Тема 3. Коллекции языка Kotlin

Романов Владимир Юрьевич
МГУ им. М.В.Ломоносова, ф-т ВМК
vladimir.romanov@gmail.com

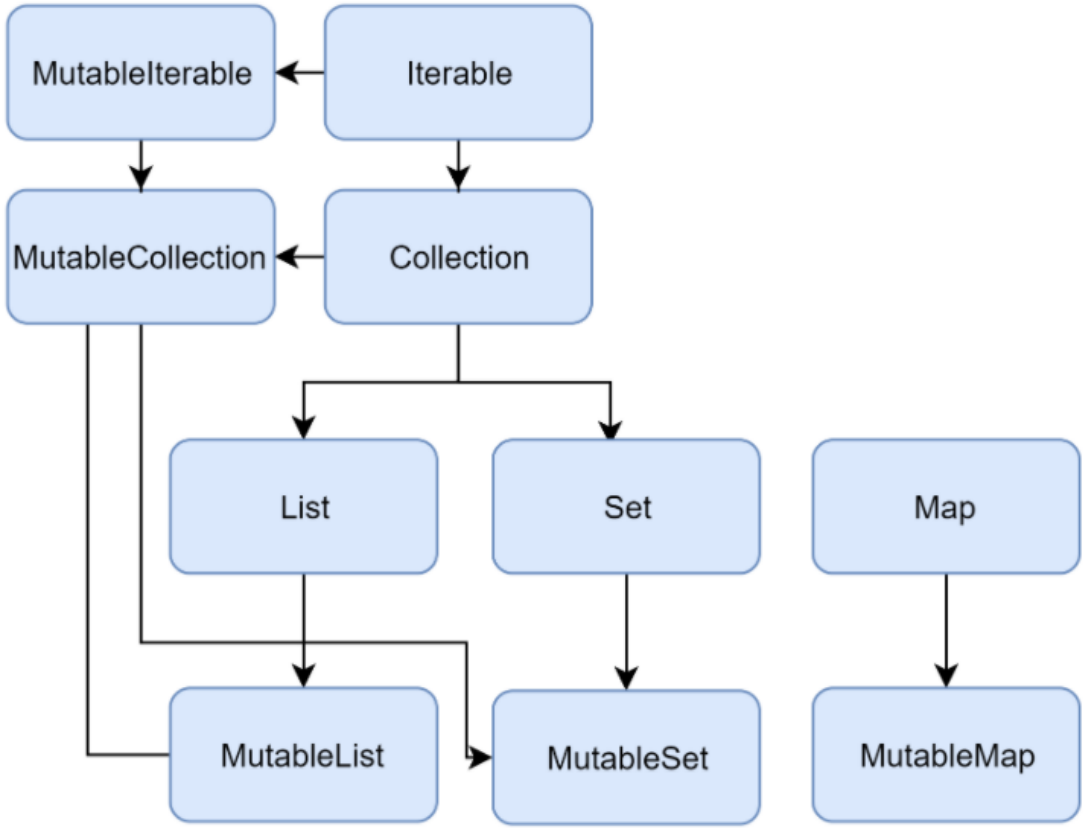
Ноябрь 13, 2020

1.4. КОЛЛЕКЦИИ В KOTLIN

Виды коллекций. Изменяемые и не изменяемые

- *Стандартная библиотека Kotlin* предоставляет реализации для основных видов коллекций:
  - Множество
  - Список
  - Карта
- Пара интерфейсов представляет каждый вида коллекции:
  - интерфейс *только для чтения* - только операции для доступа к элементам коллекции.
  - интерфейс *изменяемый*, расширяет интерфейс *только для чтения* операциями записи:
    добавления, удаления и обновления его элементов.

Виды коллекций. Схема



Коллекции в языке Kotlin

Виды коллекций. Collection (Коллекция)

- Интерфейс **Collection** поддерживает доступ к коллекции только *для чтения*

```
public interface Collection<out E> : Iterable<E> {
 public val size: Int
 public fun isEmpty(): Boolean
 public operator fun contains(element: E): Boolean
 override fun iterator(): Iterator<E>
 public fun containsAll(elements: Collection<E>)
          : Boolean
}
```

- Потомки **Collection** интерфейсы **List** и **Set**
- Коллекции *только для чтения* ***ковариантны (out)***. Т.е если класс **Rectangle** наследуется от **Shape**, можно использовать **List<Rectangle>** везде, где требуется **List<Shape>**.
- Если **Rectangle :> Shape** то **List<Rectangle> :> List<Shape>**

Виды коллекций. Итераторы и итерирумые

```
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
}
```

Виды коллекций. MutableCollection (Изменяемая коллекция)

- **MutableCollection** - это коллекция с *операциями записи*, такими как **add** и **remove**.
- Потомки **Collection** классы **List** и **Set**
- Изменяемые коллекции *не ковариантны*, т.к это привело бы к сбоям во время выполнения.
- Если **MutableList<Rectangle>** был подтипом **MutableList<Shape>**, в него можно было бы вставить других наследников **Shape** (например, **Circle**), тем самым нарушив его аргумент типа **Rectangle**.

