## Функции и их параметры

Одним из строительных блоков программы являются функции. Функция определяет некоторое действие. В Kotlin функция объявляется с помощью ключевого слова **fun**, после которого идет название функции. Затем после названия в скобках указывается список параметров. Если функция возвращает какое-либо значение, то после списка параметров через запятую можно указать тип возвращаемого значения. И далее в фигурных скобках идет тело функции.

|  |  |
| --- | --- |
| 1  2  3 | fun имя\_функции (параметры) : возвращаемый\_тип{      выполняемые инструкции  } |

Параметры необязательны.

Например, определим и вызовем функцию, которая просто выводит некоторую строку на консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | fun main() {        hello() // вызов функции hello      hello() // вызов функции hello      hello() // вызов функции hello  }  // определение функции hello  fun hello(){      println("Hello")  } |

Функции можно определять в файле вне других функций или классов, сами по себе, как например, определяется функция main. Такие функции еще называют функциями верхнего уровня (top-level functions).

Здесь кроме главной функции main также определена функция hello, которая не принимает никаких параметров и ничего не возвращает. Она просто выводит строку на консоль.

Функция hello (и любая другая определенная функция, кроме main) сама по себе не выполняется. Чтобы ее выполнить, ее надо вызвать. Для вызова функции указывается ее имя (в данном случае "hello"), после которого идут пустые скобки.

Таким образом, если необходимо в разных частях программы выполнить одни и те же действия, то можно эти действия вынести в функцию, и затем вызывать эту функцию.

**Передача параметров**

Через параметры функция может получать некоторые значения извне. Параметры указываются после имени функции в скобках через запятую в формате имя\_параметра : тип\_параметра. Например, определим функцию, которая просто выводит сообшение на консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | fun main() {        showMessage("Hello Kotlin")      showMessage("Привет Kotlin")      showMessage("Salut Kotlin")  }    fun showMessage(message: String){      println(message)  } |

Функция showMessage() принимает один параметр типа String. Поэтому при вызове функции в скобках необходимо передать значение для этого параметра: showMessage("Hello Kotlin"). Причем это значение должно представлять тип String, то есть строку. Значения, которые передаются параметрам функции, еще назвают аргументами.

Консольный вывод программы:

Hello Kotlin

Привет Kotlin

Salut Kotlin

Другой пример - функция, которая выводит данные о пользователе на консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | fun main() {        displayUser("Tom", 23)      displayUser("Alice", 19)      displayUser("Kate", 25)  }  fun displayUser(name: String, age: Int){      println("Name: $name   Age: $age")  } |

Функция displayUser() принимает два параметра - name и age. При вызове функции в скобках ей передаются значения для этих параметров. При этом значения передаются параметрам по позиции и должны соответствовать параметрам по типу. Так как вначале идет параметр типа String, а потом параметр типа Int, то при вызове функции в скобках вначале передается строка, а потом число.

**Аргументы по умолчанию**

В примере выше при вызове функций showMessage и displayUser мы обязательно должны предоставить для каждого их параметра какое-то определенное значение, которое соответствует типу параметра. Мы не можем, к примеру, вызвать функцию displayUser, не передав ей аргументы для параметров, это будет ошибка:

|  |  |
| --- | --- |
| 1 | displayUser() |

Однако мы можем определить какие-то параметры функции как необязательные и установить для них значения по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | fun displayUser(name: String, age: Int = 18, position: String="unemployed"){      println("Name: $name   Age: $age  Position: $position")  }    fun main() {        displayUser("Tom", 23, "Manager")      displayUser("Alice", 21)      displayUser("Kate")  } |

В данном случае функция displayUser имеет три параметра для передачи имени, возраста и должности. Для первого параметр name значение по умолчанию не установлено, поэтому для него значение по-прежнему обязательно передавать значение. Два последующих - age и position являются необязательными, и для них установлено значение по умолчанию. Если для этих параметров не передаются значения, тогда параметры используют значения по умолчанию. Поэтому для этих параметров в принципе нам необязательно передавать аргументы. Но если для какого-то параметра определено значение по умолчанию, то для всех последующих параметров тоже должно быть установлено значение по умолчанию.

