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.

```
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:

```
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:

```
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.

Функции высокого порядка

Функции высокого порядка (high order function) - это функции, которые либо принимают функцию в качестве параметра, либо возвращают функцию, либо и то, и другое.

Функция как параметр функции

Чтобы функция могла принимать другую функцию через параметр, этот параметр должен представлять тип функции:

```
fun main() {

    displayMessage(::morning)
    displayMessage(::evening)
}
fun displayMessage(mes: () -> Unit){
    mes()
}
fun morning(){
    println("Good Morning")
}
fun evening(){
    println("Good Evening")
}
```

В данном случае функция `displayMessage()` через параметр `mes` принимает функцию типа `() -> Unit`, то есть такую функцию, которая не имеет параметров и ничего не возвращает.

```
fun displayMessage(mes: () -> Unit){
```

При вызове этой функции мы можем передать этому параметру функцию, которая соответствует этому типу:

```
displayMessage(::morning)
```

Рассмотрим пример параметра-функции, которая принимает параметры:

```
fun main() {

    action(5, 3, ::sum)          // 8
    action(5, 3, ::multiply)     // 15
    action(5, 3, ::subtract)     // 2
}

fun action (n1: Int, n2: Int, op: (Int, Int)-> Int){
    val result = op(n1, n2)
    println(result)
}
fun sum(a: Int, b: Int): Int{
    return a + b
}
fun subtract(a: Int, b: Int): Int{
    return a - b
}
fun multiply(a: Int, b: Int): Int{
    return a * b
}
```

Здесь функция `action` принимает три параметра. Первые два параметра - значения типа Int. А третий параметр представляет функцию, которая имеет тип `(Int, Int)-> Int`, то есть принимает два числа и возвращает некоторое число.

В самой функции action вызываем эту параметр-функцию, передавая ей два числа, и полученный результат выводим на консоль.

При вызове функции action мы можем передать для ее третьего параметра конкретную функцию, которая соответствует этому параметру по типу:

```
action(5, 3, ::sum)          // 8
action(5, 3, ::multiply)     // 15
action(5, 3, ::subtract)     // 2
```

Возвращение функции из функции

В более редких случаях может потребоваться возвратить функцию из другой функции. В этом случае для функции в качестве возвращаемого типа устанавливается тип другой функции. А в теле функции возвращается лямбда выражение. Например:

```
fun main() {
    val action1 = selectAction(1)
    println(action1(8,5))    // 13

    val action2 = selectAction(2)
    println(action2(8,5))    // 3
}
fun selectAction(key: Int): (Int, Int) -> Int{
    // определение возвращаемого результата
    when(key){
        1 -> return ::sum
        2 -> return ::subtract
        3 -> return ::multiply
        else -> return ::empty
    }
}
fun empty (a: Int, b: Int): Int{
    return 0
}
fun sum(a: Int, b: Int): Int{
    return a + b
}
fun subtract(a: Int, b: Int): Int{
    return a - b
}
fun multiply(a: Int, b: Int): Int{
    return a * b
}
```

Здесь функция `selectAction` принимает один параметр - key, который представляет тип `Int`. В качестве возвращаемого типа у функции указан

тип `(Int, Int) -> Int`. То есть `selectAction` будет возвращать некую функцию, которая принимает два параметра типа Int и возвращает объект типа Int.

В теле функции selectAction в зависимости от значения параметра key возвращается определенная функция, которая соответствует типу `(Int, Int) -> Int`.

Далее в функции `main` определяется переменная `action1` хранит результат функции `selectAction`. Так как `selectAction()` возвращает функцию, то и переменная action1 будет хранить эту функцию. Затем через переменную action1 можно вызвать эту функцию.

Поскольку возвращаемая функция соответствует типу `(Int, Int) -> Int`, то при вызове в `action1` необходимо передать два числа, и соответственно мы можем получить результат и вывести его на консоль.

Анонимные функции в Kotlin

Анонимные функции выглядят как обычные за тем исключением, что они не имеют имени. Анонимная функция может иметь одно выражение:

```
fun(x: Int, y: Int): Int = x + y
```

Либо может представлять блок кода:

```
fun(x: Int, y: Int): Int{return x + y}
```

Анонимную функцию можно передавать в качестве значения переменной:

```
fun main() {
        val message = fun()=println("Hello")
        message()
}
```

Здесь переменной `message` передается анонимная функция `fun()=println("Hello")`. Эта анонимная функция не принимает параметров и просто выводит на консоль строку "Hello". Таким образом, переменная `message` будет представлять тип `() -> Unit`.

Далее мы можем вызывать эту функцию через имя переменной как обычную функцию: `message()`.

Другой пример - анонимная функция с параметрами:

```
fun main() {
        val sum = fun(x: Int, y: Int): Int = x + y
        val result = sum(5, 4)
        println(result)     // 9
}
```

В данном случае переменной `sum` присваивается анонимная функция, которая принимает два параметра - два целых числа типа Int и возвращает их сумму.

Также через имя переменной мы можем вызвать эту анонимную функцию, передав ей некоторые значения для параметров и получить ее результат: `val result = sum(5, 4)`

Анонимная функция как аргумент функции

Анонимную функцию можно передавать в функцию, если параметр соответствует типу этой функции:

```
fun main() {
        doOperation(9,5, fun(x: Int, y: Int): Int = x + y ) // 14
        doOperation(9,5, fun(x: Int, y: Int): Int = x - y) // 4
        val action = fun(x: Int, y: Int): Int = x * y
        doOperation(9, 5, action) // 45
}
fun doOperation(x: Int, y: Int, op: (Int, Int) ->Int) {
        val result = op(x, y)
        println(result)
}
```

Возвращение анонимной функции из функции

И также фунция может возвращать анонимную функцию в качестве результата:

```
fun main() {

    val action1 = selectAction(1)
    val result1 = action1(4, 5)
    println(result1)        // 9
```

```
    val action2 = selectAction(3)
    val result2 = action2(4, 5)
    println(result2)        // 20

    val action3 = selectAction(9)
    val result3 = action3(4, 5)
    println(result3)        // 0
}

fun selectAction(key: Int): (Int, Int) -> Int{
    // определение возвращаемого результата
    when(key){
        1 -> return fun(x: Int, y: Int): Int = x + y
        2 -> return fun(x: Int, y: Int): Int = x - y
        3 -> return fun(x: Int, y: Int): Int = x * y
        else -> return fun(x: Int, y: Int): Int = 0
    }
}
```

Здесь функция `selectAction()` в зависимости от переданного значения возвращает одну из четырех анонимных функций. Последняя анонимная функция `fun(x: Int, y: Int): Int = 0` просто возвращает число 0.

При обращении к `selectAction()` переменная получит определенную анонимную функцию:

```
val action1 = selectAction(1)
```

То есть в данном случае переменная `action1` хранит ссылку на функцию `fun(x: Int, y: Int): Int = x + y`

Лямбда-выражения

Лямбда-выражения представляют небольшие кусочки кода, которые выполняют некоторые действия. Фактически лямбды преставляют сокращенную запись функций. При этом лямбды, как и обычные и анонимные функции, могут передаваться в качестве значений переменным и параметрам функции.

Лямбда-выражения оборачиваются в фигурные скобки:

```
{println("hello")}
```

В данном случае лямбда-выражение выводит на консоль строку "hello".

Лямбда-выражение можно сохранить в обычную переменную и затем вызывать через имя этой переменной как обычную функцию.

```
fun main() {

    val hello = {println("Hello Kotlin")}
    hello()
    hello()
}
```

В данном случае лямбда сохранена в переменную hello и через эту переменную вызывается два раза. Поскольку лямбда-выражение представляет сокращенную форму функции, то переменная hello имеет тип функции () -> Unit.

```
val hello: ()->Unit = {println("Hello Kotlin")}
```

Также лямбда-выражение можно запускать как обычную функцию, используя круглые скобки:

```
fun main() {

    {println("Hello Kotlin")}()
}
```

Следует учитывать, что если до подобной записи идут какие-либо инструкции, то Kotlin автоматически может не определять, что определения лямбда-выражения составляет новую инструкцию. В этом случае предыдущую инструкции можно завершить точкой с запятой:

```
fun main() {

    {println("Hello Kotlin")}();
    {println("Kotlin on Metanit.com")}()
}
```

Передача параметров

Лямбды как и функции могут принимать параметры. Для передачи параметров используется стрелка ->. Параметры указываются слева от стрелки, а тело

лямбда-выражения, то есть сами выполняемые действия, справа от стрелки.

```
fun main() {

    val printer = {message: String -> println(message)}
    printer("Hello")
    printer("Good Bye")
}
```

Здесь лямбда-выражение принимает один параметр типа String, значение которого выводится на консоль. Переменная printer в данном случае имеет тип `(String) -> Unit`.

При вызове лямбда-выражения сразу при его определении в скобках передаются значения для его параметров:

```
fun main() {

    {message: String -> println(message)}("Welcome to Kotlin")
}
```

Если параметров несколько, то они передаются слева от стрелки через запятую:

```
fun main() {

    val sum = {x:Int, y:Int -> println(x + y)}
    sum(2, 3)   // 5
    sum(4, 5)   // 9
}
```

Если в лямбда-выражении надо выполнить не одно, а несколько действий, то эти действия можно размещать на отдельных строках после стрелки:

```
val sum = {x:Int, y:Int ->
    val result = x + y
    println("$x + $y = $result")
}
```

Возвращение результата

Выражение, стоящее после стрелки, определяет результат лямбда-выражения. И этот результат мы можем присвоить, например, переменной.

Если лямбда-выражение формально не возвращает никакого результата, то фактически, как и в функциях, возвращается значение типа Unit:

```
val hello = { println("Hello")}
val h = hello()              // h представляет тип Unit

val printer = {message: String -> println(message)}
val p = printer("Welcome")    // p представляет тип Unit
```

В обоих случаях используется функция println, которая формально не возвращает никакого значения (точнее возвращает объект типа Unit).

Но также может возвращаться конкретное значение:

```
fun main() {

    val sum = {x:Int, y:Int -> x + y}

    val a = sum(2, 3)   // 5
    val b = sum(4, 5)   // 9
    println("a=$a  b=$b")
}
```

Здесь выражение справа от стрелки x + y продуцирует новое значение - сумму чисел, и при вызове лямбда-выражения это значение можно передать переменной. В данном случае лямбда-выражение имеет тип `(Int, Int) -> Int`.

Если лямбда-выражение многострочное, состоит из нескольких инструкций, то возвращается то значение, которое генерируется последней инструкцией:

```
val sum = {x:Int, y:Int ->
    val result = x + y
    println("$x + $y = $result")
    result
}
```

Последнее выражение по сути представляет число - сумму чисел x и y и оно будет возвращаться в качестве результата лямбда-выражения.

Лямбда-выражения как аргументы функций

Лямбда-выражения можно передавать параметрам функции, если они представляют один и тот же тип функции:

```
fun main() {

    val sum = {x:Int, y:Int -> x + y }
    doOperation(3, 4, sum)                          // 7
    doOperation(3, 4, {a:Int, b: Int -> a * b}) // 12

}
fun doOperation(x: Int, y: Int, op: (Int, Int) ->Int){

    val result = op(x, y)
    println(result)
}
```

Типизиция параметров лямбды

При передаче лямбды параметру или переменной, для которой явным образом указан тип, мы можем опустить в лямбда-выражении типы параметров:

```
fun main() {
    val sum: (Int, Int) -> Int = {x, y -> x + y }
    doOperation(3, 4, {a, b -> a * b})
}
fun doOperation(x: Int, y: Int, op: (Int, Int) ->Int){

    val result = op(x, y)
    println(result)
}
```

Здесь в случае с переменной sum Kotlin видит, что ее тип `(Int, Int) -> Int`, то есть и первый, и второй параметр представляют тип Int. Поэтому при присвоении переменной лямбды `{x, y -> x + y }` Kotlin автоматически поймет, что параметры `x` и `y` представляют именно тип `Int`.

То же самое касается и вызова функции `doOperation()` - при передаче в него лямбды Kotlin автоматически поймет какой параметр какой тип представляет.

trailing lambda

Если параметр, который принимает функцию, является последним в списке, то при передачи ему лямбда-выражения, саму лямбду можно прописать после

списка параметров. Например, возьмем выше использованную функцию doOperation():

```
fun doOperation(x: Int, y: Int, op: (Int, Int) ->Int){

    val result = op(x, y)
    println(result)
}
```

Здесь параметр, который представляет функцию - параметр op, является последним в списке параметров. Поэтому вместо того, чтобы написать так:

```
doOperation(3, 4, {a, b -> a * b}) // 12
```

Мы также можем написать так:

```
doOperation(3, 4) {a, b -> a * b} // 12
```

То есть вынести лямбду за список параметров. Это так называемая конечная лямбда или trailing lambda

Возвращение лямбда-выражения из функции

Также фукция может возвращать лямбда-выражение, которое соответствует типу ее возвращаемого результата:

```
fun main() {
    val action1 = selectAction(1)
    val result1 = action1(4, 5)
    println(result1)        // 9

    val action2 = selectAction(3)
    val result2 = action2(4, 5)
    println(result2)        // 20

    val action3 = selectAction(9)
    val result3 = action3(4, 5)
    println(result3)        // 0
}
fun selectAction(key: Int): (Int, Int) -> Int{
    // определение возвращаемого результата
    when(key){
```

```
        1 -> return {x, y -> x + y }
        2 -> return {x, y -> x - y }
        3 -> return {x, y -> x * y }
        else -> return {x, y -> 0 }
    }
}
```

Неиспользуемые параметры

Обратим внимание на предыдущий пример на последнюю лямбду:

```
else -> return {x, y -> 0 }
```

Если в функцию selectAction() передается число, отличное от 1, 2, 3, то возвращается лямбда-выражение, которое просто возвращает число 0. С одной стороны, это лямбда-выражение должно соответствовать типу возвращаемого результата функции `selectAction()` - `(Int, Int) -> Int`

С другой стороны, оно не использует параметры, эти параметры не нужны. В этом случае вместо неиспользуемых параметров можно указать прочерки:

```
else -> return {_, _ -> 0 }
```