{ this is Kotlin }

Collections

Tiberiu Tofan

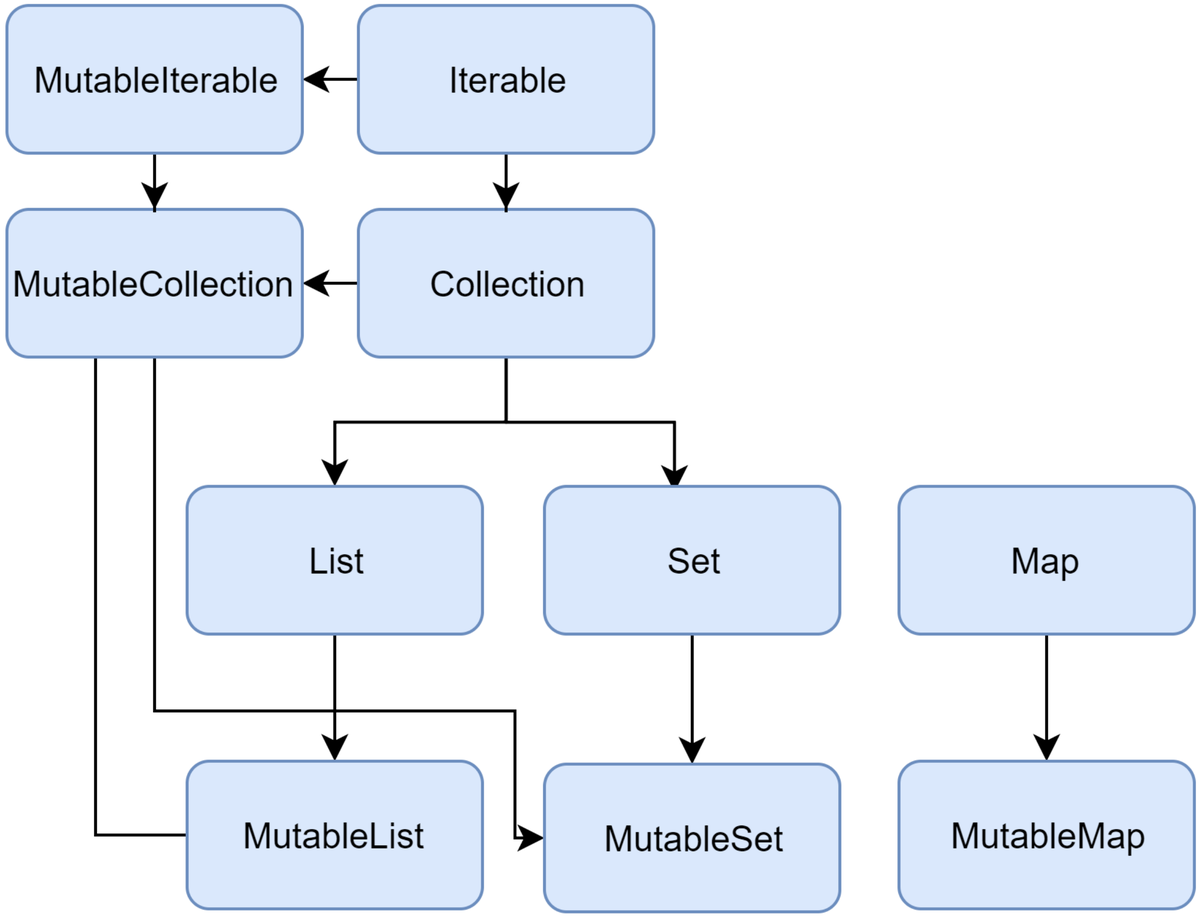
Collections

overview

- **List** is an ordered collection with access to elements by indices – integer numbers that reflect their position. Elements can occur more than once in a list.
- **Set** is a collection of unique elements. It reflects the mathematical abstraction of set: a group of objects without repetitions. Generally, the order of set elements has no significance.
- **Map** (or dictionary) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value. The values can be duplicates.

Collection Types

readonly or mutable



source: https://kotlinlang.org/docs/collections-overview.html?q=a#collection

List

overview

- ordered
- provided index access
- can have duplicates
- `List` - readonly
- `MutableList`
- default implementation: `ArrayList`

List

readonly

```
val ns: List<Int> = listOf(1, 2, 3, 4, 5, 6)
ns.add(7)
println(ns[2])
println(ns.javaClass)
val concat = ns + listOf(7, 8, 9)
println(concat)
```

compilation error: Unresolved reference: add

3

operator fun get

class java.util.Arrays$ArrayList

operator fun plus

[1, 2, 3, 4, 5, 6, 7, 8, 9]

concat is a new list

List
mutable

```
val ns: MutableList<Int> = mutableListOf(1, 2, 3, 4, 5, 6)
ns -= 6
ns[2] = 10
println(ns)
println(ns.javaClass)
```

operator fun minusAssign

operator fun set

[1, 2, 10, 4, 5]

class java.util.ArrayList

Set

overview

- unique elements
- order usually not
- do not provide index access
- `Set` - readonly
- `MutableSet`
- default implementation: `LinkedHashSet`

Set

```
val ns: Set<Int> = setOf(1, 2, 3, 4, 2)
println(ns) [1, 2, 3, 4]
println(ns.javaClass) class java.util.LinkedHashSet
```

Set

a set with nullable elements can have a single null value

```
val names: Set<String?> = mutableSetOf("Josh", "Martin", "Venkat", null, null)
println(names) [Josh, Martin, Venkat, null]

val otherNames = setOf("Andrei", "Roman")

val allNames = names union otherNames
println(allNames) [Josh, Martin, Venkat, null, Andrei, Roman]
```

Map

overview

- does not inherit `Collection`
- key-value pairs
- opitmized for access/searching by key
- `Map` is readonly
- `MutableMap`
- default implementation: `LinkedHashMap`

