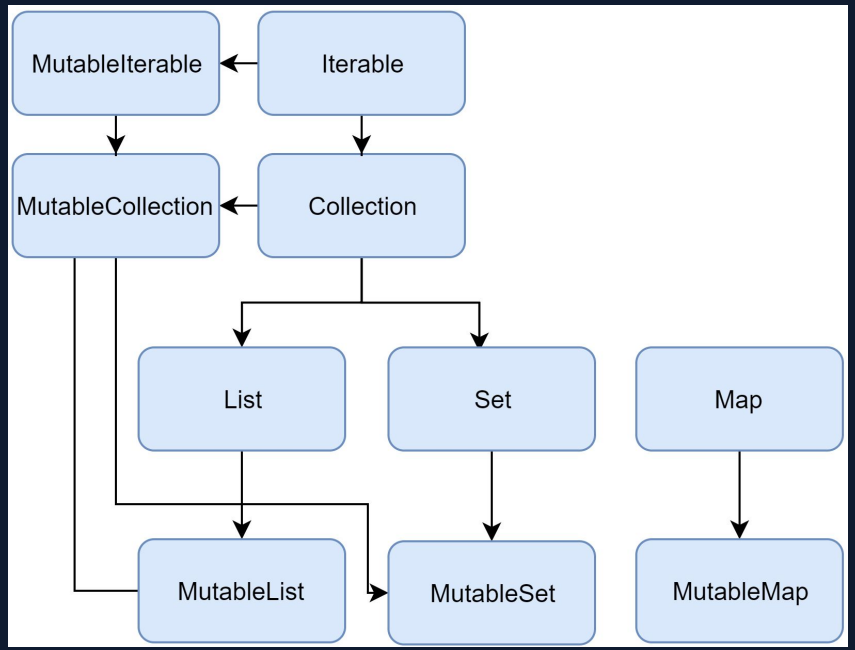
Collections

2020/12/15

Overview

A collection usually contains a number of objects (this number may also be zero) of the same type.



List

```
listOf(1, 2, 3, 4)
```

Set

```
setOf(1, 2, 3, 4)
```

Map

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

Basic Constructing

From elements

```
val numbersSet = setOf("one", "two", "three", "four")
```

Empty collections

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

Initializer for lists

```
val doubled = List(3, { it * 2 })
```

Concrete type

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))
val presizedSet = HashSet<Int>(32)
```

Copying

```
val sourceList = mutableListOf(1, 2, 3)
val referenceList = sourceList
referenceList.add(4)
println("Source size: ${sourceList.size}")  // Source size: 4
```

Invoking function on other collections

filter

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

map & mapIndexed

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 }) // [3, 6, 9]
println(numbers.mapIndexed { idx, value -> value * idx }) // [0, 2, 6]
```

associationWith

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
// {one=3, two=3, three=5, four=4}
```

Iterators

Iterators

- Set
- List

```
val numbers = listOf("one", "two", "three", "four")
val listIterator = numbers.listIterator()
while (listIterator.hasNext()) listIterator.next()
while (listIterator.hasPrevious()) {
    print("Index: ${listIterator.previousIndex()} , value: ${listIterator.previous()}")
}
```

Iterable

```
val numbers = listOf("one", "two", "three", "four")
for (item in numbers) { println(item) }
numbers.forEach { println(it) }
```

Mutable iterators

```
val numbers = mutableListOf("one", "four", "four")
val mutableListIterator = numbers.listIterator()
mutableListIterator.next()
mutableListIterator.remove()
mutableListIterator.add("two")
mutableListIterator.set("three")
```

Ranges and Progressions

rangeTo

