1.2. Функции в Kotlin

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 1.2. Функции в Kotlin

Напечатано:: Павел Степанов
Дата: понедельник, 9 октября 2023, 11:03

Оглавление

1.2.1. Объявление функции. Аргументы функции. Вызов функции

В Kotlin функции объявляются с помощью ключевого слова fun. После слова fun указывается произвольное имя функции, затем, в скобках - перечень параметров функции. Если функция должна возвращать значение, то тип возвращаемого значения должен указываться после списка параметров через двоеточие.

```
fun double(x: Int): Int {
...
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Применение функций

При вызове функции используется традиционный подход, т.е. указывается имя функции и в скобках перечисляются фактические параметры.

```
val result = double(2)
```

Open in Playground → Target: JVM Running on v.1.9.10

Если функция является частью класса (методом), то для её вызова используется знак точки после имени объекта.

```
Sample().foo() //создаёт экземпляр класса Sample и вызывает foo
```

Open in Playground → Target: JVM Running on v.1.9.10

Инфиксная запись

Инфиксной называется запись, в которой имя функции указывается между параметрами функции (по аналогии со знаками арифметических операций).

Инфиксная функция формально принимает два параметра. Первый параметр представляет объект, который вызывает функцию. А второй параметр - данные, которые непосредственно будут передаваться функции при ее вызове.

Инфиксная функция определяется только внутри класса. Для определения инфиксной функции в ее заголовке указывается ключевое слово infix.

Функции могут быть вызваны при помощи инфиксной записи, при условии, что:

- Они являются членом другой функции или расширения
- В них используется один параметр
- Когда они помечены ключевым словом infix

```
// Определяем выражение как Int
infix fun Int.shl(x: Int): Int {
...
}
// вызываем функцию, используя инфиксную запись
1 shl 2
// то же самое, что
1.shl(2)
```

Open in Playground → Target: JVM Running on v.1.9.10

Параметры

Параметры функции записываются аналогично системе обозначений в языке Pascal, имя:тип. Параметры разделены запятыми. Каждый параметр должен быть явно указан.

```
fun powerOf(number: Int, exponent: Int) {
...
}
```

Open in Playground → Target: JVM Running on v.1.9.10

1.2.2. Аргументы по умолчанию

Параметры функции могут иметь значения по умолчанию, которые используются в случае, если аргумент функции не указан при её вызове. Это позволяет снизить уровень перегруженности кода по сравнению с другими языками.

```
fun read(b: Array, off: Int = 0, len: Int = b.size()) {
...
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Значения по умолчанию указываются после типа знаком =. Переопределённые методы всегда используют те же самые значения по умолчанию, что и их базовые методы. При переопределении методов со значениями по умолчанию эти параметры должны быть опущены:

```
open class A {
    open fun foo(i: Int = 10) { ... }
}
class B : A() {
    override fun foo(i: Int) { ... }  // значение по умолчанию указать нельзя
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Если параметр со значением по умолчанию ставится перед обычным параметром, то значение по умолчанию можно использовать только при вызове функции с именованными аргументами (см. ниже):

```
fun foo(bar: Int = 0, baz: Int) { ... }
foo(baz = 1) // Используется значение по умолчанию bar = 0
```

Open in Playground → Target: JVM Running on v.1.9.10

Но если последний аргумент - это lambda-функция (см. следующие разделы), то передача значений параметров по умолчанию не допускается:

```
fun foo(bar: Int = 0, baz: Int = 1, qux: () -> Unit) { ... }
foo(1) { println("hello") } // Использует значение по умолчанию baz = 1
foo { println("hello") }    // Использует два значения по умолчанию bar = 0 и baz = 1
```

Open in Playground → Target: JVM Running on v.1.9.10

Именованные аргументы

Имена параметров могут быть явно указаны при вызове функций. Это очень удобно, когда у функции большой список параметров, в том числе со значениями по умолчанию. Рассмотрим следующую функцию с большим количеством параметров:

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
...
}
```

Open in Playground → Target: JVM Running on v.1.9.10

мы можем вызвать её, используя аргументы по умолчанию

```
reformat(str)
```

Open in Playground → Target: JVM Running on v.1.9.10

Однако, при вызове этой функции без аргументов по умолчанию, получится что-то вроде

```
reformat(str, true, true, false, '_')
```

Open in Playground → Target: JVM Running on v.1.9.10

С помощью именованных аргументов мы можем сделать код более читабельным:

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
```

```
)
```

Или, если нам не нужны все эти аргументы

Open in Playground → Target: JVM Running on v.1.9.10

