Лабораторная работа 2. Функции

Функции

Функция – это фрагмент кода, который можно вызывать из других частей программы. Обычно это логически самодостаточный код, имеющий минимум связей с другими частями кода.

Пример функции, выводящей на экран приветствие:

```
fun hello(name: String) {
    println("Здравствуйте, $name!")
}
```

Определение функции начинается с ключевого слова fun (от англ. *function* – функция).

Затем следует имя функции – оно строится по тем же законам, что и имя переменной, может содержать буквы, цифры и знак подчёркивания, но начинаться с цифры не может.

После имени функции в круглых скобках указывается список аргументов – переменные, которые должны быть переданы в функцию при вызове. Если переменных несколько, то они перечисляются через запятую. Если аргументов нет, то круглые скобки после имени функции всё равно должны быть, просто в них ничего не будет.

Наконец, после списка аргументов в фигурных скобках идёт фрагмент кода, который и будет выполняться после вызова функции.

Вызвать функцию можно в том месте кода, где требуется выполнить действие:

```
fun main() {
    hello("Иван")
    hello("Мария")
}
```

Здесь функция вызывается два раза подряд, и в консоли появится две строки: «Здравствуйте, Иван!» и «Здравствуйте, Мария!».

Как правило, функции располагаются в коде на одном уровне, то есть следуют друг за другом:

```
fun hello() {
    ...
}

fun main() {
    ...
}
```

Компилятор допускает расположение одной функции внутри другой, но делать это следует только при необходимости.

Возврат значения

Функция может не просто выполнять действия, но и возвращать значение в вызвавший её код. Например, следующая функция вычисляет сумму ряда чисел от 1 до заданного значения:

```
fun sum(n: Int): Int {
    var result = 0
    for (i in 1..n)
        result += i
    return result
```

```
}
```

Первое изменение – это указание типа возвращаемого результата, оно указывается после круглых скобок с аргументами и двоеточия. Это может быть любой тип данных, в данном случае `Int`.

Второе изменение – это оператор возврата результата `return`. Этот оператор завершает работу функции и возвращает управление в вызвавший функцию код. После оператора return указывается переменная (или просто значение), которое и будет возвращено.

Вызывается такая функция точно так же, как и любая другая функция: её значение можно присвоить переменной или просто вывести на экран:

```
val r = sum(5) + sum(10)
println(sum(15))
```

В функции может быть несколько операторов `return`:

```
fun getColor(n: Int): Int {
    if (n == 1)
        return "Красный"
    if (n == 2)
        return "Синий"
    if (n == 3)
        return "Зеленый"
    return "Неизвестно"
}
```

Однако по возможности рекомендуется избегать такого кода: желательно, чтобы точка выхода из функции была одна. Это облегчает отладку функции и улучшает понимание кода. Например, функцию выше можно было бы переписать следующим образом:

```
fun getColor(n: Int): Int {
    val color = when (n) {
        1 -> "Красный"
        2 -> "Синий"
        3 -> "Зеленый"
        else -> "Неизвестно"
    }
    return color
}
```

Здесь оператор `return` всего один, поэтому проще отследить возвращаемое значение.

Перегрузка функций

Иногда требуется чтобы функция могла работать с разными параметрами. Например, функция seconds, которая вычисляет количество секунд, может отличаться количеством параметров:

```
fun seconds(hours: Int): Int {
    return hours * 60 * 60
}

fun seconds(hours: Int, minutes: Int): Int {
    return hours * 60 * 60 + minutes * 60
}

fun seconds(hours: Int, minutes: Int, seconds: Int): Int {
    return hours * 60 * 60 + minutes * 60 + seconds
```

```
}
```

Если при вызове указано только одно число, то будет вызван первый вариант функции, если два – второй, если три – третий. Компилятор сам подберёт подходящую функцию:

```
println(seconds(10))         // 36000
println(seconds(10, 20))     // 37200
println(seconds(10, 20, 30)) // 37230
```

Кроме того, при перегрузке могут отличаться типы аргументов. Например, функция, которая суммирует два числа, может работать с целыми или дробными числами:

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

fun add(a: Double, b: Double): Double {
    return a + b
}
```

При вызове компилятор выберет функцию в зависимости от передаваемых аргументов:

```
println(add(1, 2))     // 3
println(add(1.0, 2.0)) // 3.0
```

Также перегрузка функций может заключаться в разном количестве аргументов:

```
fun info(name: String) {
    println("Имя: $name")
}

fun info(name: String, age: Int) {
    println("Имя: $name, возраст: $age")
}
```

В этом случае компилятор выберет подходящую функцию в зависимости от аргументов, которые передаются при вызове:

```
info("Иван")     // Имя: Иван
info("Иван", 20) // Имя: Иван, возраст: 20
```

Но перегруженные функции не могут отличаться только типом возвращаемого результата, такой вариант вызовет ошибку:

```
fun test(a: Int): Int {}
fun test(a: Int): Double {}
```

Аргументы по умолчанию

Если функции отличаются только количеством аргументов, можно обойтись без перегрузки функций. Для этого аргументам, которые можно не указывать, следует задать значения по умолчанию:

```
fun hello(name: String, job: String = "") {
    println("Добро пожаловать, $name!")
    if (job != "")
        println("Ваша должность: $job")
}
```

В этом примере обязательным является параметр name, а параметр job может быть опущен – в этом случае он станет равным пустой строке:

```
hello("Петр")
hello("Ольга", "менеджер")
```

Аргументы по умолчанию всегда должны располагаться в правой части списка, и после них не должно быть аргументов без значений по умолчанию. Например, следующее определение функции вызовет ошибку:

```
fun hello(name: String = "", job: String)
```

Переменное количество аргументов

Если у функции может быть разное количество однотипных параметров, то можно передать их с помощью ключевого слова `vararg`, в этом случае они будут помещены в массив. Например, следующая функция суммирует все переданные числа:

```
fun sum(vararg nums: Int): Int {
    var result = 0
    for (i in 0 until nums.count())
        result += nums[i]
    return result
}
```

Все переданные аргументы находятся в массиве `nums`, и в цикле происходит их суммирование. Вызывать эту функцию можно обычным образом:

```
println(sum(1, 2))           // 3
println(sum(3, 4, 5))        // 12
println(sum(6, 7, 8, 9, 10)) // 40
```

Лямбда-функции

Функцию тоже можно присвоить переменной, как и любое другое значение. Это кажется необычным, но на практике часто оказывается очень полезным, особенно при написании программ, где требуется обработка событий (например, действия при нажатии на кнопку на экране).

Пример лямбда-функции, которая суммирует два переданных значения:

```
val sum = { a: Int, b: Int -> a + b }
```

Переменной sum присваивается функция, которая получает два параметра (целые числа a и b), и возвращает их сумму. Разделителем между параметрами и выражением служит стрелка (`->`). Если функция просто вычисляет значение (как в примере, сумму чисел), то не требуется даже оператор return, можно просто написать выражение. Вызывается эта функция так же, как и любая другая функция:

```
println(sum(1, 2))
```

Если требуются более сложные действия, это тоже не проблема, можно писать обычный код, а в конце поставить выражение, которое и будет результатом:

```
val f = { a: Int, b: Int ->
    val c = a + b
    val d = b - a
    c / d
}
```

В этом примере вычисляются промежуточные значения, а последней строчкой стоит выражение деления переменной c на переменную d – результат этого деления будет являться результатом всей лямбда-функции.

Лямбда-функции можно использовать в качестве параметров других функций. Например, в массиве есть встроенная функция `filter`, которая выбирает из массива значения, соответствующие условию. Но как сообщить функции `filter` это условие? Для этого в функцию `filter` передаётся лямбда-функция, которая вызывается для каждого элемента массива, проверяет каждый элемент и делает вывод подходит ли он. Если элемент подходит, лямбда-функция возвращает `true`, иначе `false`. Пример такого использования:

```
val arr = arrayOf(1, -2, 3, -4, 5, -6, 7)
val neg = arr.filter({ a -> a < 0 })
```

После этого переменная `neg` будет содержать отрицательные числа -2, -4 и -6.

Задание (по варианту)

1. Написать метод, который в переданной строке заменяет все точки на многоточие. С его помощью обработать пять разных строк и отобразить их на экране.
2. Написать метод, который в переданной строке заменяет все строчные буквы на заглавные и наоборот. С его помощью обработать пять разных строк и отобразить их на экране.
3. Написать метод, который разделяет переданную строку на две отдельных строки: первая содержит исходную строку до первой точки, а вторая – исходную строку после первой точки. С его помощью обработать пять разных строк и отобразить результаты на экране.
4. Напишите метод, проверяющий, является ли фраза палиндромом (например, «довод»).
5. Даны две строки A и B. Написать метод, определяющий, можно ли составить строку B, используя только символы из строки A (каждую букву можно использовать только один раз).
6. Дана строка, в которой без пробелов записано арифметическое выражение в виде суммы нескольких натуральных чисел,например «1+25+3». Написать метод, вычисляющий эту сумму.
7. Дано предложение в котором разное колличество пробелов между словами. Написать метод, позволяющий напечатать все его слова в порядке убывания их длин.
8. Дано предложение в котором разное колличество пробелов между словами. Написать метод, позволяющий напечатать все его различные слова.
9. Даны два предложения, в которых разное колличество пробелов между словами. Написать метод, позволяющий напечатать слова, которые встречаются в двух предложениях только один раз.
10. Дано натуральное число n. Написать метод, позволяющий напечатать это число русскими словами (тринадцать, сто пять, двести сорок один, тысяча и т. д.).