Pair

```
val p: Pair<Int, String> = Pair(1, "one")
val p: Pair<Int, String> = 1 to "one"

println("the pair is ${p.first} to ${p.second}")
```

the pair is 1 to one

destructure

```
val (n, word) = p

println("$n is $word")
```

1 is one

Map
readonly

```
val designers: Map<String, String> = mapOf(
    "Scala" to "Martin Odersky",
    "Java" to "James Gosling",
    "Kotlin" to "JetBrains",
    "Groovy" to "James Strachan",
    "Closure" to "Rich Hickey")

val designerOfScala = designers["Scala"]

val containsHaskell = "Haskell" in designers

println(designers.javaClass)
```

get operator

class java.util.LinkedHashMap

Map
mutable

```
val words = mutableMapOf(1 to "one", 2 to "two")
```

set operator

```
words[3] = "three"
```

plusAsign operator

```
words += 4 to "four"

println(words)
```

{1=one, 2=two, 3=three, 4=four}

Map
properties

```
val words = mutableMapOf(1 to "one", 2 to "two")
```

```
words.keys
words.values
words.entries
```

properties for the set of keys and the collection of values

Hashing

key in a HashMap

- must implement equals/hashCode, adhering to the contract
- hash collision - hashCode modulo map size is the same for two different values
- rehashing - when changing the size of the map, all hash-codes will be recomputed

Sets in relations to Maps

Most Set implementations are backed by a Map with a constant value

Immutability by Contract

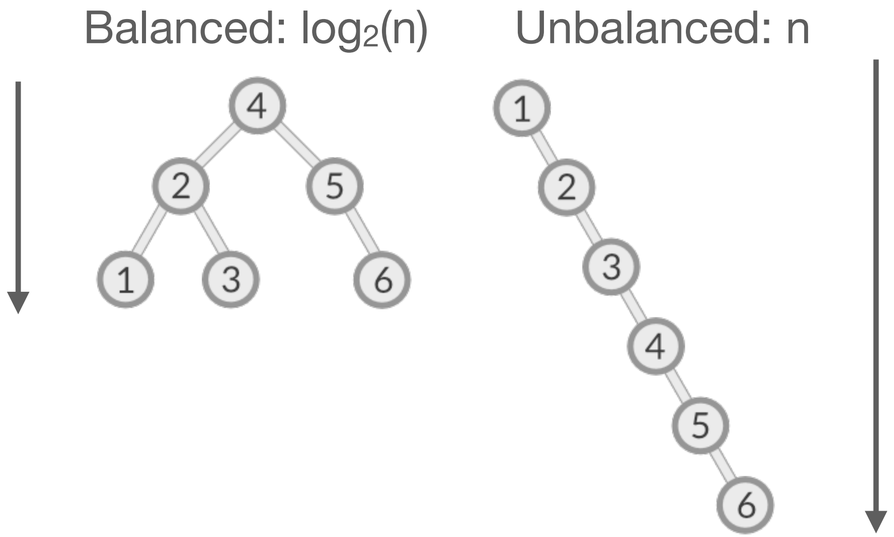
```
val immutable: Set<Int> = setOf(1, 2, 3, 4)
val mutable: MutableSet<Int> = immutable as MutableSet
mutable += 5
println(immutable) [1, 2, 3, 4, 5]
```

Try the same hack with `List` and `Map`. Observe the differences!

Time Complexities Refresh

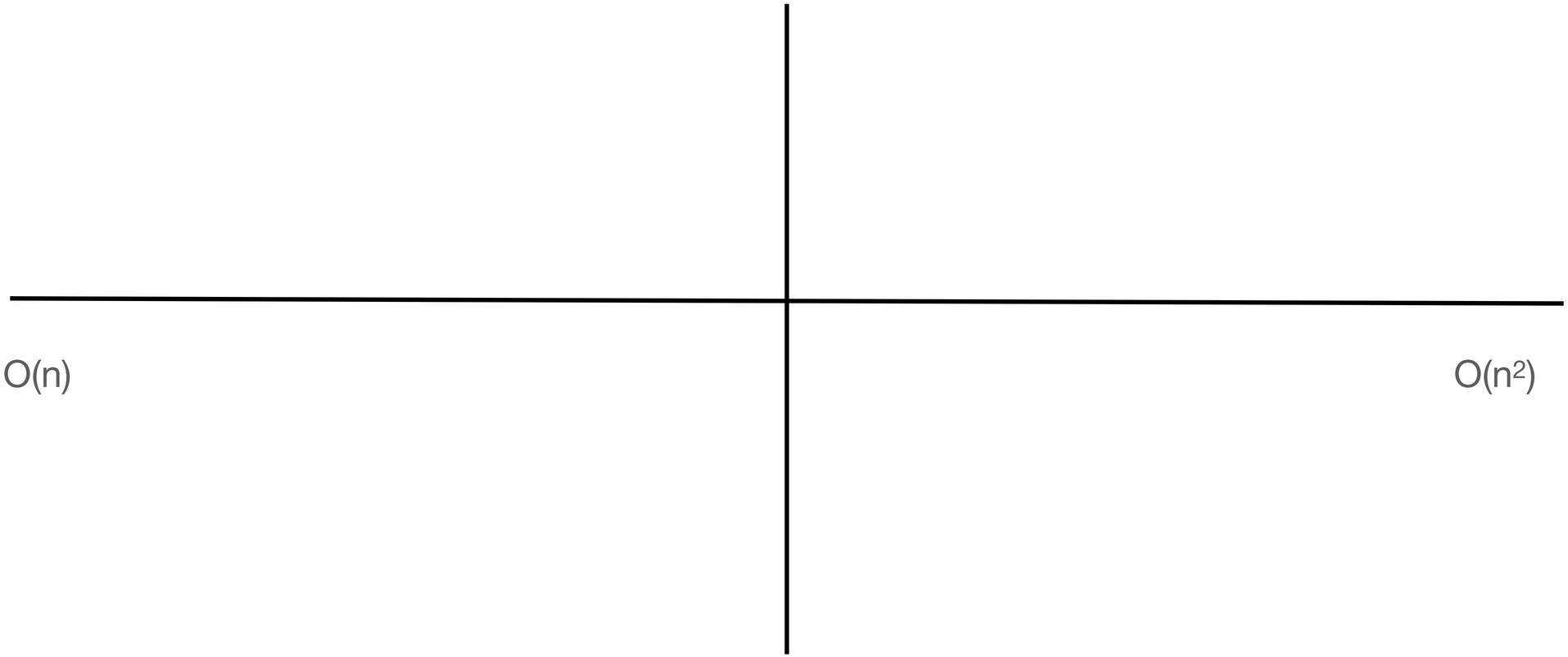
maximum number of operations that an algorithm may perform

