Scala School

Лекция 6: Методы коллекций

Views

```
> Seq.fill(5)(0).map(_.toString)
res1: Seq[String] = List(0, 0, 0, 0, 0)

> Seq.fill(5)(0).map(_.toString).view
res2: scala.collection.SeqView[String,Seq[String]] = SeqView(...)

> Seq.fill(5)(0).map(_.toString).view.force
res3: Seq[String] = List(0, 0, 0, 0, 0)

> Seq.fill(5)(0).map(_.toString).view.toList
res4: List[String] = List(0, 0, 0, 0, 0)
```

Views

```
> Seq.fill(5)(0).map(_.toString).view.map(_ + "!")
res44: scala.collection.SeqView[String,Seq[_]] = SeqViewM(...)

> Seq.fill(5)(0).map(_.toString).view.map(_ + "!").filter(_.length >= 2).map(_.length * 2).map(_ * 2)
res45: scala.collection.SeqView[Int,Seq[_]] = SeqViewMFMM(...)

> Seq.fill(5)(0).map(_.toString).view.map(_ + "!").filter(_.length >= 2).map(_.length * 2).map(_ *
2).force
res46: Seq[Int] = List(8, 8, 8, 8, 8)
```

Traversable methods

`xs foreach f` Executes function `f` for every element of `xs`.

```
> Traversable(1,2,3).foreach(println)
1
2
3
> Traversable(1,2,3).foreach(println(_))
1
2
3
> Traversable(1,2,3).foreach(i => println(i))
1
2
3
```

Traversable methods

`xs ++ ys` A collection consisting of the elements of both xs and ys. ys is a TraversableOnce collection, i.e., either a Traversable or an Iterator.

```
> Traversable(1,2,3) ++ Traversable(4,5,6)
res55: Traversable[Int] = List(1, 2, 3, 4, 5, 6)
```

Traversable methods

`xs map f` The collection obtained from applying the function f to every element in `xs`.

```
> Traversable(Option(1),None,Option(3)).map(_.map(_.toString))
res58: Traversable[Option[String]] = List(Some(1), None, Some(3))
```

Traversable methods

`xs flatMap f` The collection obtained from applying the collection-valued function `f` to every element in `xs` and concatenating the results.

```
> Traversable(Option(1),None,Option(3)).flatMap(_.map(_.toString))
res59: Traversable[String] = List(1, 3)

> Traversable(Option(1),None,Option(3)).map(_.map(_.toString)).flatten
res60: Traversable[String] = List(1, 3)
```

Traversable methods

`xs collect f` The collection obtained from applying the partial function `f` to every element in `xs` for which it is defined and collecting the results.

```
> Traversable(1,2,3,4,5).collect { case i if i > 2 => i * 2 }
res65: Traversable[Int] = List(6, 8, 10)

> Traversable(1,2,3,4,5).collectFirst { case i if i > 2 => i * 2 }
res66: Option[Int] = Some(6)
```

Traversable methods

Conversions:

`xs.toArray`	Converts the collection to an array.
`xs.toList`	Converts the collection to a list.
`xs.toIterable`	Converts the collection to an iterable.
`xs.toSeq`	Converts the collection to a sequence.
`xs.toIndexedSeq`	Converts the collection to an indexed sequence.
`xs.toStream`	Converts the collection to a lazily computed stream.
`xs.toSet`	Converts the collection to a set.
`xs.toMap`	Converts the collection of key/value pairs to a map. If the collection does not have pairs as elements, calling this operation results in a static type error.

Traversable methods

`xs collect f` The collection obtained from applying the partial function `f` to every element in `xs` for which it is defined and collecting the results.

```
> Traversable(1,2,3,4,5).collect { case i if i > 2 => i * 2 }
res65: Traversable[Int] = List(6, 8, 10)

> Traversable(1,2,3,4,5).collectFirst { case i if i > 2 => i * 2 }
res66: Option[Int] = Some(6)
```

Traversable methods

`xs copyToBuffer buf` Copies all elements of the collection to buffer `buf`.

```
> import scala.collection.mutable._
import scala.collection.mutable._
> val b = new ArrayBuffer[Int]
b: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()
> Traversable(1,2,3,4,5) copyToBuffer (b)
> b
res74: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4, 5)
```

Traversable methods

`xs copyToArray(arr, s, n)` Copies at most `n` elements of the collection to array `arr` starting at index `s`. The last two arguments are optional.

```
> val a = new Array[Int](10)
a: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

> Traversable(1,2,3,4,5) copyToArray (a)

> a
res99: Array[Int] = Array(1, 2, 3, 4, 5, 0, 0, 0, 0, 0)
```