Объявление MutableCollection (Изменяемая коллекция)

```
public interface MutableCollection<E>
      : Collection<E>, MutableIterable<E>
{
 override fun iterator(): MutableIterator<E>

 public fun add(element: E): Boolean
 public fun remove(element: E): Boolean
 public fun addAll(elements: Collection<E>)
          : Boolean
 public fun removeAll(elements: Collection<E>)
          : Boolean
 public fun retainAll(elements: Collection<E>)
          : Boolean
 public fun clear(): Unit
}
```

Виды коллекций. Модифицируемые итераторы и итерирумые

– В интерфейс **MutableIterator** добавлена функция **remove**

```
public interface MutableIterable<out T>
      : Iterable<T>
{
  override fun iterator(): MutableIterator<T>
}
public interface MutableIterator<out T>
                : Iterator<T>
{
    public fun remove(): Unit
}
```

Виды коллекций. List (Список)

- **List** - это упорядоченная коллекция с доступом к элементам по их номеру.
- Элементы могут встречаться в списке более одного раза.
- Пример списка - это предложение:
  группа слов порядок которых важен и которые могут повторяться.
- В Kotlin реализацией **List** по умолчанию является **ArrayList**, который можно рассматривать как массив с изменяемым размером.

```
val stringList: List<String>
   = listOf("This", "is", "cat")
```

Объявление Списка

```
public interface List<out E> : Collection<E> {
 override val size: Int
 override fun isEmpty(): Boolean
 override fun contains(element: E): Boolean
 override fun iterator(): Iterator<E>
 override fun containsAll(elements:
              Collection<E>): Boolean

 public operator fun get(index: Int): E
 public fun indexOf(element: E): Int
 public fun lastIndexOf(element: E): Int
 public fun listIterator(): ListIterator<E>
 public fun listIterator(index: Int)
          : ListIterator<E>
 public fun subList(fromIndex: Int, toIndex: Int)
          : List<E>
}
```

Пример использования списка

```
val numbers
    = listOf("one", "two", "three", "four")

println("Number of elements: ${numbers.size}")
println("Third element: ${numbers.get(2)}")
println("Fourth element: ${numbers[3]}")
println(""""Index of element "two"
  | ${numbers.indexOf("two")}""".trimMargin())
```

Виды коллекций. MutableList (Изменяемый список) (1)

```
public interface MutableList<E>
                : List<E>, MutableCollection<E> {
 override fun add(element: E): Boolean
 override fun remove(element: E): Boolean
 override fun addAll(elements: Collection<E>)
            : Boolean
 override fun removeAll(elements: Collection<E>)
            : Boolean
 override fun retainAll(elements: Collection<E>)
            : Boolean
 override fun clear(): Unit
 // ...
}
```

Виды коллекций. MutableList (Изменяемый список) (2)

```
public interface MutableList<E>
                : List<E>, MutableCollection<E> {
 // ...
 override fun listIterator()
            : MutableListIterator<E>
 override fun listIterator(index: Int)
            : MutableListIterator<E>
 override fun subList(fromIndex: Int,
                    toIndex: Int)
            : MutableList<E>
 // ...
}
```

Виды коллекций. MutableList (Изменяемый список) (3)

```
public interface MutableList<E>
                : List<E>, MutableCollection<E> {
 // ...

 public fun addAll(index: Int,
                   elements: Collection<E>)
            : Boolean
 public operator fun set(index: Int, element: E)
                    : E
 public fun add(index: Int, element: E): Unit
 public fun removeAt(index: Int): E
}
```

Пример использования изменяемого списка

```
 val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

Виды коллекций. Set (Множество)

- **Set** - это множество уникальных элементов.
- Абстракция множества: группа элементов без повторения.
- Пример множества алфавит - это набор букв.
- Умалчиваемая реализация множества **LinkedHashSet**
- Другая реализация **HashSet** не сохраняет порядок элементов.
- Но **HashSet** требует меньше памяти для хранения того же количества элементов.

```
val stringSet: Set<String> = setOf("a", "b", "c")
```

Объявление множества

```
public interface Set<out E> : Collection<E> {
 override val size: Int
 override fun isEmpty(): Boolean
 override fun contains(element: E): Boolean
 override fun iterator(): Iterator<E>
 override fun containsAll(elements: Collection<E>)
            : Boolean
}
```

Пример использования множества

```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")

if (numbers.contains(1))
    println("1 is in the set")

val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal:" +
        " ${numbers == numbersBackwards}")
```

Объявление изменяемого множества

– Умалчиваемая реализация изменяемого множества **LinkedHashSet**

```
public interface MutableSet<E>
      : Set<E>, MutableCollection<E>
{
 override fun iterator(): MutableIterator<E>

 override fun add(element: E): Boolean
 override fun remove(element: E): Boolean

 override fun addAll(elements: Collection<E>)
            : Boolean
 override fun removeAll(elements: Collection<E>)
            : Boolean
 override fun retainAll(elements: Collection<E>)
            : Boolean
 override fun clear(): Unit
}
```

Пример использования изменяемого множества

```
val numbers = setOf(1, 2, 3, 4)
val numbersBackwards = setOf(4, 3, 2, 1)

println(numbers.first() == numbersBackwards.first())
println(numbers.first() == numbersBackwards.last())
```

Виды коллекций. Map (Карта)

- **Map** - Карта (или словарь) - это набор *пар* (*entry*) *ключ-значение*.
- *Ключи* уникальны, и каждый из них соответствует ровно одному *значению*.
- *Значения* могут повторяться.
- Пример: *ключ* это английское слово
  значение это список русских слов - переводов английского слова
- По умолчанию реализация карты ***LinkedHashMap***. Она сохраняет порядок вставки элементов
- Другая реализация карты **HashMap** не сохраняет порядок элементов.

```
val numbersMap: Map<String, Int>
  = mapOf("One" to 1,
          "Two" to 2,
          "Three" to 3,
          "Four" to 4)
```

Объявление карты

```
public interface Map<K, out V> {
 public val size: Int
 public fun isEmpty(): Boolean
 public fun containsKey(key: K): Boolean
 public fun containsValue(value: V): Boolean
 public operator fun get(key: K): V?

 public val keys: Set<K>
 public val values: Collection<V>
 public val entries: Set<Map.Entry<K, V>>

 public interface Entry<out K, out V> {
    public val key: K
    public val value: V
 }
}
```

Примеры использования карты

```
val numbersMap = mapOf("key1" to 1,
                       "key2" to 2,
                       "key3" to 3,
                       "key4" to 1)

println("All keys: ${numbersMap.keys}")
println("All values: ${numbersMap.values}")

if ("key2" in numbersMap)
   println("""Value by key "key2":"""
           + """ ${numbersMap["key2"]}""")
if (1 in numbersMap.values)
   println("The value 1 is in the map")

if (numbersMap.containsValue(1))
   println("The value 1 is in the map")
```

Виды коллекций. MutableMap (Изменяемая карта)