```
var a: Int = 1
for (i in a.rangeTo(4)) println(i)
```

operation

- downTo
- step
- until

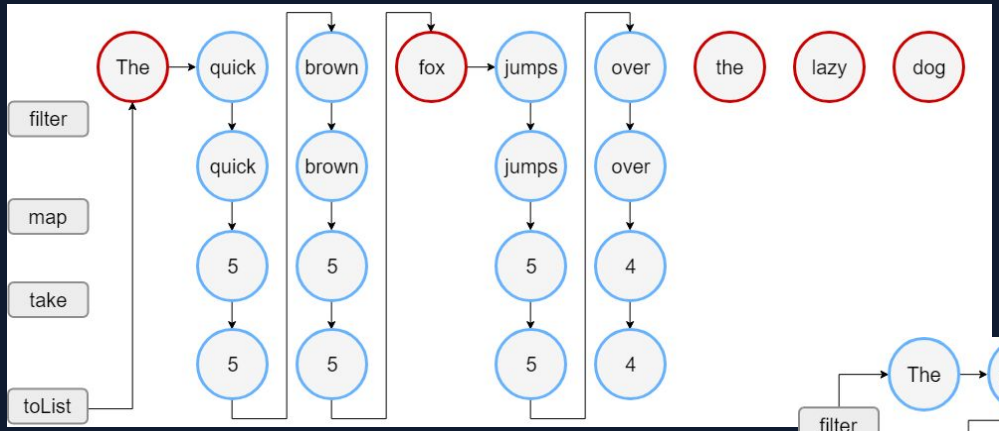
1. if (i in 1..4)
2. for (i in 1..4)
3. for (i in 4 downTo 1)
4. for (i in 1..8 step 2)
5. for (i in 8 downTo 1 step 2)
6. for (i in 1 until 5)

By multiple endpoint values

```
val versionRange = Version(1, 11)..Version(1, 30)
// override fun compareTo(other: Version): Int {
//     if (this.major != other.major) return this.major - other.major
//     else return this.minor - other.minor }
println(Version(0, 9) in versionRange) // false
println(Version(1, 20) in versionRange) // true
```

Sequences

Sequence type that represents lazily evaluated collections.



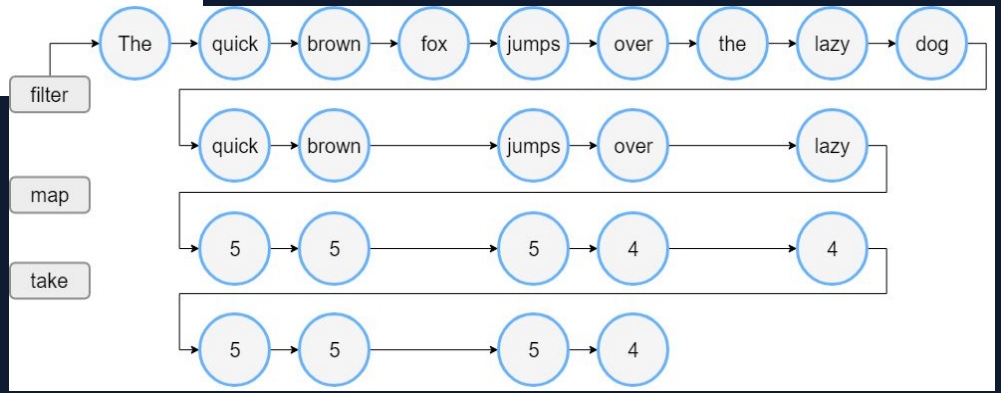
Iterable ▶️

```
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)
```

◀️ Sequence

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val wordsSequence = words.asSequence()
val lengthsSequence = 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")
// terminal operation: obtaining the result as a List
println(lengthsSequence.toList())
```



Sequences

From elements

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

From Iterable

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

From function

```
val oddNumbers = generateSequence(1) { it + 2 } // `it` is the previous element
println(oddNumbers.take(5).toList()) // 1, 3, 5, 7, 9

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

From chucks

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

Operations

▼ 참고 자료 ▼

https://medium.com/hongbeomi-dev/kotlin-collection-%ED%95%A8%EC%88%98-7a4b1290bce4

Mapping

map & mapIndexed

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 }) // [3, 6, 9]
println(numbers.mapIndexed { idx, value -> value * idx }) // [0, 2, 6]
```

mapNotNull & mapIndexedNotNull

```
val numbers = setOf(1, 2, 3)
println(numbers.mapNotNull { if ( it == 2) null else it * 3 }) // [3, 9]
println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value * idx })
// [2, 6]
```

mapKeys & mapValues

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
println(numbersMap.mapKeys { it.key.toUpperCase() })
// {KEY1=1, KEY2=2, KEY3=3, KEY11=11}
println(numbersMap.mapValues { it.value + it.key.length })
// {key1=5, key2=6, key3=7, key11=16}
```

Zipping

zip

- a.zip(b)
- a zip b

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

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

println(colors.zip(animals) { color, animal -> "The ${animal.capitalize()} is $color"})
// [The Fox is red, The Bear is brown, The Wolf is grey]
```

unzip

```
val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
println(numberPairs.unzip())
// ([one, two, three, four], [1, 2, 3, 4])
```

