



Содержание

- ★ Функции (Functions) 
- ★ Опционалы (Optionals) 

Функции (Functions)

Функции — *это автономные фрагменты кода, выполняющие определенную задачу*. Вы даете *функции имя, которое определяет, что она делает*, и это имя используется для «**вызова**» функции для *выполнения ее задачи*, когда это необходимо

Каждая функция в Swift имеет тип, состоящий из типов **параметров функции** и типа **возвращаемого значения**. Вы можете использовать этот тип, как и любой другой тип в Swift, что упрощает передачу функций в качестве параметров другим функциям и возврат функций из функций

Функции также могут быть написаны внутри других функций, чтобы инкапсулировать полезную функциональность во вложенную область действия функции

Определение и вызов функций

- ❑ Когда вы определяете **функцию**, вы можете дополнительно определить одно или несколько именованных типизированных значений, которые функция принимает в качестве входных данных, известных как **параметры**. Вы также можете дополнительно определить тип значения, которое функция будет возвращать в качестве вывода, когда это будет сделано, известный как **тип возвращаемого значения**
- ❑ Каждая функция имеет **имя функции**, которое описывает задачу, которую выполняет функция. Чтобы использовать функцию, вы **«вызываете»** эту функцию по ее имени и передаете ей входные значения (известные как **аргументы**), которые соответствуют типам параметров функции.
- ❑ Аргументы функции *всегда должны указываться в том же порядке, что и список параметров функции*

Определение и вызов функций

Функция в приведенном ниже примере называется `greet(person:)`, потому что именно это она и делает — принимает имя человека в качестве входных данных и возвращает приветствие для этого человека.

Для этого вы определяете один входной параметр — `String` с именем `person` — и возвращаемый тип `String`, который будет содержать приветствие для этого человека

```
1 func greet(person: String) -> String {  
2     let greeting = "Hello, " + person + "!"  
3     return greeting  
4 }
```

Определение и вызов функций

- ❑ Вся эта информация собрана в определении функции, перед которым стоит ключевое слово **func**. Вы указываете возвращаемый функцией тип с помощью стрелки возврата -> (дефис, за которым следует правая угловая скобка), за которым следует **имя возвращаемого типа**
- ❑ Определение описывает, что делает функция, что она ожидает получить и что она возвращает после завершения. Определение упрощает однозначный вызов функции из любого места вашего кода

```
1 print(greet(person: "Anna"))
2 // Prints "Hello, Anna!"
3 print(greet(person: "Brian"))
4 // Prints "Hello, Brian!"
```

Определение и вызов функций

Чтобы сделать тело этой функции короче, вы можете объединить создание сообщения и оператор `return` в одну строку:

```
1  func greetAgain(person: String) -> String {  
2      return "Hello again, " + person + "!"  
3  }  
4  print(greetAgain(person: "Anna"))  
5  // Prints "Hello again, Anna!"
```

Параметры функции и возвращаемые значения

Параметры функций и возвращаемые значения чрезвычайно гибки в Swift. Вы можете определить что угодно, от простой служебной функции с одним безымянным параметром до сложной функции с выразительными именами параметров и различными опциями параметров.

Функции без параметров

Функции *не обязаны определять входные параметры.*

Вот функция без входных параметров, которая всегда возвращает одно и то же строковое сообщение при любом вызове:

```
1 func sayHelloWorld() -> String {  
2     return "hello, world"  
3 }  
4 print(sayHelloWorld())  
5 // Prints "hello, world"
```


Функции с несколькими параметрами

Функции могут иметь *несколько входных параметров*, которые записываются в круглых скобках функции, разделенных запятыми.

Эта функция принимает имя человека и информацию о том, приветствовали ли его уже в качестве входных данных, и возвращает соответствующее приветствие для этого человека:

```
1 func greet(person: String, alreadyGreeted: Bool) -> String {
2     if alreadyGreeted {
3         return greetAgain(person: person)
4     } else {
5         return greet(person: person)
6     }
7 }
8 print(greet(person: "Tim", alreadyGreeted: true))
9 // Prints "Hello again, Tim!"
```

Функции без возвращаемых значений

Функции *не обязаны определять тип возвращаемого значения*.