```
public interface MutableMap<K, V>
      : Map<K, V> {
 public fun put(key: K, value: V): V?
 public fun remove(key: K): V?
 public fun putAll(from: Map<out K, V>)
           : Unit
 public fun clear(): Unit

 override val keys: MutableSet<K>
 override val values: MutableCollection<V>
 override val entries:
   MutableSet<MutableMap.MutableEntry<K, V>>

 public interface MutableEntry<K, V>
                : Map.Entry<K, V> {
   public fun setValue(newValue: V): V
 }
}
```

Примеры использования изменяемой карты

```
val numbersMap1: MutableMap<String, Int>
    = mutableMapOf("one" to 1, "two" to 2)

numbersMap1.put("three", 3)
numbersMap1["one"] = 11

println(numbersMap1)
```

Конструирование множеств

```
val emptySet = emptySet<String>()

val set1 = setOf<String>()
val numbersSet = setOf("one", "two", "three")

val mSet1 = mutableSetOf<String>()
val mNumbersSet = mutableSetOf("one", "two", "three")
val presizedSet = HashSet<Int>(32)
```

Конструирование списков

```
val emptyList = emptyList<String>()

val list1 = listOf<String>()
val numbersList = listOf("one", "two", "three")
val doubled = List(3) { it * 2 }
val linkedList = LinkedList<String>(listOf("one", "two", "three"))

val mList1 = mutableListOf<String>()
val mNumbersList = mutableListOf("one", "two", "three")
```

Конструирование карт

```
val emptyMap = emptyMap<String, Int>()

val numbersMap = mapOf("key1" to 1,
                       "key2" to 2,
                       "key3" to 3)
val numbersMap
   = mutableMapOf<String, String>()
       .apply { this["one"] = "1";
                this["two"] = "2" }
```

Конструирование копированием

```
val sourceList = mutableListOf(1, 2, 3)
val copyList = sourceList.toMutableList()
val readOnlyCopyList = sourceList.toList()

val sourceList = mutableListOf(1, 2, 3)
val copySet = sourceList.toMutableSet()
```

Конструирование вызовом функций над коллекциями

```
val strings = listOf("one", "two", "three")
val longerThan3 = strings.filter { it.length > 3 }

val numbers = setOf(1, 2, 3)
val numbers1 = numbers.map { it * 3 }
val numbers2 =
    numbers.mapIndexed { idx, value -> value * idx }

val map: Map<String, Int>
    = strings.associateWith { it.length }
```

Итераторы над коллекциями

```
val numbers = listOf("one", "two", "three")

val numbersIterator = numbers.iterator()
while (numbersIterator.hasNext())
    println(numbersIterator.next())

for (item in numbers)
    println(item)

numbers.forEach { println(it) }
```

Преобразования коллекций. Mapping

```
val numbers1 = setOf(1, 2, 3)
val numbers2 = numbers1.map { it * 3 }
val numbers3
    = numbers1.mapIndexed
      { idx, value -> value * idx }

val numbers4
   = numbers1.mapNotNull
        { if ( it == 2) null else it * 3 }
val numbers5
   = numbers1.mapIndexedNotNull
     { idx, value ->
        if (idx == 0) null else value * idx }
```

Преобразования карт. Mapping

```
val numbersMap: Map<String, Int>
    = mapOf("key1" to 1,
            "key2" to 2,
            "key3" to 3)

val numbersMap1: Map<String, Int>
    = numbersMap.mapKeys
        { it.key.toUpperCase() }

val numbersMap2: Map<String, Int>
    = numbersMap.mapValues
        { it.value + it.key.length }
```

Преобразования коллекций. Zipping

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")
println(colors zip animals)

val twoAnimals = listOf("fox", "bear")
println(colors.zip(twoAnimals))
// [(red, fox), (brown, bear), (grey, wolf)]
// [(red, fox), (brown, bear)]

val numberPairs
    = listOf("one" to 1, "two" to 2, "three" to 3)
println(numberPairs.unzip())
// ([one, two, three], [1, 2, 3])
```

Преобразования коллекций. Association

```
val numbers = listOf("one", "two", "three")

println(numbers.associateWith { it.length })
// {one=3, two=3, three=5}

println(numbers.associateWith { it.length })
// {one=3, two=3, three=5}

println(numbers.associateBy
            { it.first().toUpperCase() })
println(numbers.associateBy(
    keySelector = { it.first().toUpperCase() },
    valueTransform = { it.length } ))
// {O=one, T=three, F=four}
// {O=3, T=5, F=4}
```

Преобразования коллекций. Flattening

```
val numberSets = listOf(setOf(1, 2, 3),
                        setOf(4, 5, 6),
                        setOf(1, 2))
println(numberSets.flatten())
// [1, 2, 3, 4, 5, 6, 1, 2]

class Data(val items : List<String>)

val dataObjects = listOf(
        Data(listOf("a", "b", "c")),
        Data(listOf("1", "2", "3"))
)
val items: List<String>
    = dataObjects.flatMap { it.items }
// [a, b, c, 1, 2, 3]
```

Фильтация коллекций. predicate

```
val numbers = listOf("one", "two", "three")
val longerThan3
    = numbers.filter { it.length > 3 }
println(longerThan3)

val numbersMap =
    mapOf("key1" to 1,
          "key2" to 2,
          "key3" to 3)
val filteredMap
    = numbersMap.filter
          { (key, value) ->
            key.endsWith("1") && value > 10}
println(filteredMap)
```

Фильтация коллекций. Partitioning

```
val numbers = listOf("one", "two", "three")
val (match, rest)
    = numbers.partition { it.length > 3 }

println(match)
println(rest)
// [three, four]
// [one, two]
```