Association

associateWith

- original: key
- produced: value

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
// {one=3, two=3, three=5, four=4}
```

associateBy

- original: value
- produced: key

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

associate

```
val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")
println(names.associate { name -> parseFullName(name).let { it.lastName to
it.firstName } })
// {Adams=Alice, Brown=Brian, Campbell=Clara}
```

Flattening

flatten

```
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]
```

flatMap

```
val containers = listOf(
    StringContainer(listOf("one", "two", "three")),
    StringContainer(listOf("four", "five", "six")),
    StringContainer(listOf("seven", "eight"))
)
println(containers.flatMap { it.values })
// [one, two, three, four, five, six, seven, eight]
```

String representation

joinToString

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

println(numbers) // [one, two, three, four]
println(numbers.joinToString()) // one, two, three, four

println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": end"))
// start: one | two | three | four: end

println(numbers.joinToString(limit = 2, truncated = "<...>"))
// one, two, <...>

println(numbers.joinToString { "Element: ${it.toUpperCase()}"})
// Element: ONE, Element: TWO, Element: THREE, Element: FOUR
```

joinTo

```
val numbers = listOf("one", "two", "three", "four")
val listString = StringBuffer("The list of numbers: ")
numbers.joinTo(listString)
println(listString) // The list of numbers: one, two, three, four
```

Filtering

filter

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

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}
```

filterIndexed & filterNot

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

val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5)  }
val filteredNot = numbers.filterNot { it.length <= 3 }

println(filteredIdx) // [two, four]
println(filteredNot) // [three, four]
```

Filtering

filterIsInstance

```
val numbers = listOf(null, 1, "two", 3.0, "four")
numbers.filterIsInstance<String>().forEach {
    print( "${it.toUpperCase()} " )
}
// TWO FOUR
```

filterNotNull

```
val numbers = listOf(null, "one", "two", null)
numbers.filterNotNull().forEach {
    print( "${it.length} " )   // length is unavailable for nullable Strings
}
// 3 3
```

partition

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

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

Testing predicates

any: returns true if at least one element matches the given predicate.
none: returns true if none of the elements match the given predicate.
all: returns true if all elements match the given predicate.

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

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

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

```
val numbers = listOf("one", "two", "three", "four")
val empty = emptyList<String>()

println(numbers.any()) // true
println(empty.any()) // false

println(numbers.none()) // false
println(empty.none()) // true
```

Plus and Minus Operators

plus(+): the elements from the original collection and from the second operand.
minus(-): the elements of the original collection except the elements from the second operand.

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

val plusList = numbers + "five"
val minusList = numbers - listOf("three", "four")

println(plusList) // [one, two, three, four, five]
println(minusList) // [one, two]
```

Grouping

groupBy

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

println(numbers.groupBy { it.first().toUpperCase() })
// {O=[one], T=[two, three], F=[four, five]}
println(numbers.groupBy(keySelector = { it.first() },
                        valueTransform = { it.toUpperCase() }))
// {o=[ONE], t=[TWO, THREE], f=[FOUR, FIVE]}
```

groupBy operations

- eachCount
- fold
- reduce
- aggregate

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.groupingBy { it.first() }.eachCount())
// {o=1, t=2, f=2, s=1}
```

Retrieving Collection Parts

Slice

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

Chunked

```
val numbers = (0..13).toList()
println(numbers.chunked(3)) // [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11], [12, 13]]
println(numbers.chunked(3) { it.sum() }) // [3, 12, 21, 30, 25]
```

Windowed