```
reformat(str, wordSeparator = '_')
```

Open in Playground → Target: JVM Running on v.1.9.10

При вызове функции как с позиционными, так и с именованными аргументами все позиционные аргументы должны располагаться перед первым именованным аргументом. Например, вызов `f(1, y = 2)` разрешен, а `f(x = 1, 2)` - нет.

Функции можно передать переменное число аргументов. Для этого используется служебное слово vararg перед именем аргумента. В случае использования vararg все аргументы собираются в массив (Array), который можно обработать управляющими конструкциями:

```
fun foo(vararg strings: String) { ... }
foo(strings = *arrayOf("a", "b", "c"))
```

Open in Playground → Target: JVM Running on v.1.9.10

Обратите внимание, что синтаксис именованных аргументов не может быть использован при вызове Java функций, потому как байт-код Java не всегда сохраняет имена параметров функции.

Функции с возвращаемым типом Unit

Если функция не возвращает никакого полезного значения, её возвращаемый тип - Unit. Unit - тип только с одним значением - Unit. Это возвращаемое значение не нуждается в явном указании

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // return Unit или return необязательны
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Указание типа Unit в качестве возвращаемого значения тоже не является обязательным. Код, написанный выше, совершенно идентичен с

```
fun printHello(name: String?) {
    ...
}
```

Open in Playground → Target: JVM Running on v.1.9.10

1.2.3. Однострочные и локальные функции

Однострочные функции

Если функция состоит из возвращения значения некоторого выражения, т.е тело функции - это одна строчка, то такие функции можно записать коротко. В короткой записи можно не писать тип возвращаемого значения и после списка параметров после знака равенства указать искомое выражение.

Например, функция:

```
fun sum(a:Int, b:Int) = a + b
```

Open in Playground → Target: JVM Running on v.1.9.10

идентична функции

```
fun sum(a:Int, b:Int):Int{
return a + b
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Однострочные функции лаконичнее и читабельнее.

Локальные функции

Если функция определена внутри другой функции (вложена), то такую функцию называют локальной. Kotlin допускает такие функции.

При использовании локальной функции следует учитывать. что на нее распространяются общие правила видимости, что означает, что локальная функция видима только внутри той функции, в которой объявлена.

Рассмотрим пример. Пусть функция принимает на вход координаты трех точек и должна вычислить периметр треугольника, построенного на этих трех точках. Для простоты будем считать, что заданные точки образуют треугольник.

```
import java.util.*
fun perimetr(x1:Double, y1:Double, x2:Double, y2:Double, x3:Double, y3:Double):Double{
    fun length(x1:Double, y1:Double, x2:Double, y2:Double) = Math.sqrt(Math.pow(x1-x2, 2.0) + Math.pow(y1 - y2, 2.0))
    return length(x1, y1, x2, y2) + length(x2, y2, x3, y3) + length(x1, y1, x3, y3)
}
fun main() {
    val input = Scanner(System.in)
    var x1 = input.nextDouble()
    var y1 = input.nextDouble()
    var x2 = input.nextDouble()
    var y2 = input.nextDouble()
    var x3 = input.nextDouble()
    var y3 = input.nextDouble()
    println(perimetr(x1, y1, x2, y2, x3, y3))
}
```

Open in Playground → Target: JVM Running on v.1.9.10

В данном примере функция поиска длины отрезка по координатам концов - локальная. Попытка вызвать функцию length за пределами функции perimetr потерпит неудачу.

1.2.4. Анонимные функции

Перед тем как рассматривать анонимные функции нужно разобраться с таким понятием, как *лямбда-выражение* и как лямбда-выражения используются в Котлин.

Лямбда-выражения

Лямбда-выражение – это функция, записанная в виде выражения, и которую можно передавать как аргумент в другие функции.

Фактически лямбды представляют сокращенную запись функций. При этом лямбды могут передаваться в качестве параметра в функции.

Лямбда-выражения оборачиваются в фигурные скобки: `{println("Hello")}` и может быть как присвоена переменной, так и передана аргументом в другую функцию. Рассмотрим пример с присвоением лямбда-выражения переменной

```
val hello = {println("Hello")} //присваиваем лямбда-выражение переменной
hello() //обращаемся к переменной как к функции
```

Open in Playground → Target: JVM Running on v.1.9.10

Лямбда-выражение, как и обычная функция может иметь входные параметры и возвращаемое значение. Входные параметры пишутся сразу после открывающей фигурной скобки, а результат после стрелки. Например, напишем функцию сложения двух чисел как лямбда.

```
val sum = {a:Int, b:Int -> a + b} //присваиваем лямбда-выражение переменной
sum(5, 8) //обращаемся к переменной как к функции
```

Open in Playground → Target: JVM Running on v.1.9.10

При описании лямбда выражения можно не ограничиваться только одной командой. Такая лямбда может иметь тело, в котором возвращаемым значением считается последнее написанное.