Фильтация коллекций. Testing predicates

```
val numbers
    = listOf("one", "two", "three", "four")

println(numbers.any { it.endsWith("e") })
println(numbers.none { it.endsWith("a") })
println(numbers.all { it.endsWith("e") })

println(emptyList<Int>().all { it > 5 })
// true
// true
// false
// true
```

Операторы + и -

```
val numbers
    = listOf("one", "two", "three", "four")

val plusList = numbers + "five"
val minusList = numbers - listOf("three", "four")
println(plusList)
println(minusList)
// [one, two, three, four, five]
// [one, two]
```

Групирование. groupBy

```
val numbers
  = listOf("one", "two", "three",
           "four", "five")

println(numbers.groupBy
        { it.first().toUpperCase() })

println(numbers.groupBy(
        keySelector = { it.first() },
        valueTransform = { it.toUpperCase() }))

// {O=[one], T=[two, three], F=[four, five]}
// {o=[ONE], t=[TWO, THREE], f=[FOUR, FIVE]}
```

Восстановление частей коллекции. Slice

- Функция **slice** возвращает список элементов коллекции с заданными индексами.
- Индексы могут быть переданы как диапазон или как набор целочисленных значений.

```
val numbers = listOf("one", "two", "three",
                     "four", "five", "six")
println(numbers.slice(1..3))
println(numbers.slice(0..4 step 2))
println(numbers.slice(setOf(3, 5, 0)))
// [two, three, four]
// [one, three, five]
// [four, six, one]
```

Восстановление частей коллекции. take & drop

```
val numbers = listOf("one", "two", "three",
                     "four", "five", "six")
println(numbers.take(3))
println(numbers.takeLast(3))
println(numbers.drop(1))
println(numbers.dropLast(5))
// [one, two, three]
// [four, five, six]
// [two, three, four, five, six]
// [one]
```

Восстановление частей коллекции. takeWhile & dropWhile

```
val numbers = listOf("one", "two", "three",
                     "four", "five", "six")
println(numbers.takeWhile
            { !it.startsWith('f') })
println(numbers.takeLastWhile
            { it != "three" })
println(numbers.dropWhile
            { it.length == 3 })
println(numbers.dropLastWhile
            { it.contains('i') })
// [one, two, three]
// [four, five, six]
// [three, four, five, six]
// [one, two, three, four]
```

Восстановление частей коллекции. chunked

```
val numbers = (0..13).toList()
println(numbers.chunked(3))
// [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11], [12, 13]]

val numbers = (0..13).toList()
println(numbers.chunked(3) { it.sum() })
// [3, 12, 21, 30, 25]
```

Восстановление частей коллекции. windowed

```
val numbers = listOf("one", "two", "three",
                     "four", "five")
println(numbers.windowed(3))
// [[one, two, three], [two, three, four], [three, four, five]]

val numbers = (1..10).toList()
println(numbers.windowed(3, step = 2,
                         partialWindows = true))
println(numbers.windowed(3) { it.sum() })
// [[1, 2, 3], [3, 4, 5], [5, 6, 7], [7, 8, 9], [9, 10]]
// [6, 9, 12, 15, 18, 21, 24, 27]
```

Восстановление частей коллекции. zip

```
val numbers = listOf("one", "two", "three",
                     "four", "five")
println(numbers.zipWithNext())
println(numbers.zipWithNext()
          { s1, s2 -> s1.length > s2.length})
// [(one, two), (two, three), (three, four), (four, five)]
// [false, false, true, false]
```

Получение элемента. По позиции

```
val numbers = linkedSetOf("one", "two", "three",
                          "four", "five")
println(numbers.elementAt(3))

val numbersSortedSet = sortedSetOf("one", "two",
                                   "three", "four")
println(numbersSortedSet.elementAt(0))

println(numbers.first())
println(numbers.last())

println(numbers.elementAtOrNull(5))
println(numbers.elementAtOrElse(5)
    { index -> "The value for index $index is undefined"})
```

Получение элемента. По позиции

```
val numbers = listOf("one", "two", "three",
                     "four", "five", "six")
println(numbers.first { it.length > 3 })
println(numbers.last { it.startsWith("f") })

println(numbers.firstOrNull { it.length > 6 })

val numbers = listOf(1, 2, 3, 4)
println(numbers.find { it % 2 == 0 })
println(numbers.findLast { it % 2 == 0 })
```

Получение случайного элемента

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.random())
```

Проверка существования элемента

```
val numbers = listOf("one", "two", "three",
                     "four", "five", "six")
println(numbers.contains("four"))
println("zero" in numbers)

println(numbers.containsAll
                (listOf("four", "two")))
println(numbers.containsAll
                (listOf("one", "zero")))

println(numbers.isEmpty())
println(numbers.isNotEmpty())

val empty = emptyList<String>()
println(empty.isEmpty())
println(empty.isNotEmpty())
```

Упорядочение коллекции

```
val lengthComparator = Comparator
      { str1: String, str2: String
              -> str1.length - str2.length }
println(listOf("aaa", "bb", "c")
       .sortedWith(lengthComparator))
println(listOf("aaa", "bb", "c")
       .sortedWith(compareBy { it.length }))

val numbers = listOf("one", "two",
             "three", "four")

println("Sorted ascending: ${numbers.sorted()}")
println("Sorted descending: ${numbers.sortedDescending()}")
```

Собственный порядок коллекции

```
val numbers = listOf("one", "two",
                     "three", "four")

val sortedNumbers
   = numbers.sortedBy { it.length }
println("Sorted by length ascending: $sortedNumbers")

val sortedByLast
    = numbers.sortedByDescending { it.last() }
println("Sorted by the last letter descending: $sortedByLast")
```

Обратный порядок коллекции

```
val numbers = listOf("one", "two",
                     "three", "four")
println(numbers.reversed())

val reversedNumbers = numbers.asReversed()
println(reversedNumbers)

val reversedNumbers = numbers.asReversed()
println(reversedNumbers)

numbers.add("five")
println(reversedNumbers)
```

Случайный порядок коллекции

```
val numbers = listOf("one", "two",
                     "three", "four")
println(numbers.shuffled())
```

Агрегатные операции. min, sum, average