```
val numbers = (1..10 step 3).toList()
println(numbers.windowed(3, step=2)) // [[1, 4, 7]]
println(numbers.windowed(3, step = 2, partialWindows = true)) // [[1, 4, 7], [7, 10]]
println(numbers.windowed(3) { it.sum() }) // [12, 21]
println(numbers.zipWithNext()) // [(1, 4), (4, 7), (7, 10)]
println(numbers.zipWithNext() { s1, s2 -> s1 * s2 }) // [4, 28, 70]
```

Retrieving Collection Parts

Take

- take
- takeLast
- takeWhile
- takeLastWhile

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

Drop

- drop
- dropLast
- dropWhile
- dropLastWhile

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.drop(1)) // [two, three, four, five, six]
println(numbers.dropLast(5)) // [one]
println(numbers.dropWhile { it.length == 3 }) // [three, four, five, six]
println(numbers.dropLastWhile { it.contains('i') }) // [one, two, three, four]
```

Retrieving Single Elements by position

elementAt

```
val numbers = linkedSetOf("one", "two", "three", "four", "five")
println(numbers.elementAt(3)) // four
val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
println(numbersSortedSet.elementAt(0)) // four
```

first & last

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.first()) // one
println(numbers.last()) // five
```

elementAtOrNull & elementAtOrElse

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.elementAtOrNull(5)) // null
println(numbers.elementAtOrElse(5) { index -> "$index"}) // 5
```

Retrieving Single Elements by condition

first & last

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

firstOrNull & lastOrNull

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.firstOrNull { it.length > 6 }) // null
```

find & findLast

- find - firstOrNull
- findLast - lastOrNull

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

Retrieving Single Elements

random & randomOrNull

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.random()) // 1
var emptys = emptyList<Int>()
println(emptys.randomOrNull()) // null
```

contains & containsAll

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

println(numbers.containsAll(listOf("four", "two"))) // true
println(numbers.containsAll(listOf("one", "zero"))) // false
```

isEmpty & isNotEmpty

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.isEmpty()) // false
println(numbers.isNotEmpty()) // true

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

Collection Ordering - Comparable

To define a natural order for a user-defined type, make the type an inheritor of *Comparable*. This requires implementing the compareTo() function.

- Positive values show that the receiver object is greater.
- Negative values show that it's less than the argument.
- Zero shows that the objects are equal.

```
class Version(val major: Int, val minor: Int):
Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        } else if (this.minor != other.minor) {
            return this.minor - other.minor
        } else return 0
    }
}
```

```
fun main() {
    println(Version(1, 2) > Version(1, 3)) // false
    println(Version(2, 0) > Version(1, 5)) // true
}
```

Collection Ordering - Comparable

To define a custom order for a type, create a *Comparator* for it.

A shorter way to define a Comparator is the compareBy() function from the standard library.

```
val lengthComparator = Comparator { str1: String, str2: String -> str1.length - str2.length }
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.sortedWith(lengthComparator)) // [one, two, six, four, five, three]
// Having the lengthComparator, you are able to arrange strings by their length
//                                                    instead of the default lexicographical order.
println(numbers.sortedWith(compareBy { it.length })) // [one, two, six, four, five, three]
```

Collection Ordering

sorted & sortedDescending

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

println("${numbers.sorted()}") // [four, one, three, two]
println("${numbers.sortedDescending()}") // [two, three, one, four]
```

sortedBy & sortedBy Descending

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

val sortedNumbers = numbers.sortedBy { it.length }
println("$sortedNumbers") // [one, two, four, three]
val sortedByLast = numbers.sortedByDescending { it.last() }
println("$sortedByLast") // [four, two, one, three]
```

sortedWith

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

val sortedWith = numbers.sortedWith(compareBy { it.length })
println("$sortedWith") // [one, two, four, three]
```

Collection Ordering

reversed

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

asReversed

- more lightweight and preferable
- original list is not going to change

```
val numbers = mutableListOf("one", "two", "three", "four")
val reversedNumbers = numbers.asReversed()

println("${numbers}") // [one, two, three, four]
println("${reversedNumbers}") // [four, three, two, one]

numbers.add("five")

println("${numbers}") // [one, two, three, four, five]
println("${reversedNumbers}") // [five, four, three, two, one]
```

shuffled

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

Collection Aggregate Operations

min & max

- min
- max
- minBy(*selector*)
- maxBy(*selector*)
- minWith(*Comparator*)
- maxWith(*Comparator*)

```
val numbers = listOf(5, 42, 10, 4)
val min3Remainder = numbers.minBy { it % 3 }
println("Max: ${numbers.max()} / Min: ${numbers.min()}") // Max: 42 / Min: 4
println("Min3Remainder: ${min3Remainder}") // Min3Remainder: 42

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

sum

- sum
- sumBy(*selector*)
- maxByDobule(*selector*)