- constant time: O(1)
  - ☑ constant number of operations
- logarithmic time: O(log n)
  - ☑ the depth of a balanced tree
- linear time: O(n)
  - ☑ the complexity of iterating through a list
- linearithmic time: O(n log n)
  - ☑ fastest sorting algorithms
- exponential time: $O(n^2)$, $O(n^3)$…
  - ☑ for in for



Time Complexities Game

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
```



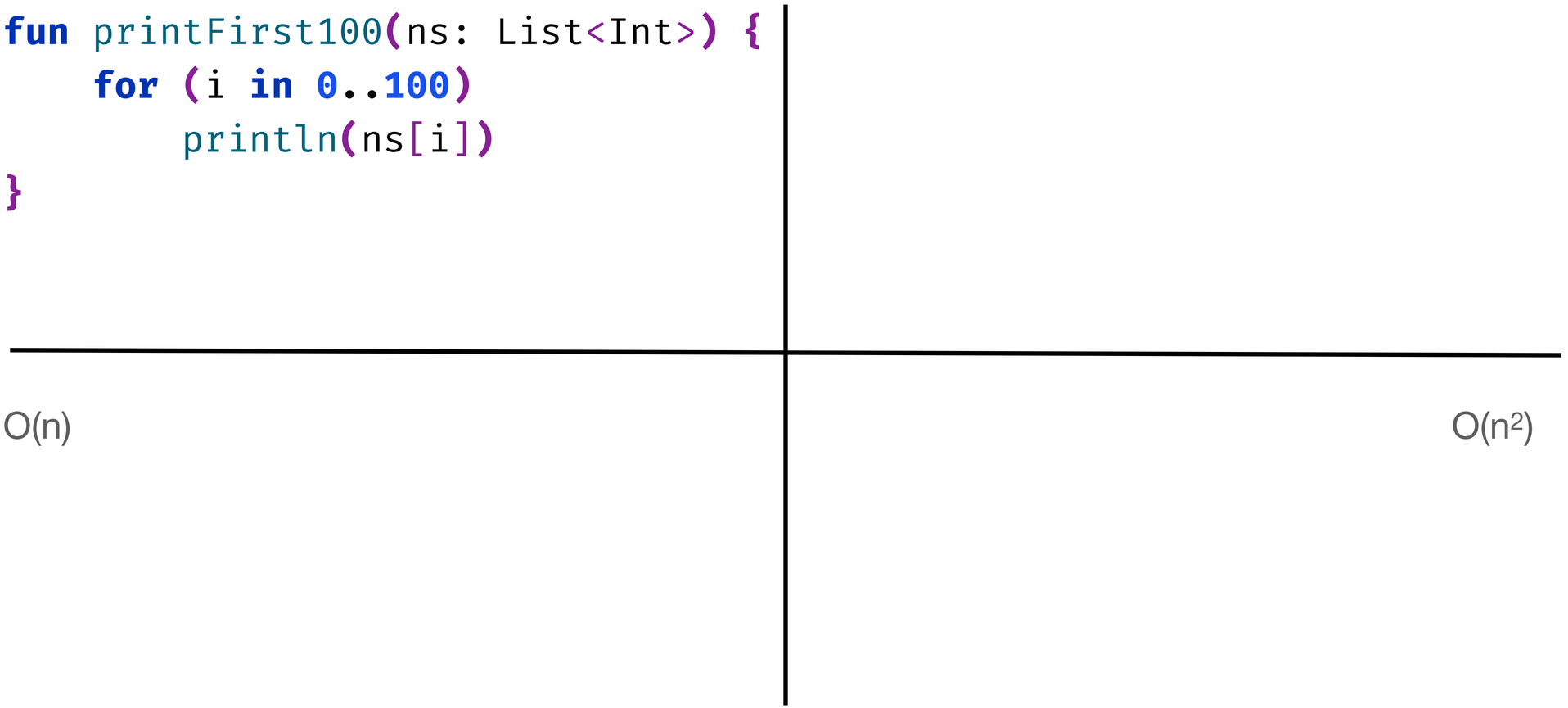
Time Complexities Game

```
treeSet += "a new element"
```

O(1)

O(log n)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
```



O(n)

O(n^2)

Time Complexities Game

list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Time Complexities Game

```
hashMap["key"]
```

O(1)

O(log n)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
```

```
treeSet += "a new element"
```

O(n)

O(n^2)

```
list.sorted()
```

list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Time Complexities Game

```
linkedList[index]
```

O(1)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
hashMap["key"]
```

O(log n)

```
treeSet += "a new element"
```

O(n)

O(n^2)

```
list.sorted()
```

list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Time Complexities Game

```
list.flatMap { list }
    .forEach { println(it) }
```

O(1)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
hashMap["key"]
```

O(log n)

```
treeSet += "a new element"
```

O(n)

```
linkedList[index]
```

O(n^2)

```
list.sorted()
```

list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Time Complexities Game

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

O(1)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
hashMap["key"]
```