```
val numbers = listOf(6, 42, 10, 4)

println("Count: ${numbers.count()}")
println("Max: ${numbers.max()}")
println("Min: ${numbers.min()}")
println("Average: ${numbers.average()}")
println("Sum: ${numbers.sum()}")
// Count: 4
// Max: 42
// Min: 4
// Average: 15.5
// Sum: 62
```

Агрегатные операции. maxBy, maxWith

- **maxBy()/minBy()** берет функцию-селектор и возвращает элемент, для которого она возвращает наибольшее или наименьшее значение.
- **maxWith()/minWith()** берет объект ***Comparator*** и возвращает наибольший или наименьший элемент в полученный соответствии с параметром ***Comparator***.

```
   val numbers = listOf(5, 42, 10, 4)
val min3Remainder = numbers.minByOrNull { it % 3 }
println(min3Remainder)

val strings = listOf("one", "two",
   "three", "four")
val longestString
   = strings.maxWithOrNull(compareBy { it.length })
println(longestString)
// 42
// three
```

Агрегатные операции. sumBy, sumByDouble

```
val numbers = listOf(5, 42, 10, 4)
println(numbers.sumBy { it * 2 })
println(numbers.sumByDouble{ it.toDouble() / 2 })
// 122
// 30.5
```

Функции fold и reduce

```
val numbers = listOf(5, 2, 10, 4)
val sum = numbers.reduce
   { sum, element -> sum + element }
println(sum)
val sumDoubled = numbers.fold(0)
   { sum, element -> sum + element * 2 }
println(sumDoubled)
```

- Функции **reduce** и **fold** последовательно применяют предоставленную операцию к элементам коллекции и возвращают накопленный результат.
- первый аргумент это ранее накопленное значение
- второй аргумент это элемент коллекции.

- **fold** принимает начальное значение и использует его как накопленное значение на первом шаге
- **reduce** на первом шаге использует первый и второй элементы

Функции foldIndexed и foldRightIndexed

```
val numbers = listOf(5, 2, 10, 4)
val sumEven = numbers.foldIndexed(0)
   { idx, sum, element ->
     if (idx % 2 == 0) sum + element else sum }
println(sumEven)

val sumEvenRight = numbers.foldRightIndexed(0)
   { idx, element, sum ->
     if (idx % 2 == 0) sum + element else sum }
println(sumEvenRight)
```

- **reduceOrNull**
- **reduceRightOrNull**
- **reduceIndexedOrNull**
- **reduceRightIndexedOrNull**

Изменения в коллекции. Добавление

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
println(numbers)

val numbers = mutableListOf(1, 2, 5, 6)
numbers.addAll(arrayOf(7, 8))
println(numbers)
numbers.addAll(2, setOf(3, 4))
println(numbers)

val numbers = mutableListOf("one", "two")
numbers += "three"
println(numbers)
numbers += listOf("four", "five")
println(numbers)
```

Изменения в коллекции. Удаление(1)

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.remove(3) // Удаляется первая 3`
println(numbers)
numbers.remove(5) // Ничего не удаляется
println(numbers)

numbers.retainAll { it >= 3 }
println(numbers)
```

Изменения в коллекции. Удаление(2)

```
val numbersSet = mutableSetOf("one", "two",
                              "three", "four")
numbersSet.removeAll(setOf("one", "two"))
println(numbersSet)

val numbers = mutableListOf("one", "two",
                 "three", "three", "four")
numbers -= "three"
println(numbers)
numbers -= listOf("four", "five")
println(numbers)
```

Списки. Доступ по индексу

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.get(0))
println(numbers[0])
// numbers.get(5)                    // exception!
println(numbers.getOrNull(5))       // null
println(numbers.getOrElse(5, {it})) // 5

val numbers = (0..13).toList()
println(numbers.subList(3, 6))
```

Списки. Линейный поиск

```
val numbers = listOf(1, 2, 3, 4, 2, 5)
println(numbers.indexOf(2))
println(numbers.lastIndexOf(2))

val numbers = mutableListOf(1, 2, 3, 4)
println(numbers.indexOfFirst { it > 2})
println(numbers.indexOfLast { it % 2 == 1})
```

Списки. Двоичный поиск в сортированных списках

- Бинарный поиск - работает значительно быстрее линейного поиска
- Требуется чтобы список был отсортирован в порядке возрастания в соответствии с определенным порядком: естественным или другим, указанным в параметре функции.
- В противном случае результат не определен.

```
val numbers = mutableListOf("one", "two",
                            "three", "four")
numbers.sort()
println(numbers)
println(numbers.binarySearch("two"))  // 3
println(numbers.binarySearch("z")) // -5
println(numbers.binarySearch("two", 0, 2)) // -3
```

Списки. Двоичный поиск с компаратором

```
data class Product(val name: String,
                   val price: Double)
val productList = listOf(
    Product("WebStorm", 49.0),
    Product("AppCode", 99.0),
    Product("DotTrace", 129.0),
    Product("ReSharper", 149.0))

println(productList.binarySearch(
    Product("AppCode", 99.0),
    compareBy<Product> { it.price }
                       .thenBy { it.name }))
val colors = listOf("Blue", "green",
         "ORANGE", "Red", "yellow")
println(colors.binarySearch("RED",
     String.CASE_INSENSITIVE_ORDER)) // 3
```

Списки. Двоичный поиск с функцией сравнения

- Двоичный поиск с *функцией сравнения* позволяет находить элементы без предоставления явных значений поиска
- Вместо этого он принимает элементы функции сравнения, отображающие значения **Int**, и ищет элемент, в котором функция возвращает ноль
- Список должен быть отсортирован в порядке возрастания в соответствии с предоставленной функцией

```
fun priceComparison(product: Product,
                    price: Double)
   = sign(product.price - price).toInt()
val productList = listOf(
       Product("WebStorm", 49.0),
       Product("AppCode", 99.0),
       Product("DotTrace", 129.0),
       Product("ReSharper", 149.0))
println(productList.binarySearch
       { priceComparison(it, 99.0) })
```