Вот версия функции `greet(person:)`, которая печатает собственное строковое значение, а не возвращает его:

```
1 func greet(person: String) {  
2     print("Hello, \(person)!")  
3 }  
4 greet(person: "Dave")  
5 // Prints "Hello, Dave!"
```

Функции с несколькими возвращаемыми значениями

Вы можете использовать тип **кортежа** (tuple) в качестве типа возвращаемого значения, чтобы **функция** возвращала несколько значений как часть одного составного возвращаемого значения.

В приведенном ниже примере определяется функция с именем **minMax(array:)**, которая находит наименьшее и наибольшее число в массиве значений **Int**:

```
1 func minMax(array: [Int]) -> (min: Int, max: Int) {  
2     var currentMin = array[0]  
3     var currentMax = array[0]  
4     for value in array[1..  
5         if value < currentMin {  
6             currentMin = value  
7         } else if value > currentMax {  
8             currentMax = value  
9         }  
10    }  
11    return (currentMin, currentMax)  
12 }
```

Функции с неявным возвратом

Если все тело **функции** представляет собой одно выражение, **функция неявно возвращает это выражение**.

Например, обе приведенные ниже функции ведут себя одинаково:

```
1  func greeting(for person: String) -> String {
2      "Hello, " + person + "!"
3  }
4  print(greeting(for: "Dave"))
5  // Prints "Hello, Dave!"
6
7  func anotherGreeting(for person: String) -> String {
8      return "Hello, " + person + "!"
9  }
10 print(anotherGreeting(for: "Dave"))
11 // Prints "Hello, Dave!"
```

Метки аргументов функций и названия параметров

- ❑ Каждый параметр функции имеет как *название аргумента*, так и *имя параметра*.
- ❑ *Название аргумента* используется при вызове функции; каждый аргумент записывается в вызове функции с названием аргумента перед ним. *Имя параметра* используется при реализации функции
- ❑ По умолчанию параметры используют имя своего параметра в качестве названия аргумента
- ❑ **Все параметры должны иметь уникальные имена.** Хотя несколько параметров могут иметь одну и то же название аргумента, уникальные названия аргументов помогают сделать ваш код более читабельным

```
1 func someFunction(firstParameterName: Int, secondParameterName: Int) {  
2     // In the function body, firstParameterName and secondParameterName  
3     // refer to the argument values for the first and second parameters.  
4 }  
5 someFunction(firstParameterName: 1, secondParameterName: 2)
```

Указание названий аргументов

Вы пишете **название аргумента** перед **именем параметра**, разделенную пробелом:

```
1 func someFunction(argumentLabel parameterName: Int) {  
2     // In the function body, parameterName refers to the argument value  
3     // for that parameter.  
4 }
```

Вот вариант функции **greet(person:)**, которая принимает имя человека и его родной город и возвращает приветствие:

```
1 func greet(person: String, from hometown: String) -> String {  
2     return "Hello \{(person)! Glad you could visit from \{(hometown)."  
3 }  
4 print(greet(person: "Bill", from: "Cupertino"))  
5 // Prints "Hello Bill! Glad you could visit from Cupertino."
```

Пропуск названий аргументов

Если вам не нужно название аргумента для параметра, **напишите знак подчеркивания (_)** вместо явного названия аргумента для этого параметра

```
1 func someFunction(_ firstParameterName: Int, secondParameterName: Int) {  
2     // In the function body, firstParameterName and secondParameterName  
3     // refer to the argument values for the first and second parameters.  
4 }  
5 someFunction(1, secondParameterName: 2)
```

Значения параметров по умолчанию

- ❑ Вы можете определить **значение по умолчанию** для любого параметра в функции, назначив значение параметру после типа этого параметра. *Если определено значение по умолчанию, вы можете опустить этот параметр при вызове функции.*
- ❑ Поместите параметры, не имеющие значений по умолчанию, в начало списка параметров функции перед параметрами, имеющими значения по умолчанию.
- ❑ Параметры, у которых нет значений по умолчанию, обычно более важны для смысла функции — их запись в первую очередь облегчает распознавание того, что вызывается одна и та же функция, независимо от того, пропущены ли какие-либо параметры по умолчанию.

```
1 func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int =  
  12) {  
2     // If you omit the second argument when calling this function, then  
3     // the value of parameterWithDefault is 12 inside the function body.  
4 }  
5 someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) //  
  parameterWithDefault is 6  
6 someFunction(parameterWithoutDefault: 4) // parameterWithDefault is 12
```