O(log n)

```
treeSet += "a new element"
```

O(n)

```
linkedList[index]
```

O(n^2)

```
list.sorted()

list.flatMap { list }
    .forEach { println(it) }
```

list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Time Complexities Game

```
hashSet += "a new element"
```

O(1)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
hashMap["key"]
```

O(log n)

```
treeSet += "a new element"
```

O(n)

```
linkedList[index]
list.forEach { println(it) }
```

O(n^2)

```
list.sorted()
list.flatMap { list }
    .forEach { println(it) }
```

list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Time Complexities Game

```
treeMap["key"]
```

O(1)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
hashMap["key"]
hashSet += "a new element"
```

O(log n)

```
treeSet += "a new element"
```

O(n)

```
linkedList[index]
list.forEach { println(it) }
```

O(n^2)

```
list.sorted()
list.flatMap { list }
    .forEach { println(it) }
```

list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Time Complexities Game

```
//print a treeset sorted
```

O(1)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
hashMap["key"]
hashSet += "a new element"
```

O(log n)

```
treeSet += "a new element"
treeMap["key"]
```

O(n)

```
linkedList[index]
list.forEach { println(it) }
```

O(n^2)

```
list.sorted()
list.flatMap { list }
    .forEach { println(it) }
```

list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Time Complexities Game

O(1)

```
fun printFirst100(ns: List<Int>) {
    for (i in 0..100)
        println(ns[i])
}
hashMap["key"]
hashSet += "a new element"
```

O(log n)

```
treeSet += "a new element"
treeMap["key"]
```

O(n)

```
linkedList[index]
list.forEach { println(it) }
//print a treeset sorted
treeSet.forEach { println(it) }
```

O(n^2)

```
list.sorted()
list.flatMap { list }
    .forEach { println(it) }
```

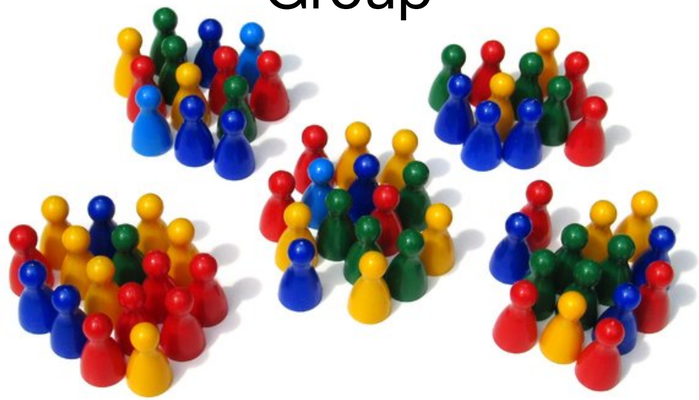
list.sorted() has a medium time complexity of O(n log n)
why is QuickSort being used and not a O(n log n) sorting algorithm?

Operations

Transform

Filter

Group



Order & sort

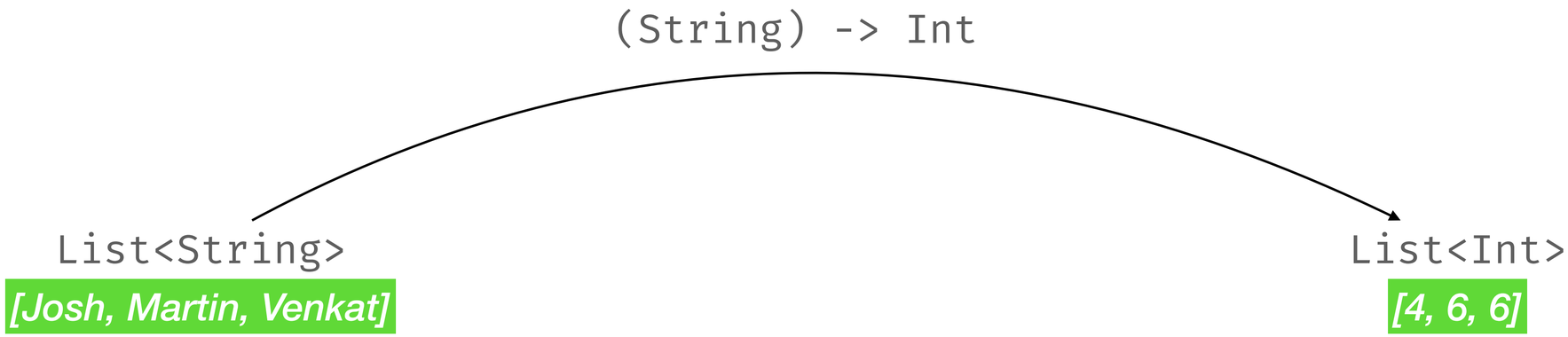
Combine

Transformations

map

```
val names = listOf("Josh", "Martin", "Venkat")
```

```
val lengths = names.map { it.length }
```



Transformations

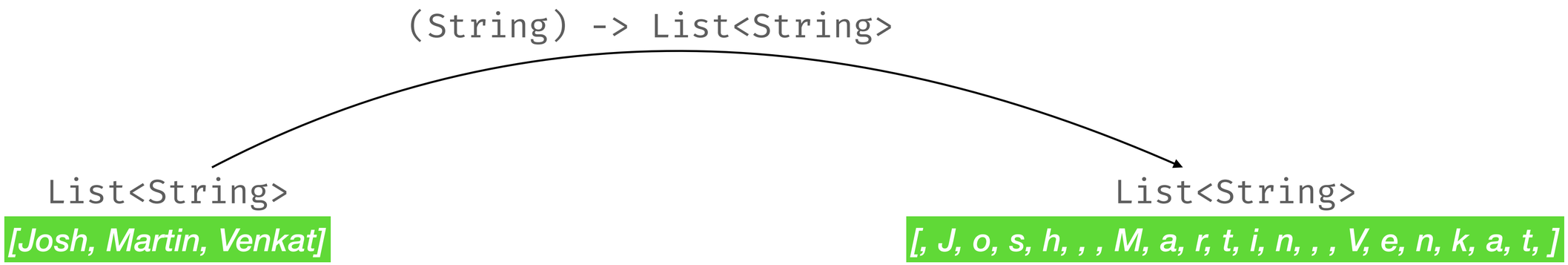
`flatMap`