Traversable methods

Size info:

`xs.isEmpty`	Tests whether the collection is empty.
`xs.nonEmpty`	Tests whether the collection contains elements.
`xs.size`	The number of elements in the collection.
`xs.hasDefiniteSize`	True if `xs` is known to have finite size.

```
> a.isEmpty
res100: Boolean = false
> a.nonEmpty
res101: Boolean = true
> a.hasDefiniteSize
res102: Boolean = true
> Stream(1,2,3).hasDefiniteSize
res103: Boolean = false
```

Traversable methods

Element Retrieval:

`xs.head`	The first element of the collection (or, some element, if no order is defined).
`xs.headOption`	The first element of `xs` in an option value, or None if `xs` is empty.
`xs.last`	The last element of the collection (or, some element, if no order is defined).
`xs.lastOption`	The last element of `xs` in an option value, or None if `xs` is empty.

```
> Traversable(1,2,3,4).head
res107: Int = 1
> Traversable(1,2,3,4).headOption
res108: Option[Int] = Some(1)
> Traversable().headOption
res109: Option[Nothing] = None
> Traversable.empty[Int].headOption
res110: Option[Int] = None
> Traversable(1,2,3,4).last
res112: Int = 4
```

Traversable methods

`xs find p` An option containing the first element in `xs` that satisfies `p`, or `None` if no element qualifies.

```
> Traversable(1,2,3,4).find(_ > 1000)
res114: Option[Int] = None

> Traversable(1,2,3,4).find(_ > 3)
res115: Option[Int] = Some(4)
```

Traversable methods

`xs.tail` The rest of the collection except `xs.head`.

```
> Traversable(1,2,3,4).tail
res116: scala.collection.mutable.Traversable[Int] = ArrayBuffer(2, 3, 4)
```

Traversable methods

`xs.init` The rest of the collection except `xs.last`.

```
> Traversable(1,2,3,4).tail
res116: scala.collection.mutable.Traversable[Int] = ArrayBuffer(2, 3, 4)
```

Traversable methods

`xs slice (from, to)` A collection consisting of elements in some index range of `xs` (from `from` up to, and excluding `to`).

```
> Traversable(1,2,3,4) slice (1,3)
res120: scala.collection.mutable.Traversable[Int] = ArrayBuffer(2, 3)
```

Traversable methods

`xs take n` A collection consisting of the first `n` elements of `xs` (or, some arbitrary `n` elements, if no order is defined).

```
> Traversable(1,2,3,4) take 2
res122: scala.collection.mutable.Traversable[Int] = ArrayBuffer(1, 2)
```

Traversable methods

`xs drop n` The rest of the collection except `xs take n`.

```
> Traversable(1,2,3,4) drop 2
res123: scala.collection.mutable.Traversable[Int] = ArrayBuffer(3, 4)
```

Traversable methods

`xs takeWhile p` The longest prefix of elements in the collection that all satisfy `p`.

```
> Traversable(1,2,3,4) takeWhile (_ < 3)
res125: scala.collection.mutable.Traversable[Int] = ArrayBuffer(1, 2)
> Traversable(1,2,3,4) takeWhile (_ > 3)
res126: scala.collection.mutable.Traversable[Int] = ArrayBuffer()
```

Traversable methods

`xs dropWhile p` The collection without the longest prefix of elements that all satisfy `p`.

```
> Traversable(1,2,3,4) dropWhile (_ < 3)
res127: scala.collection.mutable.Traversable[Int] = ArrayBuffer(3, 4)
> Traversable(1,2,3,4) dropWhile (_ > 3)
res128: scala.collection.mutable.Traversable[Int] = ArrayBuffer(1, 2, 3, 4)
```

Traversable methods

`xs filter p` The collection consisting of those elements of xs that satisfy the predicate `p`.

```
> Traversable(1,2,3,4) filter (_ > 3)
res152: scala.collection.mutable.Traversable[Int] = ArrayBuffer(4)
```

Traversable methods

`xs withFilter p` A non-strict filter of this collection. Subsequent calls to `map`, `flatMap`, `foreach`, and `withFilter` will only apply to those elements of `xs` for which the condition `p` is true.

```
> val a = Traversable(1,2,3,4).withFilter(_ > 3)
a: scala.collection.generic.FilterMonadic[Int,scala.collection.mutable.Traversable[Int]] =
scala.collection.TraversableLike$WithFilter@1063386
> a.map(_.toString)
res156: scala.collection.mutable.Traversable[String] = ArrayBuffer(4)
```