Вариативные параметры (Variadic parameters)

- ❑ **Вариативный параметр** принимает ноль или более значений указанного типа. Вы используете вариативный параметр, чтобы указать, что *параметру может быть передано различное число входных значений* при вызове функции
- ❑ Запишите **вариативные параметры**, вставив три символа точки (...) после имени типа параметра
- ❑ Значения, переданные **вариативному параметру**, становятся доступными в теле функции в виде **массива** соответствующего типа.
- ❑ Например, вариативный параметр с именем *numbers* и типом данных **Double...** доступен в теле функции в виде константы массива *numbers* типа **[Double]**

```
1 func arithmeticMean(_ numbers: Double...) -> Double {
2     var total: Double = 0
3     for number in numbers {
4         total += number
5     }
6     return total / Double(numbers.count)
7 }
8 arithmeticMean(1, 2, 3, 4, 5)
9 // returns 3.0, which is the arithmetic mean of these five numbers
10 arithmeticMean(3, 8.25, 18.75)
11 // returns 10.0, which is the arithmetic mean of these three numbers
```

Входные (In-Out) параметры

- ❑ Параметры функции являются константами по умолчанию.
- ❑ Попытка изменить значение параметра функции из тела этой функции приводит к ошибке времени компиляции.
- ❑ Это означает, что вы не можете изменить значение параметра по ошибке.
- ❑ Если вы хотите, чтобы *функция изменяла значение параметра, и вы хотите, чтобы эти изменения сохранялись после завершения вызова функции*, вместо этого определите этот параметр как **входной параметр**

Входные (In-Out) параметры

Вы пишете **входной параметр**, помещая ключевое слово **inout** прямо перед типом параметра.

Входной параметр имеет значение, которое передается функции, изменяется функцией и передается обратно из функции для замены исходного значения

```
3 func incrementNumber(_ number: inout Int) {  
4     number += 1  
5 }
```

Входные (In-Out) параметры

- ❑ Вы можете передать **только переменную** в качестве аргумента для входного параметра
- ❑ Вы не можете передать константу или литеральное значение в качестве аргумента, потому что константы и литералы не могут быть изменены
- ❑ Вы ставите амперсанд (&) *непосредственно перед именем переменной, когда передаете ее в качестве аргумента входному параметру*, чтобы указать, что она может быть изменена функцией

```
1 func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
2     let temporaryA = a  
3     a = b  
4     b = temporaryA  
5 }
```

```
1 var someInt = 3  
2 var anotherInt = 107  
3 swapTwoInts(&someInt, &anotherInt)  
4 print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")  
5 // Prints "someInt is now 107, and anotherInt is now 3"
```

Типы функций (Function Types)

- ❑ Каждая функция имеет определенный **тип функции**, состоящий из **типов параметров** и **типа возвращаемого значения** функции
- ❑ В этом примере определяются две простые математические функции с именами **addTwoInts** и **multiTwoInts**
- ❑ Каждая из этих функций принимает два значения **Int** и возвращает значение **Int**, являющееся результатом выполнения соответствующей математической операции
- ❑ Тип обеих этих функций **(Int, Int) -> Int**
- ❑ Это можно прочесть как: «Функция, которая имеет два параметра, оба типа **Int**, и которая возвращает значение типа **Int**»

```
1 func addTwoInts(_ a: Int, _ b: Int) -> Int {  
2     return a + b  
3 }  
4 func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {  
5     return a * b  
6 }
```

Типы функций (Function Types)

- ❑ Вот еще один пример для *функции без параметров или возвращаемого значения*
- ❑ Тип этой функции: **() -> Void**, или «функция, не имеющая параметров и возвращающая *Void*»

Использование типов функций

Вы используете **функциональные типы** точно так же, как и любые другие типы в Swift.