```
val names = listOf("Josh", "Martin", "Venkat")
```

```
val chars = names.map { it.split( "") }
```

[[, J, o, s, h,], [, M, a, r, t, i, n,], [, V, e, n, k, a, t,]]

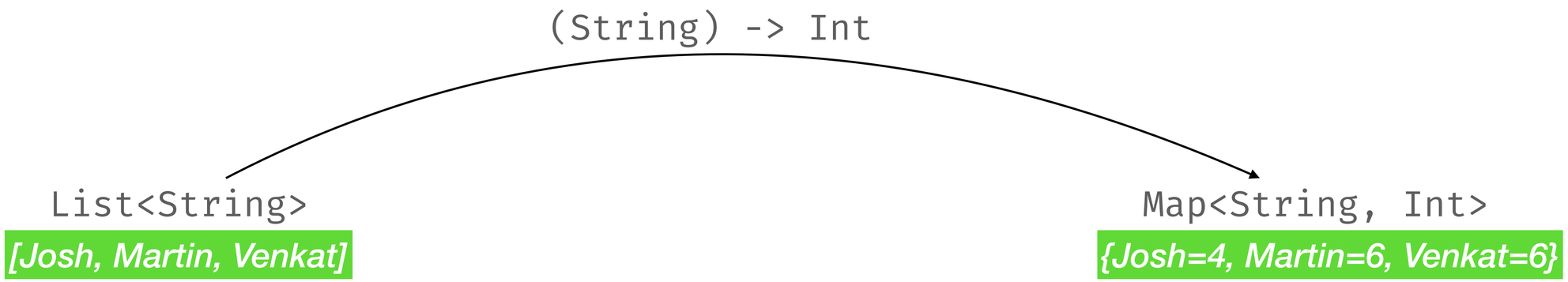
```
val chars = names.flatMap { it.split( "") }
```



Transformations

associate*

```
val names = listOf("Josh", "Martin", "Venkat")

val lengths = names.associateWith { it.length }
```

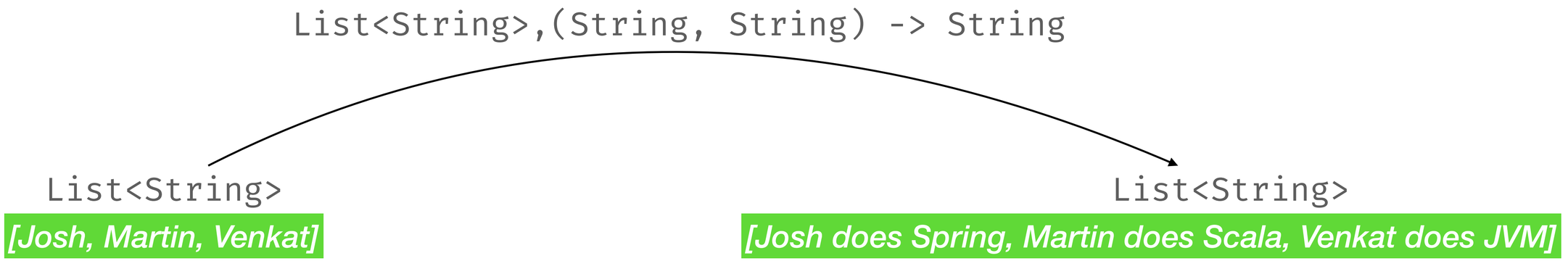


* other flavours available (associate, associateBy, associateTo, etc.) - check the docs

Transformations

zip

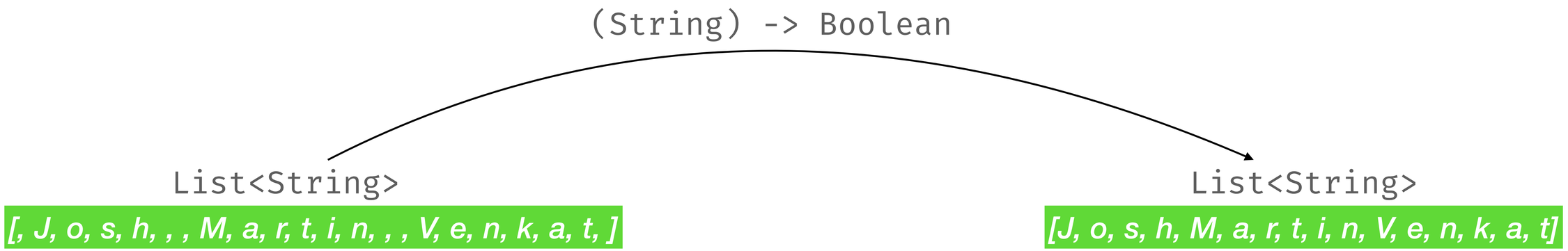
```
val names = listOf("Josh", "Martin", "Venkat")
val interests = listOf("Spring", "Scala", "JVM")


val texts = names.zip(interests) { n, i -> "$n does $i" }
```



Filtering

`filter`

