Функциональное программирование

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.

```
fun имя_функции (параметры) : возвращаемый_тип{
    выполняемые инструкции
}
```

Параметры необязательны.

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

```
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 сообщение на консоль:

```
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:

```
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:

```
displayUser()
```

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

```
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:

```
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:

```
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:

```
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:

```
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.:

```
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:

```
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.

```
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:

```
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 именованных аргументов.

Оператор *

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

```
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:

```
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:

```
fun sum(x:Int, y:Int): Int
```

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

```
return x + y
```

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

```
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

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

будет аналогична следующей:

```
fun hello() : Unit{
    println("Hello")
}
```

Формально мы даже можем присвоить результат такой функции переменной:

```
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:

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

```
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:

```
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:

```
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:

```
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:

```
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.

```
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.

Тип функции

В Kotlin все является объектом, в том числе и функции. И функции, как и другие объекты, имеют определенный тип. Тип функции определяется следующим образом:

```
(типы_параметров) -> возвращаемый_тип
```

Возьмем функцию которая не принимает никаких параметров и ничего не возвращает:

```
fun hello(){

    println("Hello Kotlin")
}
```

Она имеет тип

```
() -> Unit
```

Если функция не принимает параметров, в определении типа указываются пустые скобки. Если не указан возвращаемый тип, то фактически а в качестве типа возвращаемого значения применяется тип Unit.

Возьмем другую функцию:

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

Эта функция принимает два параметра типа Int и возвращает значение типа Int, поэтому она имеет тип

```
(Int, Int) -> Int
```

Что дает нам знание типа функции? Используя тип функции, мы можем определять переменные и параметры других функций, которые будут представлять функции.

Переменные-функции

Переменная может представлять функцию. С помощью типа функции можно определить, какие именно функции переменная может представлять:

```
fun main() {

    val message: () -> Unit
    message = ::hello
    message()
}

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

Здесь переменная message представляет функцию с типом () -> Unit, то есть функцию без параметров, которая ничего не возвращает. Далее определена как раз такая функция - hello(), соответственно мы можем передать функцию hello переменной message.

Чтобы передать функцию, перед названием функции ставится оператор ::

```
message = ::hello
```

Затем мы можем обращаться к переменной message() как к обычной функции:

```
message()
```

Так как переменная message ссылается на функцию hello, то при вызове message() фактически будет вызываться функция hello().

При этом тип функции также может выводится исходя из присваемого переменной значения:

```
val message = ::hello // message имеет тип () -> Unit
```

Рассмотрим другой пример, когда переменная ссылается на функцию с параметрами:

```
fun main() {

    val operation: (Int, Int) -> Int = ::sum
    val result = operation(3, 5)
    println(result) // 8
}
fun sum(a: Int, b: Int): Int{
    return a + b
}
```

Переменная operation представляет функцию с типом (Int, Int) -> Int, то есть функцию с двумя параметрами типа Int и возвращаемым значением типа Int. Соответственно такой переменной мы можем присвоить функцию sum, которая соответствует этому типу.

Затем через имя переменной фактически можно обращаться к функции sum(), передавая ей значения для параметров и получая ее результат:

```
val result = operation(3, 5)
```

При этом динамически можно менять значение, главное чтобы оно соответствовало типу переменной:

```
fun main() {

    // operation указывает на функцию sum
    var operation: (Int, Int) -> Int = ::sum
    val result1 = operation(14, 5)
    println(result1) // 19

    // operation указывает на функцию subtract
    operation = ::subtract
    val result2 = operation(14, 5)
    println(result2) // 9

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

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

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 }
```