Функции высшего порядка

Лямбда-выражения обычно не существуют сами по себе. Они тесно связаны с так называемыми **функциями высшего порядка**. Такие функции либо принимают другие функции в качестве аргументов, либо возвращают другие функции, либо делают и то и другое. Эти принимаемые и возвращаемые функции обычно являются лямбда-выражениями.

Рассмотрим программу:

```
fun main() {
    val firstList = listOf(1, 4, 10)
    val mult2: (Int) -> Int = {it * 2}
    val add2: (Int) -> Int = {it + 2}

    val multList = mathWithList(firstList, mult2)
    val addList = mathWithList(firstList, add2)

    println(multList)  // [2, 8, 20]
    println(addList)   // [3, 6, 12]
}

fun mathWithList(yourList: List<Int>, math: (Int) -> Int): List<In
    val newList = mutableListOf<Int>()
    for (i in yourList) {
        newList.add(math(i))
    }
    return newList
}
```

Функция *mathWithList* является функцией высшего порядка, потому что один из ее параметров имеет функциональный тип. В данном случае это параметр *math* с типом `(Int) -> Int`. Это значит, что при вызове `mathWithList()` ожидает, что один из передаваемых в нее аргументов будет лямбда-выражением. Каким именно, не важно. Главное, что оно должно иметь один целочисленный параметр и возвращать также число типа `Int`.

В теле *mathWithList* лямбда-функция *math* используется при обработке каждого элемента списка, то есть ей передается один элемент списка. Происходит это в выражении `math(i)`. Что делает *math* с этим элементом, функцию высшего порядка не волнует. Она всего лишь ожидает от `math()` возврата целого числа, который добавляет в новый список.

Что конкретно будет делать *math*, определяется переданным в *mathWithList* лямбда-выражением.

В функции **`main`** мы определили и присвоили переменным *mult2* и *add2* два лямбда-выражения, чей функциональный тип совпадает с лямбда-параметром функции *mathWithList*. Если выражение с **`it`** менее понятно, можно переписать его в более очевидном виде:

```
val mult2: (Int) -> Int = {n: Int -> n * 2}
```

Вспомним, Kotlin единственный параметр лямбды присваивает встроенной переменной **`it`**, при этом использовать **`it`** не обязательно. Но если **`it`** используется в теле лямбды, то параметр можно вообще не указывать. Поэтому лямбда-выражение `{n: Int -> n * 2}` преобразуется в `{it * 2}`.

Вернемся к тому, что мы присвоили лямбда-выражения переменным. Если лямбды просты и используются однократно, присваивать их переменным смысла не имеет. В этом случае лямбда-выражения проще непосредственно писать в вызове функции высшего порядка.

```
val multList = mathWithList(firstList, {it * 2})
val addList = mathWithList(firstList, {it + 2})
```

И тут мы сталкиваемся с еще одной особенностью языка Kotlin. Если лямбда-параметр является последним в списке параметров функции, то при вызове функции его можно вынести за скобку. В нашем случае выглядеть это будет так:

```
val multList = mathWithList(firstList) {it * 2}
val addList = mathWithList(firstList) {it + 2}
```

Обратим внимание, что в случае с переменными, которым присвоено лямбда-выражение, вынести так за скобку переменную не получится.

Переделаем нашу функцию *mathWithList* из обычной в функцию-расширение класса `List`:

```
fun main() {
    val firstList = listOf(1, 4, 10)

    val multList = firstList.mathWithList {it * 2}
    val addList = firstList.mathWithList {it + 2}

    println(multList)
    println(addList)
}

fun List<Int>.mathWithList(math: (Int) -> Int): List<Int> {
    val newList = mutableListOf<Int>()
    for (i in this) {
        newList.add(math(i))
    }
    return newList
}
```

В таком виде функция стала методом объекта типа `List<Int>`, к которому в теле функции мы можем обращаться через `this`. В круглых скобках у функции высшего порядка остался только один параметр – лямбда-выражение. Поэтому, когда `mathWithList()` вызывается, круглые скобки можно опустить.

В Kotlin у коллекций есть много встроенных в стандартную библиотеку методов, принимающих лямбда-выражения. Например, метод `filter` фильтрует коллекции согласно заданному через лямбда-выражение условию.

```
fun main() {
    val firstList = listOf(1, 4, 10)
    val evenList = firstList.filter { it % 2 == 0 }

    val secondSet = setOf(-4, 5, 7, -1)
    val posSet = secondSet.filter { it > 0 }
```

```
    println(evenList) // [4, 10]
    println(posSet) // [5, 7]
}
```

Рассмотрим еще один вариант функции высшего порядка – такую, которая возвращает лямбда-выражение:

```
fun main() {
    val stars = edges("***")
    val block = edges("|")

    println(stars("Hello")) // ***Hello***
    println(stars("World")) // ***World***
    println(block("Earth")) // |Earth|
}

fun edges(str: String): (String) -> String {
    return {"$str$it$str"}
}
```

Здесь тип переменных *stars* и *block* – это `(String) -> String`, то есть после вызова функции `edges()` переменные содержат лямбда-выражения. Это равносильно тому, как если бы им сразу присваивались лямбды:

```
val stars: (String) -> String = {"***$it***"}
val block: (String) -> String = {"|$it|"}
```

Поскольку Kotlin сам может выводить возвращаемый функцией тип из того, что стоит после `return`, то *edges* можно упростить, не указывая вручную возвращаемый функциональный тип:

```
fun edges(str: String) = {edge: String -> "$str$edge$str"}
```

Однако в этом случае не получится использовать `it`, так как нам придется указывать тип параметра лямбды явно.

В конце отметим, что функции высшего порядка являются основой концепции функционального программирования. Язык Kotlin поддерживает эту парадигму в полной мере.

PDF-версия курса с дополнительными уроками