```
val names = listOf("Josh", "Martin", "Venkat")
val interests = listOf("Spring", "Scala", "JVM")

val chars = names
    .flatMap { it.split("") }
    .filter { it.isNotEmpty() }
```

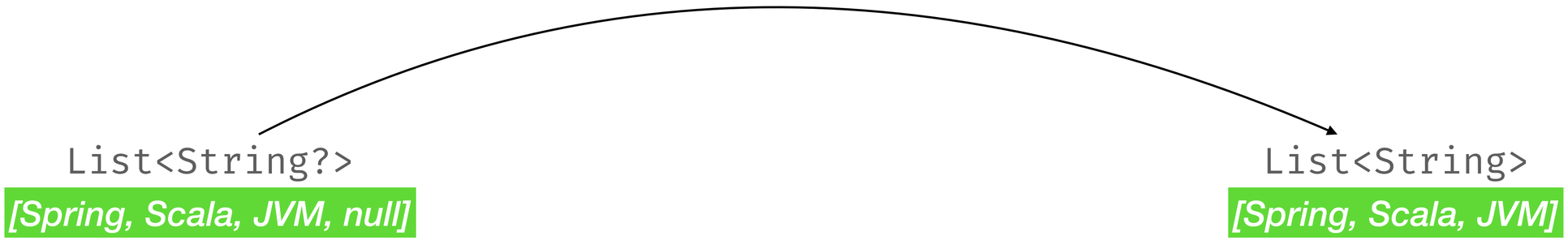


Filtering

`filterNotNull`

```
val interests = listOf("Spring", "Scala", "JVM", null)
```

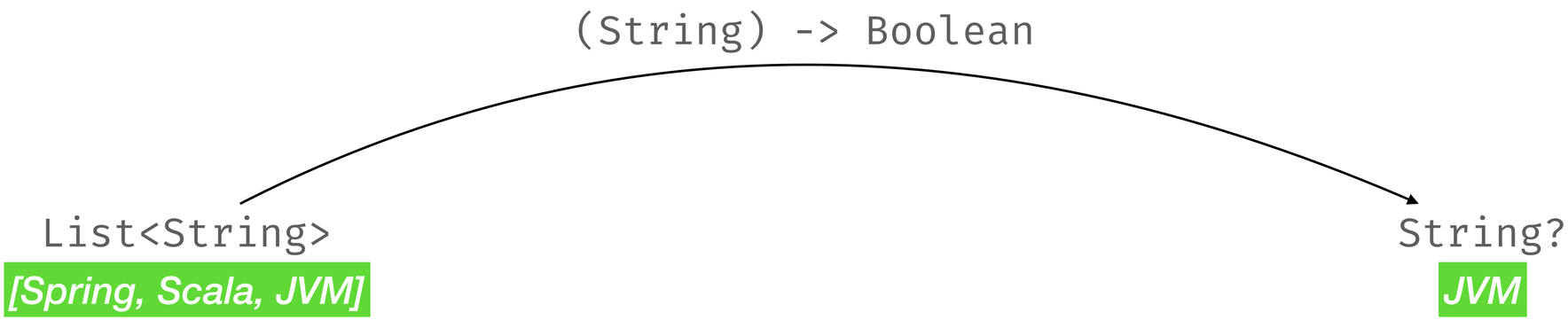
```
interests.filterNotNull()
```



Filtering

find

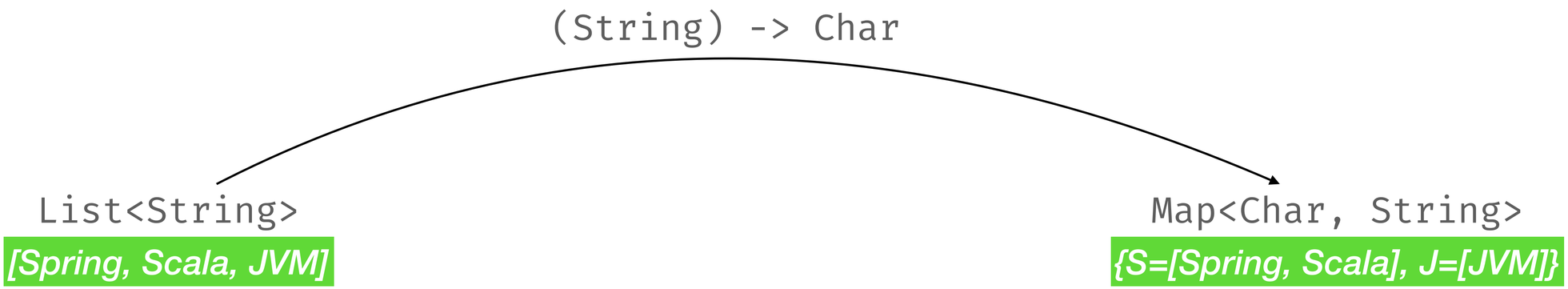
```
val interests = listOf("Spring", "Scala", "JVM")

interests.find { it.length < 4 }
```



Grouping

groupBy*

```
val interests = listOf("Spring", "Scala", "JVM")

val interestsByInitialisms = interests.groupBy { it.first() }
```

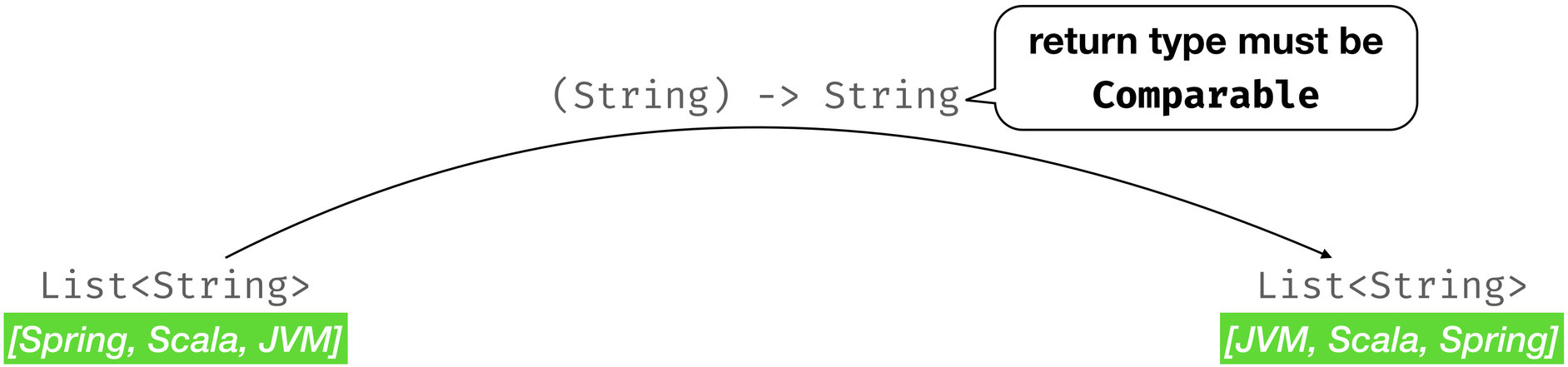


* other flavours available - check the docs

Sorting

sortedBy