```
val numbers = listOf(5, 42, 10, 4)
println(numbers.sum()) // 61
println(numbers.sumBy { it * 2 }) // 122
println(numbers.sumByDouble { it.toDouble() / 2 }) // 30.5
```

count & average

```
val numbers = listOf(5, 42, 10, 4)
println("Count: ${numbers.count()}") // Count: 4
println("Average: ${numbers.average()}") // Average: 15.5
```

Collection Aggregate Operations

fold

- fold
- foldRight
- foldIndexed
- foldRightIndexed

```
val numbers = listOf(5, 2, 10, 4)

val sumDoubled = numbers.fold(0) { sum, element -> sum + element * 2 }
println(sumDoubled) // 42

val sumDoubledRight = numbers.foldRight(0)
                        { element, sum -> sum + element * 2 }
println(sumDoubledRight) // 42

val sumEven = numbers.foldIndexed(0)
                        { idx, sum, element -> if (idx % 2 == 0) sum + element else sum }
println(sumEven) // 15

val sumEvenRight = numbers.foldRightIndexed(0)
                        { idx, element, sum -> if (idx % 2 == 0) sum + element else sum }
println(sumEvenRight) //15
```

Collection Aggregate Operations

reduce

- reduce
- reduceOrNull
- reduceRight
- reduceRightOrNull
- reduceIndexed
- reduceIndexedOrNull
- reduceRightIndexed
- reduceRightIndexed OrNull

```
val numbers = listOf(5, 2, 10, 4)

val sumDoubled = numbers.reduce() { sum, element -> sum + element * 2 }
println(sumDoubled) // 37

val sumDoubledRight = numbers.reduceRight()
                        { element, sum -> sum + element * 2 }
println(sumDoubledRight) // 38

val sumEven = numbers.reduceIndexed()
                        { idx, sum, element -> if (idx % 2 == 0) sum + element else sum }
println(sumEven) // 15

val sumEvenRight = numbers.reduceRightIndexed()
                        { idx, element, sum -> if (idx % 2 == 0) sum + element else sum }
println(sumEvenRight) //19
```

Collection Write Operations

For mutable collections, there are also *write operations* that change the collection state. Such operations include adding, removing, and updating elements.

For example, sort() sorts a mutable collection in-place, so its state changes; sorted() creates a new collection that contains the same elements in the sorted order.

```
val numbers = mutableListOf("one", "two", "three", "four")
val sortedNumbers = numbers.sorted()
println(numbers == sortedNumbers)  // false
numbers.sort()
println(numbers == sortedNumbers)  // true
```

Collection Write Operations

add

- add
- addAll
- plusAssign (+=)

```
val numbers = mutableListOf(1, 2)
numbers.add(5) // [1, 2, 5]
numbers.addAll(arrayOf(7, 8)) // [1, 2, 5, 7, 8]
numbers.addAll(2, setOf(3, 4)) // [1, 2, 3, 4, 5, 7, 8]
numbers += 9 // [1, 2, 3, 4, 5, 7, 8, 9]
numbers += listOf(0, 6) // [1, 2, 3, 4, 5, 7, 8, 9, 0, 6]
```

remove

- remove
- removeAll
- retainAll
- crear
- minusAssign (-=)

```
val numbers = mutableListOf(1, 2, 3, 4, 5, 6, 7)
numbers.remove(3) // [1, 2, 4, 5, 6, 7]
numbers.remove(3) // [1, 2, 4, 5, 6, 7]
numbers.retainAll { it >= 3 } // [4, 5, 6, 7]
numbers.removeAll(arrayOf(4, 5)) // [6, 7]
numbers -= 6 // [7]
numbers.clear() // []
```

Specific Operations

- List:
  - Document: List Specific Operations
  - Properties and Functions: List
- Set:
  - Document: Set Specific Operations
  - Properties and Functions: Set
- Map:
  - Document: Map Specific Operations
  - Properties and Functions: Map

Thank you!