Консольный вывод программы

Name: Tom Age: 23 Position: Manager

Name: Alice Age: 21 Position: unemployed

Name: Kate Age: 18 Position: unemployed

**Именованные аргументы**

По умолчанию значения передаются параметрам по позиции: первое значение - первому параметру, второе значение - второму параметру и так далее. Однако, используя именованные аргументы, мы можем переопределить порядок их передачи параметрам:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | fun main() {        displayUser("Tom", position="Manager", age=28)      displayUser(age=21, name="Alice")      displayUser("Kate", position="Middle Developer")  } |

При вызове функции в скобках мы можем указать название параметра и с помощью знака равно передать ему нужное значение.

При этом, как видно из последнего случае, необязательно все аргументы передавать по имени. Часть аргументов могут передаваться параметрам по позиции. Но если какой-то аргумент передан по имени, то остальные аргументы после него также должны передаваться по имени соответствующих параметров.

Также если до обязательного параметра функции идут необязательные параметры, то для обязательного параметра значение передается по имени:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | fun displayUser(age: Int = 18, name: String){      println("Name: $name   Age: $age")  }  fun main() {        displayUser(name="Tom", age=28)      displayUser(name="Kate")  } |

**Изменение параметров**

По умолчанию все параметры функции равносильны val-переменным, поэтому их значение нельзя изменить. Например, в случае следующей функции при компиляции мы получим ошибку:

|  |  |
| --- | --- |
| 1  2  3  4 | fun double(n: Int){      n = n \* 2   // !Ошибка - значение параметра нельзя изменить      println("Значение в функции double: $n")  } |

Однако если параметр предствляет какой-то сложный объект, то можно изменять отдельные значения в этом объекте. Например, возьмем функцию, которая в качестве параметра принимает массив:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | fun double(numbers: IntArray){      numbers[0] = numbers[0] \* 2      println("Значение в функции double: ${numbers[0]}")  }    fun main() {        var nums = intArrayOf(4, 5, 6)      double(nums)      println("Значение в функции main: ${nums[0]}")  } |

Здесь функция double принимает числовой массив и увеличивает значение его первого элемента в два раза. Причем изменение элемента массива внутри функции приведет к тому, что также будет изменено значение элемента в том массиве, который передается в качестве аргумента в функцию, так как этот один и тот же массив. Консольный вывод:

Значение в функции double: 8

Значение в функции main: 8

## Переменное количество параметров. Vararg

Функция может принимать переменное количество параметров одного типа. Для определения таких параметров применяется ключевое слово **vararg**. Например, нам необходимо передать в функцию несколько строк, но сколько именно строк, мы точно не знаем. Их может быть пять, шесть, семь и т.д.:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | fun printStrings(vararg strings: String){      for(str in strings)          println(str)  }  fun main() {        printStrings("Tom", "Bob", "Sam")      printStrings("Kotlin", "JavaScript", "Java", "C#", "C++")  } |

Функция printStrings принимает неопределенное количество строк. В самой функции мы можем работать с параметром как с последовательностью строк, например, перебирать элементы последовательности в цикле и производить с ними некоторые действия.

При вызове функции мы можем ей передать любое количество строк.

Другой пример - подсчет суммы неопределенного количества чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | fun sum(vararg numbers: Int){      var result=0      for(n in numbers)          result += n      println("Сумма чисел равна $result")  }  fun main() {        sum(1, 2, 3, 4, 5)      sum(1, 2, 3, 4, 5, 6, 7, 8, 9)  } |

Если функция принимает несколько параметров, то обычно vararg-параметр является последним.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | fun printUserGroup(count:Int, vararg users: String){      println("Count: $count")      for(user in users)          println(user)  }    fun main() {        printUserGroup(3, "Tom", "Bob", "Alice")  } |