```
val interests = listOf("Spring", "Scala", "JVM")

val sortedInterests = interests.sortedBy { it }
```

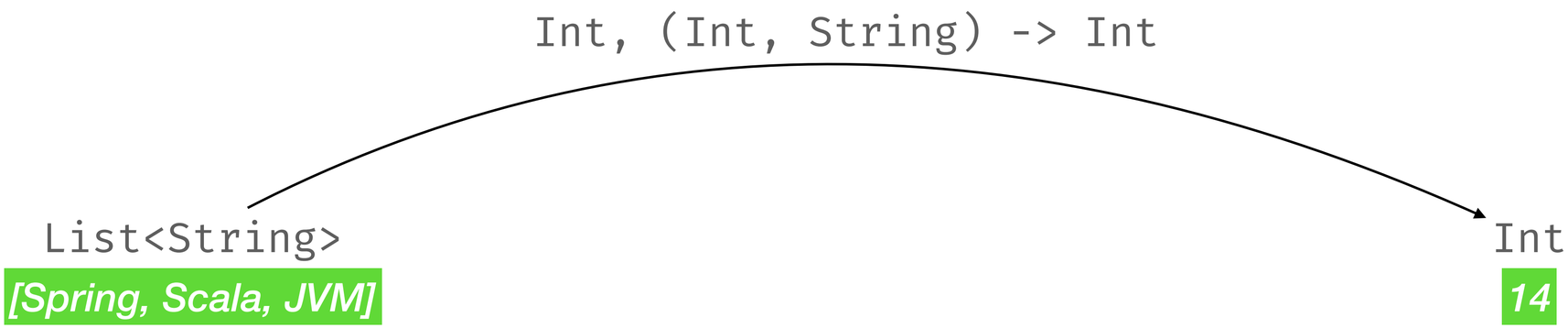


* other flavours available - check the docs

Combine

fold

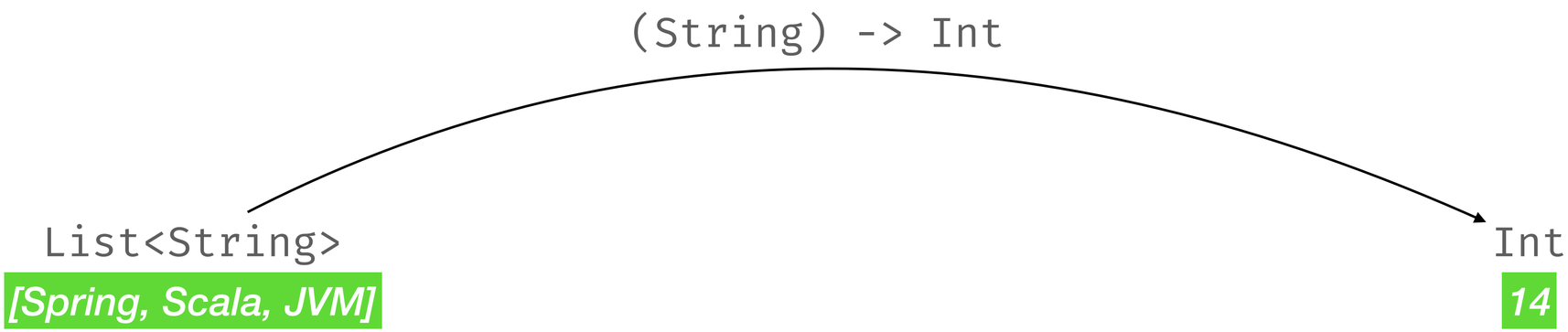
```
val interests = listOf("Spring", "Scala", "JVM")

val charCount = interests.fold(0) { sum, s -> sum + s.length }
```



Combine

sumOf

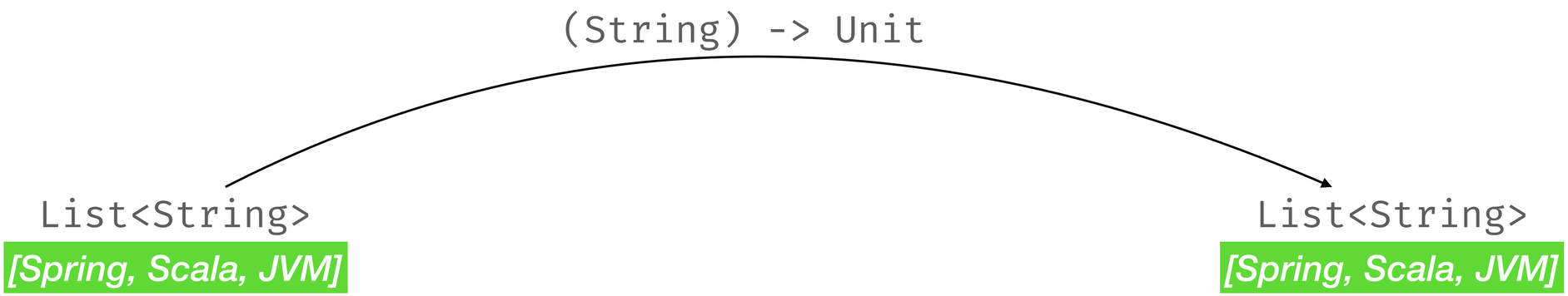
```
val interests = listOf("Spring", "Scala", "JVM")