Например, вы можете определить константу или переменную как функцию и присвоить этой переменной соответствующую функцию:

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

Это можно прочесть как:

«Определите переменную с именем *mathFunction*, которая имеет тип «функции, которая принимает два значения *Int* и возвращает значение *Int*»

Установите эту новую переменную так, чтобы она ссылалась на функцию с именем *addTwoInts*»

Функция *addTwoInts*(_:_) имеет тот же тип, что и переменная *mathFunction*, поэтому это назначение разрешено средством проверки типов Swift

Типы функций как типы параметров

- ❑ Вы можете использовать тип функции, такой как **(Int, Int) -> Int**, в качестве типа параметра для другой функции.
- ❑ Это позволяет оставить некоторые аспекты реализации функции на усмотрение вызывающего объекта при вызове функции

```
1 func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int)
2 {
3     print("Result: \(mathFunction(a, b))")
4 }
5 printMathResult(addTwoInts, 3, 5)
6 // Prints "Result: 8"
```


Типы функций как возвращаемые типы

- ❑ Вы можете использовать **тип функции** в качестве возвращаемого типа другой функции
- ❑ Вы делаете это, записывая полный тип функции сразу после стрелки возврата (->) возвращаемой функции

Типы функций как возвращаемые типы

- ❑ В следующем примере определяются две простые функции с именами *stepForward(_:)* и *stepBackward(_:)*. Функция *stepForward(_:)* возвращает значение, на единицу превышающее ее входное значение, а функция *stepBackward(_:)* возвращает значение, на единицу меньшее, чем ее входное значение. Обе функции имеют тип **(Int) -> Int**
- ❑ Функция с именем *chooseStepFunction(backward:)*, возвращаемый тип которой **(Int) -> Int**
- ❑ Функция *chooseStepFunction(backward:)* возвращает функцию *stepForward(_:)* или функцию *stepBackward(_:)* на основе логического параметра **backward**

```
1 func stepForward(_ input: Int) -> Int {  
2     return input + 1  
3 }  
4 func stepBackward(_ input: Int) -> Int {  
5     return input - 1  
6 }
```

```
1 func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
2     return backward ? stepBackward : stepForward  
3 }
```

Closure (Замыкания)

- ❑ **Замыкания (Closure)** — это автономные блоки функций, которые можно передавать и использовать в вашем коде
- ❑ **Closure** - это анонимные функции (т.е функции у которых нету имен), их можно хранить в переменных и передавать как параметры в функциях
- ❑ **Closure** Swift имеют чистый, ясный стиль с оптимизацией, которая поощряет краткий, свободный от беспорядка синтаксис в распространенных сценариях. Эти оптимизации включают в себя:
 - a. Вывод типов параметров и возвращаемых значений из контекста
 - b. Неявные возвраты из замыканий с одним выражением
 - c. Сокращенные имена аргументов
 - d. Синтаксис закрытия замыкания

Синтаксис Closure

- ❑ Параметры в **замыканиях (closure)** могут быть входными (**in-out**) параметрами, но не могут иметь значения по умолчанию.
- ❑ Параметры **Variadic** можно использовать, если вы присвоите **имя** параметру Variadic.
- ❑ **Кортежи (tuples)** также можно использовать в качестве типов параметров и типов возвращаемых значений

```
{ (parameters) -> return type in  
  statements  
}
```

```
1 reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in  
2   return s1 > s2  
3 })
```

Определение типа из контекста

- Поскольку **замыкание** сортировки передается методу в качестве аргумента, Swift может сделать вывод о типах его параметров и типе возвращаемого значения
- Метод *sorted(by:)* вызывается для массива строк, поэтому его аргумент должен быть функцией типа **(String, String) -> Bool**
- Это означает, что типы **(String, String)** и **Bool** не нужно записывать как часть определения выражения замыкания
- Поскольку все типы могут быть выведены, стрелку возврата (->) и круглые скобки вокруг имен параметров также можно опустить

