Знакомство с функциональной парадигмой языка программирования Kotlin.

Романов Владимир Юрьевич
МГУ им. М.В.Ломоносова, ф-т ВМК
romanov.rvy@yandex.ru

Октябрь 31, 2020

2.1. ОБЗОР ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ЯЗЫКА KOTLIN

Функции в языке Kotlin:

Особенности функций в языке Kotlin:

- могут храниться в переменных и структурах данных
- передаваться как параметры других функций
- возвращаться как результат работы других функций

Для этих целей в языке используются:

- ***функциональные типы***
- конструкции языка называемые ***лямбда-выражениями***

Анонимная функция в Kotlin

Переменной **sum** присваивается *анонимная функция*:

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

fun main() {
    println("sum: " + sum(1,2))
}
```

Функциональный тип

Явное задание функционального типа **(Int,Int) -> Int**
для переменной **sum**:

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

fun main() {
    println("sum: " + sum(1,2))
}
```

Псевдоним функционального типа

Задание псевдонима **BinFun** для функционального типа **(Int,Int) -> Int**:

```
typealias BinFun = (Int, Int) -> Int

val sum: BinFun
    = fun(a: Int, b: Int): Int = a + b

fun main() {
    println("sum: " + sum(1,2))
}
```

Лямбда - выражение

Замена анонимной функции

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

на лямбда-выражение

```
{ a: Int, b: Int -> a + b }
```

```
typealias BinFun = (Int, Int) -> Int

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

fun main() {
    println("sum: " + sum(1,2))
}
```

Выведение типа переменной

Выведение типа **Int** для результата вызова функции:

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

fun main() {
    println("sum: " + sum(1,2))
}
```

Лямбда-выражение для вызова функции

Явное использование лямбда-выражения для вызова функции с помощью операции ():

```
fun main() {
    println("sum: "
        + { a: Int, b: Int -> a + b }(1,2)  )
}
```

Явное использование лямбда-выражения для вызова функции с помощью функции **invoke**

```
fun main() {
    println("sum: "
        + { a: Int, b: Int -> a + b }.invoke(1,2)  )
}
```

Пример функции с лямбдой-параметром

Функцию ***forEach*** можно вызвать для любой коллекции.
Она принимает один аргумент ***action***:
функцию, определяющую какие действия надо выполнить для каждого элемента.

```
public inline
fun <T> Iterable<T>.forEach
    (action: (T) -> Unit): Unit {
        for (element in this)
            action(element)
}
```

Лямбда-выражение как последний параметр функции

Особенности синтаксиса языка Kotlin при использовании лямбда выражений

```
data class Person(val name: String, var age: Int)
fun Person.print()
    = println("$name age is $age years")

val people: List<Person>
    = listOf( Person("Аня", 5), Person("Оля", 7) )

people.forEach({ p: Person -> p.print() })
people.forEach() { p: Person -> p.print() }
people.forEach { p: Person -> p.print() }
people.forEach { p -> p.print() }
people.forEach { it.print() }
```

Лямбда-выражение как ссылка на функцию

Лямбда-выражения - параметр хранимый в переменной которые можно передавать в качестве параметров функций

```
data class Person(val name: String, var age: Int)

fun Person.print()
    = println("$name is $age years")

people.forEach(Person::print)
```

Пример лямбда-выражения как последнего параметра

```
fun <T> Iterable<T>.joinToString
        (separator: CharSequence = ", ",
         prefix: CharSequence = "",
         postfix: CharSequence = "",
         limit: Int = -1,
         truncated: CharSequence = "...",
         transform: ((T) -> CharSequence)? = null)
  : String

people.joinToString(" ") { p: Person -> p.name }

people.joinToString(prefix="(", postfix=")")
      { p: Person -> "${p.name}[${p.age}]" }
```

ЛЯМБДА ВЫРАЖЕНИЯ ПРИ РАБОТЕ С КОЛЛЕКЦИЯМИ

Лямбда в функциях filter, map, forEach

```
fun PutPiecePlayer.allCorrectMoves(
            emptySquares: List<Square>,
            piece: Piece)
          : MutableList<Move>
= emptySquares
  .filter { s: Square -> piece.isCorrectMove(s) }
  .map { s: Square -> piece.makeMove(s) }
  .toMutableList()
```

Лямбда в функциях filter, map, forEach (упрощение)

Для лямбда выражения с единственным параметром ***it***