Однако это необязательно, но если после vararg-параметра идут еще какие-нибудь параметры, то при вызове функции значения этим параметрам передаются через именованные аргументы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | fun printUserGroup(group: String, vararg users: String, count:Int){      println("Group: $group")      println("Count: $count")      for(user in users)          println(user)  }  fun main() {        printUserGroup("KT-091", "Tom", "Bob", "Alice", count=3)  } |

Здесь функция printUserGroup принимает три параметра. Значения параметрам до vararg-параметра передаются по позициям. То есть в данном случае "KT-091" будет представлять значение для параметра group. Последующие значения интерпретируются как значения для vararg-параметра вплоть до именнованных аргументов.

**Оператор \***

Оператор **\*** (spread operator) (не стоит путать со знаком умножения) позволяет передать параметру в качестве значения элементы из массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | fun changeNumbers(vararg numbers: Int, koef: Int){      for(number in numbers)          println(number \* koef)  }  fun main() {        val nums = intArrayOf(1, 2, 3, 4)      changeNumbers(\*nums, koef=2)  } |

Обратите внимание на звездочку перед nums при вызове функции: changeNumbers(**\*nums**, koef=2). Без применения данного оператора мы столкнулись бы с ошибкой, поскольку параметры функции представляют не массив, а неопределенное количество значений типа Int.

## Возвращение результата. Оператор return

Функция может возвращать некоторый результат. В этом случае после списка параметров через двоеточие указывается возвращаемый тип. А в теле функции применяется оператор **return**, после которого указывается возвращаемое значение.

Например, определим функцию, которая возвращает сумму двух чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | fun sum(x:Int, y:Int): Int{        return x + y  }  fun main() {        val a = sum(4, 3)      val b = sum(5, 6)      val c = sum(6, 9)      println("a=$a  b=$b  c=$c")  } |

В объявлении функции sum после списка параметров через двоеточие указывается тип Int, который будет представлять тип возвращаемого значения:

|  |  |
| --- | --- |
| 1 | fun sum(x:Int, y:Int): Int |

В самой функции с помощью оператора return возвращаем полученное значение - результат операции сложения:

|  |  |
| --- | --- |
| 1 | return x + y |

Так как функция возвращает значение, то при ее вызове это значение можно присвоить переменной:

|  |  |
| --- | --- |
| 1 | val a = sum(4, 3) |

**Тип Unit**

Если функция не возвращает какого-либо результата, то фактически неявно она возвращает значение типа Unit. Этот тип аналогичен типу **void** в ряде языков программирования, которое указывает, что функция ничего не возвращает. Например, следующая функция

|  |  |
| --- | --- |
| 1  2  3 | fun hello(){      println("Hello")  } |

будет аналогична следующей:

|  |  |
| --- | --- |
| 1  2  3 | fun hello() : Unit{      println("Hello")  } |

Формально мы даже можем присвоить результат такой функции переменной:

|  |  |
| --- | --- |
| 1  2 | val d = hello()  val e = hello() |

Однако практического смысла это не имеет, так как возвращаемое значение представляет объект Unit, который больше никак не применяется.

Если функция возвращает значение Unit, мы также можем использовать оператор return для возврата из функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | fun checkAge(age: Int){      if(age < 0 || age > 110){          println("Invalid age")          return      }      println("Age is valid")  }  fun main() {        checkAge(-10)      checkAge(10)  } |

В данном случае если значение параметра age выходит за пределы диапазона от 0 до 110, то с помощью оператора return осуществляется выход из функции, и последующие инструкции не выполняются. При этом если функция возвращает значение Unit, то после оператора return можно не указывать никакого значения.

## Однострочные и локальные функции

**Однострочные функции**

Однострочные функции (single expression function) используют сокращенный синтаксис определения функции в виде одного выражения. Эта форма позволяет опустить возвращаемый тип и оператор return.

|  |  |
| --- | --- |
| 1 | fun имя\_функции (параметры\_функции) = тело\_функции |

Функция также определяется с помощью ключевого слова **fun**, после которого идет имя функции и список параметров. Но после списка параметров не указывается возвращаемый тип. Возвращаемый тип будет выводится компилятором. Далее через оператор присвоения = определяется тело функции в виде одного выражения.