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

Неявные возвраты из замыканий с одним выражением

Замыкания с *одним выражением* могут неявно возвращать результат своего единственного выражения, опуская ключевое слово **return** в своем объявлении, как в этой версии предыдущего примера

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

Сокращенные имена аргументов

- ❑ Swift автоматически предоставляет **сокращенные имена аргументов** для встроенных замыканий, которые можно использовать для ссылки на значения аргументов замыкания по именам **\$0**, **\$1**, **\$2** и т. д.
- ❑ Если вы используете эти **сокращенные имена аргументов** в своем выражении замыкания, вы *можете опустить список аргументов замыкания в его определении*.
- ❑ Тип **сокращенных имен аргументов** выводится из ожидаемого типа функции, а используемый сокращенный аргумент с наибольшим номером определяет количество аргументов, которые принимает замыкание.
- ❑ Ключевое слово **in** также можно опустить, потому что замыкающее выражение полностью состоит из его тела

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

Завершающее замыкание (Trailing Closure)

- ❑ Если вам нужно передать выражение замыкания в функцию в качестве последнего аргумента функции, а выражение замыкания длинное, может быть полезно вместо этого записать его как **завершающее замыкание**
- ❑ Вы пишете **завершающее замыкание** *после круглых скобок вызова функции*, несмотря на то, что завершающее замыкание по-прежнему является аргументом функции
- ❑ Когда вы используете синтаксис **завершающего замыкания**, вы не записываете метку аргумента для первого замыкания как часть вызова функции
- ❑ Вызов функции может включать в себя несколько завершающих замыканий

```
1 func someFunctionThatTakesAClosure(closure: () -> Void) {  
2     // function body goes here  
3 }  
4  
5 // Here's how you call this function without using a trailing closure:  
6  
7 someFunctionThatTakesAClosure(closure: {  
8     // closure's body goes here  
9 })  
10  
11 // Here's how you call this function with a trailing closure instead:  
12  
13 someFunctionThatTakesAClosure() {  
14     // trailing closure's body goes here  
15 }
```


Опционалы (Optionals)

- ❑ Вы используете **Optionals** в ситуациях, когда **значение может отсутствовать**
- ❑ **Optional** представляет две возможности: либо есть значение, и вы можете развернуть его, чтобы получить доступ к этому значению, либо значения вообще нет
- ❑ Вот пример того, как можно использовать **optionals**, чтобы справиться с отсутствием значения. Тип Swift **Int** имеет инициализатор, который пытается преобразовать значение **String** в значение **Int**
- ❑ Однако не каждую строку можно преобразовать в целое число. Строка «123» может быть преобразована в числовое значение **123**, но строка «hello, world» не имеет очевидного числового значения для преобразования

```
1 let possibleNumber = "123"
2 let convertedNumber = Int(possibleNumber)
3 // convertedNumber is inferred to be of type "Int?", or "optional Int"
```

Nil

Вы устанавливаете **optional** переменную в состояние без значения, присваивая ей специальное значение **nil**

Если вы определяете **optional** переменную *без указания значения по умолчанию*, для нее автоматически устанавливается значение **nil**

```
1 var serverResponseCode: Int? = 404
2 // serverResponseCode contains an actual Int value of 404
3 serverResponseCode = nil
4 // serverResponseCode now contains no value
```

```
1 var surveyAnswer: String?
2 // surveyAnswer is automatically set to nil
```

Операторы if и Forced Unwrapping

- ❑ Вы можете использовать оператор `if`, чтобы узнать, содержит ли необязательный параметр значение, сравнивая необязательный параметр с `nil`
- ❑ Вы выполняете это сравнение с помощью оператора «равно» (`==`) или оператора «не равно» (`!=`)

```
1  if convertedNumber != nil {  
2      print("convertedNumber contains some integer value.")  
3  }  
4  // Prints "convertedNumber contains some integer value."
```