```
fun PutPiecePlayer.allCorrectMoves(
            emptySquares: List<Square>,
            piece: Piece)
          : MutableList<Move>
= emptySquares
  .filter { piece.isCorrectMove(it) }
  .map { piece.makeMove(it) }
  .toMutableList()
```

Функция filter (реализация)

```
public inline
fun <T> Iterable<T>.filter
        (predicate: (T) -> Boolean): List<T>
```

- Функция **filter** - это расширение *интерфейса* **Iterable**
- Применима к реализующему **Iterable** классу **List**
- Применима к реализующему **Iterable** классу **Collection** <: **List**
- Применима к реализующему **Iterable** классу **Range**

```
emptySquares.filter
      { s: Square -> piece.isCorrectMove(s) }

(0..10).filter { it % 2 == 0 }
      .forEach{ println(it) }
```

Интерфейсы Iterable и Iterator

```
public interface Iterable<out T> {
    public operator fun iterator(): Iterator<T>
}

public interface Iterator<out T> {
    public operator fun next(): T
    public operator fun hasNext(): Boolean
}
```

Функция map (реализация)

public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R>

- Функция **filter** - это расширение *интерфейса* **Iterable**
- Применима к реализующему **Iterable** классу **List**
- Применима к реализующему **Iterable** классу **Collection** <: **List**
- Применима к реализующему **Iterable** классу **IntRange** <: **IntProgression**

```
emptySquares
  .filter { s: Square -> piece.isCorrectMove(s) }
  .map { s: Square -> piece.makeMove(s) }

(0..10).map { it * 2 }
       .forEach{ println(it) }
```

Функции с предикатами all, any, count и find

```
data class Person(val name: String, var age: Int)
val people: List<Person>
    = listOf( Person("Аня", 5), Person("Оля", 7) )

val anyLess7: Boolean  = people.any{ it.age < 7}
val allMore10: Boolean = people.all{ it.age > 10}
val countMore10: Int = people.count{ it.age > 10}
val countMore: Person? = people.find{ it.age > 10}
```

Группирование элементов коллекции

Объявление функции **groupBy**

```
public inline
fun <T, K> Iterable<T>.groupBy
        (keySelector: (T) -> K): Map<K, List<T>>
```

Примеры группирования элементов коллекции по возрасту и имени

```
var ageGroup: Map<Int, List<Person>>
    = people.groupBy { p: Person -> p.age }

var nameGroup: Map<String, List<Person>>
    = people.groupBy { p: Person -> p.name }
```

ФУНКЦИОНАЛЬНЫЙ ТИП С ПОЛУЧАТЕЛЕМ

Функция, анонимная функция, лямбда

```
fun sub1(the:Int, other: Int)
    = the - other
```

Функция **sub1** - это ***функция*** с телом-выражением

```
val sub2 = fun(the:Int, other: Int)
         = the - other
```

Анонимная функция присвоена переменной **sub2** имеющей *функциональный тип* **(Int, Int)->Int**.
Имя функции перенесено в имя переменной.

```
val sub3 = {the:Int, other: Int -> the - other}
```

Лямбда-выражение присвоено переменной **sub3**.

Функция-расширение, анонимная функция-расширение, лямбда с получателем

```
fun Int.sum1(other: Int)
    = this + other
```

Функция **sum1** это - ***функция-расширение*** с телом-выражением

```
val sum2 = fun Int.(other: Int): Int
         = this + other
```

Анонимная функция-расширение присвоена переменной **sum2** имеющей *функциональный тип* **Int.(Int)->Int**.
Имя функции перенесено в имя переменной.