Например:

```
val sum = {a:Int, b:Int -> a + b} //присваиваем лямбда-выражение переменной
sum(5, 8) //обращаемся к переменной как к функции
```

Open in Playground → Target: JVM Running on v.1.9.10

Лямбда может содержать несколько команд. В таком случае результатом считается значение последнего выражения. Например:

```
fun main() {
    var res = {x:Int, y:Int ->
                var result = 2 * x + y
                result //возвращаемое из лямбда значение
            }
    println(res(12, 15))
}
```

Open in Playground → Target: JVM Running on v.1.9.10

Важно: если лямбда-выражение содержит несколько команд, то каждая пишется с новой строки. Также важно помнить, что слово return в лямбда-выражениях не применяется. Возвращается всегда результат последней команды.

Перед тем, как рассмотреть пример, в котором лямбда передается в функцию как параметр, определим понятие типа лямбда-выражения.

Если в лямбда выражении убрать тело, имена параметров и оставить только типы параметров и тип возвращаемого значения, то мы получим запись, которая и называется типом лямбда-выражения. Например, типом лямбда-выражений из предыдущих примеров будет: `(Int, Int) -> Int`. Таким образом в типе лямбда-выражения мы указываем какие типы параметров и в каком порядке принимает лямбда и какой тип возвращает как результат.

Если лямбда-выражение не возвращает значения, то типом будет `Unit`, например: `(Int, Int) - > Unit`, что означает, что параметрами являются два целых числа, а результат не возвращается.

Для передачи лямбда-выражения в функцию, необходимо определить у функции параметр, тип которого соответствует типу лямбда-выражения. Такие функции (принимающие параметрами лямбда-выражения) в Котлин называются функциями высших порядков. Рассмотрим пример:

```
fun action (n1: Int, n2: Int, operation: (Int, Int)-> Int){
    val result = operation(n1, n2)
    println(result)
}
fun main(){
```

```
    action(4, 5, {x:Int, y:Int -> x * y})
    var sum = {x:Int, y:Int -> x + y}
    action(6, 7, sum)
}
```

Open in Playground → Target: JVM Running on v.1.9.10

В данном примере функция action третьим параметром может получить любую функцию, соответствующую шаблону (типу) `(Int, Int) Int`. Из тела функции видно, что полученная под именем operation функция применяется к первым параметрам action. При вызове action можно как непосредственно написать лямбда-выражение третьим параметром, так и определить его заранее и передать функции имя соответствующей переменной.

Анонимные функции

Анонимные функции выглядят также, как и обычные за одним исключением: они не имеют имени. Анонимные функции по смыслу похожи на лямбда-выражение и применяются практически также, но есть важно отличие: анонимные функции могут использовать `return` и возвращать результат из любого блока своего кода. Рассмотрим пример:

```
fun main() {
    operation(9,5, fun(x: Int, y: Int): Int { return x + y })   // анонимная функция как параметр
    operation(9,5, fun(x: Int, y: Int): Int = x - y)
}
fun operation(x: Int, y: Int, op: (Int, Int) ->Int){
    val result = op(x, y)
    println(result)
}
```

Open in Playground → Target: JVM Running on v.1.9.10

1.2.5. Функциональные типы

Вернемся еще раз к типам лямбда-выражений. В Котлин структуры вида `(Int, Int) -> Double` используются довольно часто и называются *функциональными выражениями*.

Kotlin использует семейство функциональных типов, таких как `(Int) -> String`, для объявлений, которые являются частью функций: val onClick: () -> Unit = ...

- Эти типы имеют специальные обозначения, которые соответствуют сигнатурам функций, то есть их параметрам и возвращаемым значениям:
- У всех функциональных типов есть список с типами параметров и возвращаемый тип. Например: `(A, B) -> C` обозначает тип, который предоставляет функции два принятых аргумента типа A и B, а также возвращает значение типа C. Список с типами параметров может быть пустым, как, например, в () -> A. Возвращаемый тип Unit (необходимый, если функция ничего не возвращает) не может быть опущен.
- У функциональных типов может быть дополнительный тип - получатель (receiver), который указывается в объявлении перед точкой: тип A.(B) -> C описывает функции, которые могут быть вызваны для объекта-получателя A с параметром B и возвращаемым значением C. Литералы функций с объектом-приёмником часто используются вместе с этими типами.

Объявление функционального типа также может включать именованные параметры: (x: Int, y: Int) -> Point. Именованные параметры могут быть использованы для описания смысла каждого из параметров.