Например, функция возведения числа в квадрат:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | fun square(x: Int) = x \* x    fun main() {        val a = square(5)   // 25      val b = square(6)   // 36      println("a=$a  b=$b")  } |

В данном случае функция square возводит число в квадрат. Она состоит из одного выражения x \* x. Значение этого выражения и будет возвращаться функцией. При этом оператор return не используется.

Такие функции более лаконичны, более читабельны, но также опционально можно и указывать возвращаемый тип явно:

|  |  |
| --- | --- |
| 1 | fun square(x: Int) : Int = x \* x |

**Локальные функции**

Одни функции могут быть определены внутри других функций. Внутренние или вложенные функции еще называют локальными.

Локальные функции могут определять действия, которые используются только в рамках какой-то конкретной функции и нигде больше не применяются.

Например, у нас есть функция, которая сравнивает два возраста:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | fun compareAge(age1: Int, age2: Int){        fun ageIsValid(age: Int): Boolean{          return age > 0 && age < 111      }      if( !ageIsValid(age1) || !ageIsValid(age2)) {          println("Invalid age")          return      }        when {          age1 == age2 -> println("age1 == age2")          age1 > age2 -> println("age1 > age2")          age1 < age2 -> println("age1 < age2")      }  }  fun main() {        compareAge(20, 23)      compareAge(-3, 20)      compareAge(34, 134)      compareAge(15, 8)  } |

Однако извне могут быть переданы некорректные данные. Имеет ли смысл сравнивать возраст меньше нуля с другим? Очевидно нет. Для этой цели в функции определена локальная функция ageIsValid(), которая возвращает true, если возраст является допустимым. Больше в программе эта функция нигде не используется, поэтому ее можно сделать локальной.

При этом локальная может использоваться только в той функции, где она определена.

Причем в данном случае удобнее сделать локальную функцию однострочной:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | fun compareAge(age1: Int, age2: Int){        fun ageIsValid(age: Int)= age > 0 && age < 111        if( !ageIsValid(age1) || !ageIsValid(age2)) {          println("Invalid age")          return      }        when {          age1 == age2 -> println("age1 == age2")          age1 > age2 -> println("age1 > age2")          age1 < age2 -> println("age1 < age2")      }  } |

## Перегрузка функций

Перегрузка функций (function overloading) представляет определение нескольких функций с одним и тем же именем, но с различными параметрами. Параметры перегруженных функций могут отличаться по количеству, типу или по порядку в списке параметров.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | fun sum(a: Int, b: Int) : Int{      return a + b  }  fun sum(a: Double, b: Double) : Double{      return a + b  }  fun sum(a: Int, b: Int, c: Int) : Int{      return a + b + c  }  fun sum(a: Int, b: Double) : Double{      return a + b  }  fun sum(a: Double, b: Int) : Double{      return a + b  } |

В данном случае для одной функции sum() определено пять перегруженных версий. Каждая из версий отличается либо по типу, либо количеству, либо по порядку параметров. При вызове функции sum компилятор в зависимости от типа и количества параметров сможет выбрать для выполнения нужную версию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | fun main() {        val a = sum(1, 2)      val b = sum(1.5, 2.5)      val c = sum(1, 2, 3)      val d = sum(2, 1.5)      val e = sum(1.5, 2)  } |

При этом при перегрузке не учитывает возвращаемый результат функции. Например, пусть у нас будут две следующие версии функции sum:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | fun sum(a: Double, b: Int) : Double{      return a + b  }  fun sum(a: Double, b: Int) : String{      return "$a + $b"  } |

Они совпадают во всем за исключением возвращаемого типа. Однако в данном случае мы сталкивамся с ошибкой, так как перегруженные версии должны отличаться именно по типу, порядку или количеству параметров. Отличие в возвращаемом типе не имеют значения.