```
val sum3: Int.(Int) -> Int
        = { other -> this + other }
```

Лямбда-выражение с получателем присвоено переменной **sum3**.

Функциональный тип с получателем

- У функциональных типов может быть дополнительный тип - ***получатель (receiver)***
- Получатель указывается в объявлении перед точкой.
  Например, 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-получателем часто используются вместе с этими типами.

Лямбда-выражение с получателем

- Возможность вызова методов другого объекта в теле лямбда-выражений без дополнительных квалификаторов.
- Такие конструкции называются ***лямбда-выражениями с получателями***.

Стандартная функция with

- Функция ***with*** позволяет выполнить несколько операций над одним и тем же объектом-получателем, не повторяя имени объекта.
- Объект-получатель передается первым параметром функции, исполняемый блок - вторым параметром.
- возвращает результат лямбда-выражения

```
val file = File("file.txt")
with(file) {
    setReadable(true)
    setWritable(true)
    setExecutable(false)
}
```

Стандартная функция with (объявление)

```
public inline
fun <T, R> with(receiver: T, block: T.() -> R): R {
    return receiver.block()
}
```

- Использование *получателя лямбды* возможно через ключевое слово ***this***.
- ключевое слово ***this*** может быть опущено

```
val file = File("file.txt")
with(file) {
    this.setReadable(true)
    this.setWritable(true)
    this.setExecutable(false)
}
```

Стандартная функция apply

- Функция ***apply*** позволяет выполнить несколько операций над одним и тем же объектом-получателем, не повторяя имени объекта.
- Функция ***apply*** работает *почти так же*, как ***with***
  - ***apply*** возвращает *объект-получатель*
  - ***with*** возвращает *результат лямбды*

```
val file = File("file.txt").apply {
    setReadable(true)
    setWritable(true)
    setExecutable(false)
}
```

Без использования apply и с ее использованием

```
val file = File("file.txt")
file.setReadable(true)
file.setWritable(true)
file.setExecutable(false)
```

То же самое без повторения имени объекта с использованием функции ***apply***

```
val file = File("file.txt").apply {
    setReadable(true)
    setWritable(true)
    setExecutable(false)
}
```

Стандартная функция apply (объявление)

```
public inline
fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

- Использование *получателя лямбды* возможно через ключевое слово ***this***.
- ключевое слово ***this*** может быть опущено

```
val file = File("file.txt").apply {
    this.setReadable(true)
    this.setWritable(true)
    this.setExecutable(false)
}
```

Стандартная функция let

- Функция ***let*** определяет переменную в область видимости заданной лямбды и позволяет использовать ключевое слово ***it*** для ссылки на нее
- Функция ***let*** облегчает работу с выражениями, допускающими значение ***null***.
- Вместе с оператором безопасного вызова она позволяет в одном коротком выражении:
  - вычислить выражение,
  - проверить результат на **null** и сохранить его в переменной

```
fun greeting(guest: String?): String {
  return guest ?. let{ "Hello $it." }
                ?: "Bye."
}
```

Чаще всего она используется для передачи аргумента, который может оказаться равным ***null,*** в функцию, которая ожидает параметра, не равного ***null.***

Стандартная функция let (объявление)

Функция ***let*** возвращает результат лямбды

```
public inline
fun <T, R> T.let(block: (T) -> R): R {
   return block(this)
}
```

Стандартная функция run

- Функция ***run*** похожа на ***apply***
  - ***apply*** возвращает объект-приемник.
  - ***run*** возвращает результат лямбды

```
public inline
fun <T, R> T.run(block: T.() -> R): R {
    return block()
}
public inline
fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

Стандартная функция run (использование)

- Функция ***run*** может использоваться для выполнения ссылки на функцию относительно объекта-приемника

```
fun nameIsLong(name: String) = name.length >= 20
fun createMessage(isNameLong: Boolean)
    = if (isNameLong) "Name is too long"
      else "Welcome"

"Romanov Vladimir Yrievich"
   .run(::nameIsLong)
   .run(::createMessage)
   .run(::println)
```

Без функции ***run***

```
println(createMessage(
   nameIsLong("Romanov Vladimir Yrievich")))
```

Стандартные функции

Функция	Передает объект-приемник в лямбду как аргумент?	Ограничивает относительную область видимости?	Возвращает
let	Да	Нет	Результат лямбды
apply	Нет	Да	Объект-приемник
run	Нет	Да	Результат лямбды
with	Нет	Да	Результат лямбды

Пример использования лямбда выражения с получателем

Лямбда-выражения могут быть использованы как литералы функций с приёмником, когда тип получателя может быть выведен из контекста.

```
class HTML {
  fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML() // создание объекта-получателя
  html.init() // передача получателя в лямбду
  return html
}

html {   // лямбда с получателем начинается тут
  body() // вызов метода объекта-получателя
}
```