val charCount = interests.sumOf { it.length }
```



Iteration

onEach

```
val interests = listOf("Spring", "Scala", “JVM")
                    .onEach { println(it) }
```

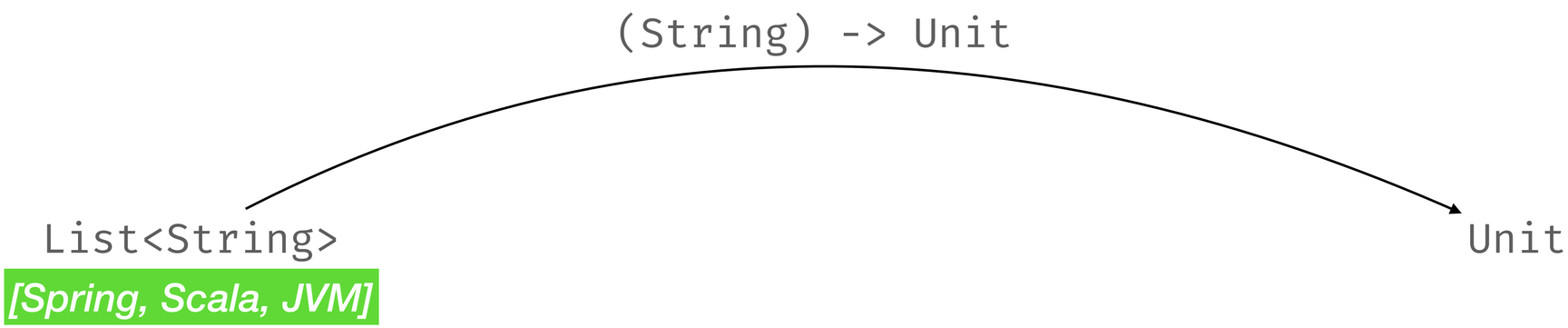


Iteration

forEach

```
val interests = listOf("Spring", "Scala", “JVM")
```

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



What is the major difference between the operations we just covered and Java Streams API?

Sequences
are lazy

```
val words: List<String> = listOf("tech", "radar", "on", "kotlin", "android", "101")
```

What is the length of the first palindrome?

```
words
    .onEach { println(it) }
    .filter { it.isPalindrome() }
    .map { it.length }
    .first()
```

tech
radar
on
kotlin
android
101

```
words.asSequence()
    .onEach { println(it) }
    .filter { it.isPalindrome() }
    .map { it.length }
    .first()
```

tech
radar

Sequences

are lazy

```
val words: List<String> = listOf("tech", "radar", "on", "kotlin", "android", "101")
```

```
val longWords = words
    .onEach { print(it) }
    .filter { it.length > 5 }
```

```
val longWords = words.asSequence()
    .onEach { print(it) }
    .filter { it.length > 5 }
```

```
longWords.forEach { print("$it ") }
longWords.forEach { print("$it ") }
```

Sequences

are lazy

```
val words: List<String> = listOf("tech", "radar", "on", "kotlin", "android", "101")
```

```
val longWords = words
    .onEach { print(it) }
    .filter { it.length > 5 }
```

```
val longWords = words.asSequence()
    .onEach { print(it) }
    .filter { it.length > 5 }
```

```
longWords.forEach { print("$it ") }
```

```
longWords.forEach { print("$it ") }
```

tech radar on kotlin android 101

kotlin android

kotlin android

Sequences

are lazy

```
val words: List<String> = listOf("tech", "radar", "on", "kotlin", "android", "101")
```

```
val longWords = words
    .onEach { print(it) }
    .filter { it.length > 5 }
```

```
val longWords = words.asSequence()
    .onEach { print(it) }
    .filter { it.length > 5 }
```

```
longWords.forEach { print("$it ") }
```

```
longWords.forEach { print("$it ") }
```

tech radar on kotlin android 101

kotlin android

kotlin android

tech radar on kotlin kotlin android android 101

tech radar on kotlin kotlin android android 101

Sequences
iterations

```kotlin
val words: List<String> = listOf("tech", "radar", "on", "kotlin", "android", "101")
```

- onEach - intermediate

```kotlin
words.asSequence()
    .onEach { println(it) }
```

without a terminal operation it doesn't print anything

- forEach - terminal

```kotlin
words.asSequence()
    .forEach { println(it) }
```

tech
radar
on
kotlin
android
101

Sequences

infinite sequences

```
fun generatePrimes(): Sequence<BigInteger> =
    generateSequence(1) { it + 1 }
        .map { it.toBigInteger() }
        .filter { it.isProbablePrime(100) }
```

- print the first n prime numbers

```
generatePrimes().take(n).forEach(::println)
```

- print the n^{th} prime number

```
var primeN = generatePrimes().drop(n - 1).first()
println("the ${n}th prime number is $primeN")
```

Sequences

operation types

- construct - from elements, chunks, lists, generators, etc.
- intermediate operations (produce another sequence): `map`, `flatMap`, `filter`, `take`, `takeWhile`, `drop`, `dropWhile`, `zip`, etc.
- terminal operations: `toList`, `first`, `last`, `single`, `any`, `all`, etc

- **lazy**
- **similar to Collections API**

- **a sequence is only computed when a terminal operation is called**
- **when the terminal operation finishes the evaluation of the sequence stops**

Sequences
from Collections

- reuse most of the intermediate operation
- just add `.asSequence` & a terminal operation

Persistent Data Structures

https://github.com/Kotlin/kotlinx.collections.immutable

Questions?