль будет выведено 5
}
print(a) // В консоль будет выведено Optional(5)
```

> 💡 В Swift 5.7 добавили новый способ распаковки опционалом, теперь можно не использовать конструкцию if let <имя переменной> = <опционал>, достаточно написать if let <имя переменной>.

```
if let b = a { // До Swift 5.7
    print(b)
}

if let a = a { // До Swift 5.7
    print(a)
}

if let a { // Swift 5.7
    print(a)
}
```

Guard let

Guard let, так же как и if let, используется для распаковки опционала и имеет следующую конструкцию:

```
guard let <имя для переменной, в которой будет распакованное значение, не опционал> = <опционал, который нужно распаковать>
else {
    <действия, если распаковка не удалась>
    return
}
<действия, если распаковка не удалась>
```

```
guard let b = a else { // Так как в переменной a есть значение, будет выполнена функция print
    return // Не будет выполнено, так как в в переменной a есть значение
}
print(b) // Будет выполнено, так как в в переменной a есть значение
```

Так как if и guard очень похожи, но используются в разных случаях. Guard let стоит использовать, если условие является “отсекающим”, то есть, если оно не выполнено, то нет смысла выполнять следующие действия. If же используется, когда в процессе выполнения нужно проверить какое-либо условие, но если оно будет false, то продолжить выполнять дальнейшие действия.

🔥 Так как в guard используется return, в playground стоит использовать его внутри функции

```
if let a = a { // Если в a есть значение, то a будет выведено в консоль, если нет, то ничего страшного и выполнятся дальнейшие действия, то есть b будет уменьшено на 4
    print(a)
}
b -= 4

func test() {
    b -= 10
    guard let a = a else { // Если a не nil, то будут выполнены следующие действия и b уменьшится на 4, а если нет, то выполнение функции прекратится
```

```
        return
    }
    b -= 4
}
```

🔥 Если распакованное значение не используется внутри if или guard, то будет выведено предупреждение об этом, в таком случае лучше не распаковывать значение, а при помощи операторов сравнения проверить, не является ли оно nil (<имя переменной> != nil).

```
100 func test() {
101     b -= 10
102     guard let a = a else {
103         return
104     }
105     b -= 4
106 }
```

Value 'a' was defined but never used; consider replacing with boolean test

Replace 'let a = a' with 'a != nil'

Fix

Optional chaining

Optional chaining (или цепочка опционалов) - механизм, при котором последовательно вызываются несколько опционалов, при этом, как только в цепочке находится nil - проверка всех последующих значений прекращается и возвращается nil.

Что можно почитать еще?

1. SwiftBook. Функции
2. SwiftBook. Перечисления
3. https://otus.ru/journal/swift/

Используемая литература

1. https://docs.swift.org/swift-book/LanguageGuide/Functions.html
2. https://developer.apple.com/documentation/swift/optional
3. https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html