Операторы if и Forced Unwrapping

- ❑ Как только вы убедитесь, что необязательный элемент действительно содержит значение, вы можете получить доступ к его базовому значению, добавив восклицательный знак (!) в конце имени необязательного параметра.
- ❑ Восклицательный знак фактически говорит: «Я знаю, что этот необязательный элемент определенно имеет значение; пожалуйста, используйте его».
- ❑ Это известно как **принудительное разворачивание (force unwrapping)** опционального значения

```
1  if convertedNumber != nil {  
2      print("convertedNumber has an integer value of \(convertedNumber!).")  
3  }  
4  // Prints "convertedNumber has an integer value of 123."
```

Optional Binding (if let)

- ❑ Вы используете **optional binding**, чтобы узнать, содержит ли **optional** значение, и если да, то сделать это значение доступным как временную константу или переменную.
- ❑ **Optional binding** может использоваться с операторами **if** и **while** для проверки значения внутри optional и для извлечения значения в константу или переменную как часть одного действия

```
if let constantName = someOptional {  
    statements  
}
```

```
1 if let actualNumber = Int(possibleNumber) {  
2     print("The string \"\"(possibleNumber)\" has an integer value of \"  
   (actualNumber)\"")  
3 } else {  
4     print("The string \"\"(possibleNumber)\" couldn't be converted to an  
   integer")  
5 }  
6 // Prints "The string "123" has an integer value of 123"
```

Optional Binding (if let)

- ❑ Вы можете включить столько необязательных привязок и логических условий в один оператор `if`, сколько вам нужно, разделив их запятыми
- ❑ Если какое-либо из значений в необязательных привязках равно нулю или какое-либо логическое условие оценивается как ложное, все условие оператора `if` считается ложным
- ❑ Следующие операторы `if` эквивалентны

```
1  if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber <
    secondNumber && secondNumber < 100 {
2      print("\(firstNumber) < \(secondNumber) < 100")
3  }
4  // Prints "4 < 42 < 100"
5
6  if let firstNumber = Int("4") {
7      if let secondNumber = Int("42") {
8          if firstNumber < secondNumber && secondNumber < 100 {
9              print("\(firstNumber) < \(secondNumber) < 100")
10             }
11         }
12     }
13     // Prints "4 < 42 < 100"
```

Optional Binding (guard let)

Guard-let - выполняет аналогичную логику что и **if-let**, единственное отличие это в синтаксисе написания

```
3 let possibleNumber = "123"
4
5 guard let actualNumber = Int(possibleNumber) else {
6     print("The string \"\(possibleNumber)\" couldn't be converted to an
7         integer")
8     fatalError("Error")
9 }
10 print("The string \"\(possibleNumber)\" has an integer value of
    \(actualNumber)")
```

Nil-Coalescing Operator

- ❑ **Nil coalescing operator** (`a ?? b`) разворачивает необязательное `a`, если оно содержит значение, или возвращает значение по умолчанию `b`, если `a` равно `nil`
- ❑ Выражение `a` всегда имеет необязательный тип.
- ❑ Выражение `b` должно соответствовать типу, хранящемуся внутри `a`

```
1  let defaultColorName = "red"
2  var userDefinedColorName: String? // defaults to nil
3
4  var colorNameToUse = userDefinedColorName ?? defaultColorName
5  // userDefinedColorName is nil, so colorNameToUse is set to the default of
   "red"
```


Optional Chaining

- ❑ **Optional chaining** — это процесс запроса и вызова свойств, методов и индексов для необязательного параметра (**optional**), который в настоящее время может быть нулевым
- ❑ Если необязательный элемент содержит значение, вызов свойства, метода или индекса завершается успешно
- ❑ Если необязательное значение равно **nil**, вызов свойства, метода или индекса возвращает **nil**
- ❑ Несколько запросов могут быть объединены в цепочку, и вся цепочка корректно завершается ошибкой, если какое-либо звено в цепочке равно нулю

```
1 class Person {  
2     var residence: Residence?  
3 }  
4  
5 class Residence {  
6     var numberOfRooms = 1  
7 }
```

```
1 if let roomCount = john.residence?.numberOfRooms {  
2     print("John's residence has \(roomCount) room(s).")  
3 } else {  
4     print("Unable to retrieve the number of rooms.")  
5 }  
6 // Prints "John's residence has 1 room(s)."
```

Thanks for your attention!