Списки. Добавление и обновление

```
val numbers = mutableListOf("one",
                    "five", "six")
numbers.add(1, "two")
numbers.addAll(2, listOf("three", "four"))
println(numbers)

val numbers = mutableListOf("one",
                  "five", "three")
numbers[1] =  "two"
println(numbers)

val numbers = mutableListOf(1, 2, 3, 4)
numbers.fill(3)
println(numbers)
```

Списки. Удаление

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.removeAt(1)
println(numbers)

numbers.removeFirst()
numbers.removeLast()
println(numbers)

val empty = mutableListOf<Int>()
// empty.removeFirst()
// NoSuchElementException: List is empty.
empty.removeFirstOrNull() // null
```

Списки. Сортировка (1)

```
val numbers = mutableListOf("one", "two",
                            "three", "four")
numbers.sort()
println("Sort into ascending: $numbers")
numbers.sortDescending()
println("Sort into descending: $numbers")

numbers.sortBy { it.length }
println("Sort into ascending by length: $numbers")

numbers.sortByDescending { it.last() }
println("Sort into descending by the last letter: $numbers")
```

Списки. Сортировка (2)

```
numbers.sortWith(compareBy<String>
        { it.length }.thenBy { it })
println("Sort by Comparator: $numbers")

numbers.shuffle()
println("Shuffle: $numbers")

numbers.reverse()
println("Reverse: $numbers")
```

Множества. Функции union, intersect и subtract

```
val numbers = setOf("one", "two", "three")

println(numbers union setOf("four", "five"))
println(setOf("four", "five") union numbers)

println(numbers intersect setOf("two", "one"))
println(numbers subtract setOf("three", "four"))
println(numbers subtract setOf("four", "three"))
// [one, two, three, four, five]
// [four, five, one, two, three]
// [one, two]
// [one, two]
// [one, two]
```

Карты. Получение ключей и значений

- **getOrElse** работает так же, как и для списков: значения для несуществующих ключей возвращаются из данной лямбда-функции.
- **getOrDefault** возвращает указанное значение по умолчанию, если ключ не найден.

```
val numbersMap = mapOf("one" to 1,
                       "two" to 2,
                       "three" to 3)
println(numbersMap.get("one"))
println(numbersMap["one"])
println(numbersMap.getOrDefault("four", 10))
println(numbersMap["five"])   // null
// numbersMap.getValue("six") // exception!
println(numbersMap.keys)
println(numbersMap.values)
```

Карты. Фильтрация (1)

```
val numbersMap = mapOf("key1" to 1,
                       "key2" to 2,
                       "key3" to 3,
                       "key11" to 11)
val filteredMap = numbersMap.filter
   { (key, value) ->
      key.endsWith("1") && value > 10}
println(filteredMap)
// {key11=11}
```

Карты. Фильтрация (2)

```
val filteredKeysMap = numbersMap.filterKeys
                      { it.endsWith("1") }
val filteredValuesMap = numbersMap.filterValues
                       { it < 10 }
println(filteredKeysMap)
println(filteredValuesMap)
// {key1=1, key11=11}
// {key1=1, key2=2, key3=3}
```

Карты. Операции + и -

```
val numbersMap = mapOf("one" to 1,
                       "two" to 2,
                       "three" to 3)
println(numbersMap + Pair("four", 4))
println(numbersMap + Pair("one", 10))
println(numbersMap + mapOf("five" to 5,
                           "one" to 11))
// {one=1, two=2, three=3, four=4}
// {one=10, two=2, three=3}
// {one=11, two=2, three=3, five=5}

println(numbersMap - "one")
println(numbersMap - listOf("two", "four"))
// {two=2, three=3}
// {one=1, three=3}
```

Карты. Добавление и обновление (1)

```
val numbersMap = mutableMapOf("one" to 1,
                              "two" to 2)
numbersMap.put("three", 3)
println(numbersMap)
// {one=1, two=2, three=3}
numbersMap.putAll(setOf("four" to 4,
                        "five" to 5))
println(numbersMap)
// {one=1, two=2, three=3, four=4, five=5}
val previousValue = numbersMap.put("one", 11)
println("value associated with 'one'," +
        " before: $previousValue," +
        " after: ${numbersMap["one"]}")
println(numbersMap)
// value associated with 'one', before: 1, after: 11
// {one=11, two=2}
```

Карты. Добавление и обновление (2)

```
val numbersMap = mutableMapOf("one" to 1,
                              "two" to 2)
numbersMap["three"] = 3
// Аналог numbersMap.set("three", 3)

numbersMap += mapOf("four" to 4, "five" to 5)
println(numbersMap)
// {one=1, two=2, three=3, four=4, five=5}
```

Карты. Удаление (1)

```
val numbersMap = mutableMapOf("one" to 1,
                              "two" to 2,
                              "three" to 3)
numbersMap.remove("one")
println(numbersMap)
// {two=2, three=3}

numbersMap.remove("three", 4)
// Ничего не удаляется.
println(numbersMap)
// {two=2, three=3
```

Карты. Удаление (2)

```
val numbersMap = mutableMapOf("one" to 1,
                              "two" to 2,
                              "three" to 3,
                              "threeAgain" to 3)
numbersMap.keys.remove("one")
println(numbersMap)
// {two=2, three=3, threeAgain=3}

numbersMap.values.remove(3)
println(numbersMap)
// {two=2, threeAgain=3}
```

Карты. Удаление (3)

```
val numbersMap = mutableMapOf("one" to 1,
                              "two" to 2,
                              "three" to 3)
numbersMap -= "two"
println(numbersMap)
// {one=1, three=3}

numbersMap -= "five"
// Ничего не удаляется.
println(numbersMap)
// {one=1, three=3}
```

ПОСЛЕДОВАТЕЛЬНОСТИ (SEQUENCES)

Отличие коллекций и последовательностей.

- **Коллекции** - при работе с коллекциями функции немедленно создают промежуточные коллекции. Результат, полученныи на каждом промежуточном шаге, сразу же сохраняется во временном списке.
- **Последовательности (sequences)** - дают способ реализации таких вычислений, позволяющий избежать создания временных промежуточных объектов.
- Любую **коллекцию** можно преобразовать в **последовательность**, вызвав функцию-расширение **asSequence()**.
- Любую **последовательность** можно преобразовать в **коллекцию**, вызвав функцию-расширение **toList()**.
- Для **последовательности** все операции применяются к каждому элементу поочередно. Это означает, что некоторые элементы могут не подвергнуться преобразованию, если результат будет вычислен прежде, чем до них дойдет очередь.

Пример последовательности

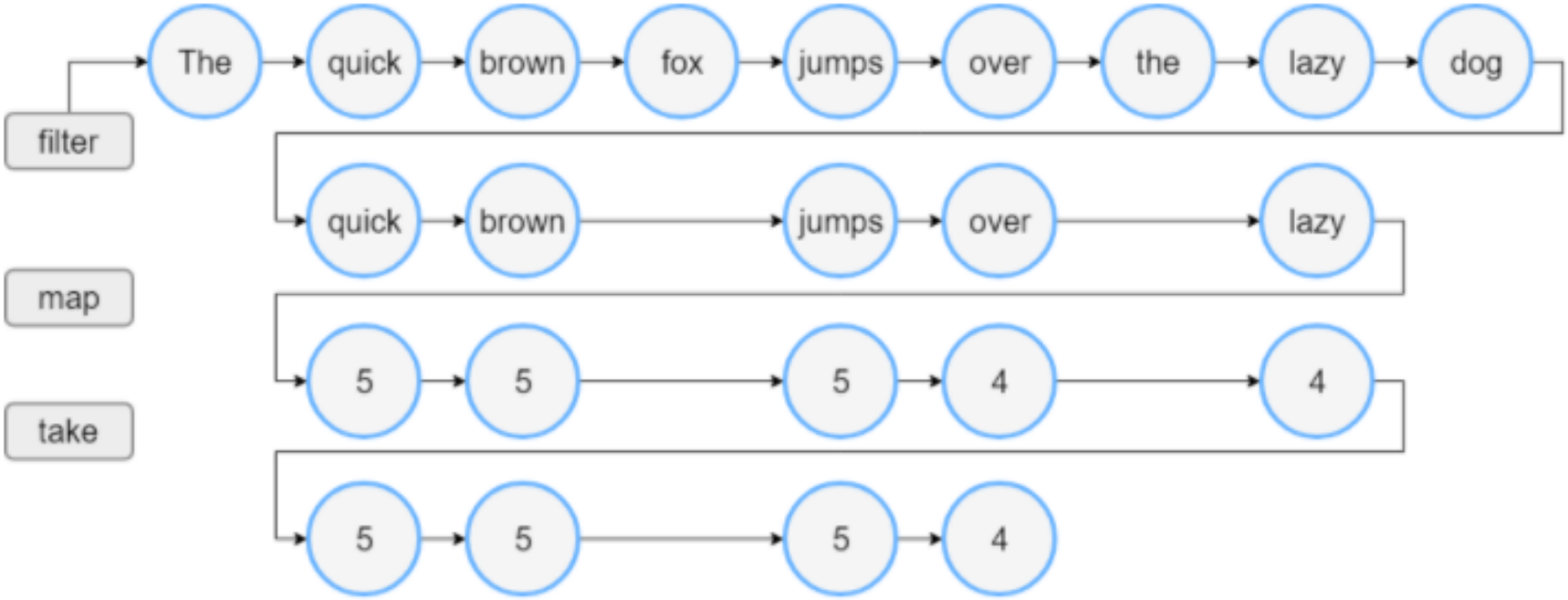
```
val lst = listOf(1, 2, 3, 4)
        .asSequence()
        .map { print("map($it) ");
              it * it }
        .filter { print("filter($it) ");
                 it % 2 == 0 }
        .toList()
// map(1) filter(1) map(2) filter(4) map(3) filter(9) map(4) filter(16)
```

Порядок обработки элементов в коллекции (текст)

```
val words =
    "The quick brown fox jumps over the lazy dog"
    .split(" ")
val lengthsList
    = words
    .filter { println("filter: $it");
              it.length > 3 }
    .map { println("length: ${it.length}");
           it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)
```

Порядок обработки элементов в коллекции (схема)



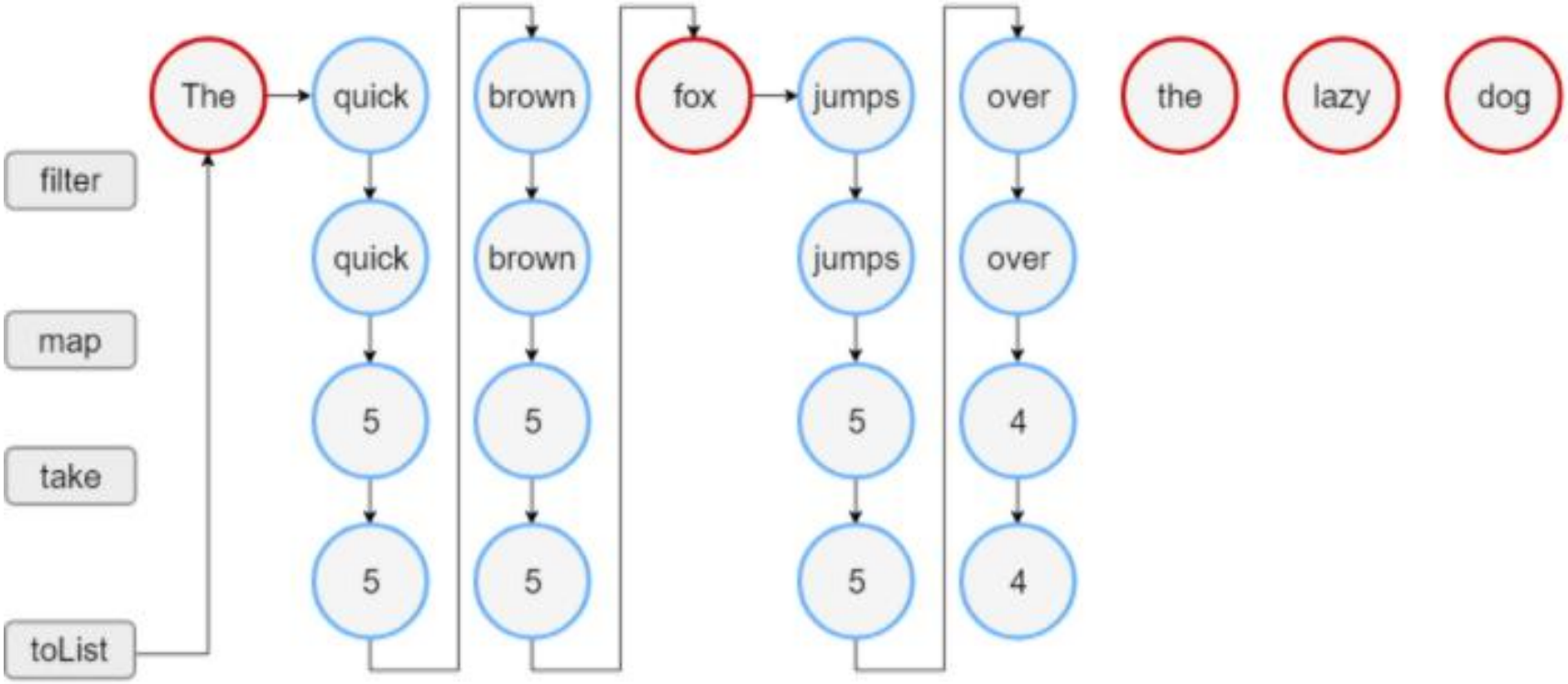
Порядок обработки элементов в коллекции. 23 шага

Порядок обработки элементов в последовательности (текст)

```
val words =
    "The quick brown fox jumps over the lazy dog"
    .split(" ")
val wordsSequence = words.asSequence()

val lengthsList =
    wordsSequence
    .filter { println("filter: $it");
              it.length > 3 }
    .map { println("length: ${it.length}");
           it.length }
    .take(4)
println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)
```

Порядок обработки элементов в последовательности (схема)



Порядок обработки элементов в последовательности. 18 шага

Последовательность. Промежуточная и завершающая операции

- *Промежуточная операция* возвращает другую последовательность, которая знает, как преобразовать элементы исходной последовательности.
- Выполнение *промежуточных операций* всегда откладывается.
- *Завершающая операция* возвращает результат, который может быть коллекцией, элементом, числом или объектом
- *Завершающая операция* заставляет выполниться все отложенные вычисления.

```
sequence
    .map { . . . }     // Промежуточная операция
    .filter { . . . ) // Промежуточная операция
    .toList()          // Завершающая операция
```

Конструирование последовательностей из элементов и итерируемых

```
val numbersSequence = sequenceOf("four", "three",
                                 "two", "one")

val numbers = listOf("one", "two",
                     "three", "four")
val numbersSequence = numbers.asSequence()
```

Конструирование последовательностей из функций

- Создание последовательности с помощью функции, вычисляющей ее элементы.
- Необходимо указать первый элемент как явное значение или результат вызова функции.
- Генерация последовательности останавливается, когда предоставленная функция возвращает значение **null**.

```
val oddNumbers
   = generateSequence(1) { it + 2 }
     // `it` это предыдущий элемент.
println(oddNumbers.take(5).toList())
// [1, 3, 5, 7, 9]

//println(oddNumbers.count())
// error: последовательность бесконечная.
```

Пример конечной последовательности

– Генерация последовательности останавливается, когда предоставленная функция возвращает значение **null**

```
val oddNumbersLessThan10
   = generateSequence(1)
      { if (it + 2 < 10) it + 2 else null }
println(oddNumbersLessThan10.count())
// 5
```

Конструирование последовательностей из кусков (1)

```
val oddNumbers = sequence {
    yield(1)
    yieldAll(listOf(3, 5))
    yieldAll(generateSequence(7) { it + 2 })
}
println(oddNumbers.take(5).toList())
// [1, 3, 5, 7, 9]
```

- функция **sequence()** позволяет создавать элементы последовательности один за другим или кусками произвольного размера
- Эта функция принимает *лямбда-выражение*, содержащее вызовы функций **yield** и **yieldAll**.

Конструирование последовательностей из кусков (2)

```
val oddNumbers = sequence {
    yield(1)
    yieldAll(listOf(3, 5))
    yieldAll(generateSequence(7) { it + 2 })
}
println(oddNumbers.take(5).toList())
// [1, 3, 5, 7, 9]
```

– Функции **yield** и **yieldAll** возвращают элемент потребителю последовательности и приостанавливают выполнение **sequence** до тех пор, пока потребитель не запросит следующий элемент.

Конструирование последовательностей из кусков (3)

```
val oddNumbers = sequence {
    yield(1)
    yieldAll(listOf(3, 5))
    yieldAll(generateSequence(7) { it + 2 })
}
println(oddNumbers.take(5).toList())
// [1, 3, 5, 7, 9]
```

- **yield** принимает в качестве аргумента единственный элемент;
- **yieldAll** может принимать объект **Iterable**, **Iterator** или другую последовательность.
- Аргумент **Sequence** у **yieldAll** может быть бесконечным. Однако такой вызов должен быть последним. все последующие вызовы никогда не будут выполнены.