Traversable methods

`xs filterNot p`	The collection consisting of those elements of `xs` that do not satisfy the predicate `p`.

```
> Traversable(1,2,3,4).filterNot(_ > 2)
res157: scala.collection.mutable.Traversable[Int] = ArrayBuffer(1, 2)
```

Traversable methods

`xs splitAt n` Split `xs` at a position, giving the pair of collections `(xs take n, xs drop n)`.

```
> Traversable(1,2,3,4).splitAt(3)
res158: (scala.collection.mutable.Traversable[Int], scala.collection.mutable.Traversable[Int]) = (ArrayBuffer(1, 2,
3),ArrayBuffer(4))
> Traversable(1,2,3,4).splitAt(1)
res159: (scala.collection.mutable.Traversable[Int], scala.collection.mutable.Traversable[Int]) =
(ArrayBuffer(1),ArrayBuffer(2, 3, 4))
```

Traversable methods

`xs span p` Split `xs` according to a predicate, giving the pair of collections `(xs takeWhile p, xs.dropWhile p)`.

```
> Traversable(1,2,3,4) span (_ > 2)
res161: (scala.collection.mutable.Traversable[Int], scala.collection.mutable.Traversable[Int]) =
(ArrayBuffer(),ArrayBuffer(1, 2, 3, 4))
> Traversable(1,2,3,4) span (_ < 2)
res162: (scala.collection.mutable.Traversable[Int], scala.collection.mutable.Traversable[Int]) =
(ArrayBuffer(1),ArrayBuffer(2, 3, 4))
```

Traversable methods

`xs partition p` Split `xs` into a pair of collections; one with elements that satisfy the predicate `p`, the other with elements that do not, giving the pair of collections `(xs filter p, xs.filterNot p)`

```
> Traversable(1,2,3,4) partition (_ > 2)
res163: (scala.collection.mutable.Traversable[Int], scala.collection.mutable.Traversable[Int]) = (ArrayBuffer(3,
4),ArrayBuffer(1, 2))
> Traversable(1,2,3,4) partition (_ < 2)
res164: (scala.collection.mutable.Traversable[Int], scala.collection.mutable.Traversable[Int]) =
(ArrayBuffer(1),ArrayBuffer(2, 3, 4))
```

Traversable methods

`xs groupBy f` Partition `xs` into a map of collections according to a discriminator function `f`.

```
> Traversable(1,2,3,4) groupBy (_ < 2)
res167: scala.collection.immutable.Map[Boolean,scala.collection.mutable.Traversable[Int]] = Map(false -> ArrayBuffer(2,
3, 4), true -> ArrayBuffer(1))
> Traversable("1","2","344","4w") groupBy (_.length)
res169: scala.collection.immutable.Map[Int,scala.collection.mutable.Traversable[String]] = Map(2 -> ArrayBuffer(4w), 1 ->
ArrayBuffer(1, 2), 3 -> ArrayBuffer(344))
```

Traversable methods

```
xs containsSlice n
```

```
> Seq("1","2","3") containsSlice Seq("2", "3")
res272: Boolean = true
> Seq("1","2","3") containsSlice Seq("2")
res273: Boolean = true
> Seq("1","2","3") containsSlice Seq("2","1")
res274: Boolean = false
```

Traversable methods

`xs forall p` A boolean indicating whether the predicate `p` holds for all elements of `xs`.

```
> Traversable(1,2,3,4) forall (_ < 2)
res170: Boolean = false
> Traversable(1,2,3,4) forall (_ < 20)
res171: Boolean = true
```

Traversable methods

`xs exists p` A boolean indicating whether the predicate `p` holds for some element in `xs`.

```
> Traversable(1,2,3,4) exists (_ < 20)
res173: Boolean = true
> Traversable(1,2,3,4) exists (_ > 20)
res174: Boolean = false
```

Traversable methods

`xs count p` The number of elements in `xs` that satisfy the predicate `p`.

```
> Traversable(1,2,3,4) count (_ > 20)
res176: Int = 0
> Traversable(1,2,3,4) count (_ > 2)
res177: Int = 2
```

Traversable methods

`xs.foldLeft(z)(op)`

`(z /: xs)(op)`

Apply binary operation `op` between successive elements of `xs`, going left to right and starting with `z`.

```
val average = seq.sum / seq.length
val average = seq.foldLeft((0.0, 1)) ((acc, i) => ((acc._1 + (i - acc._1) / acc._2), acc._2 + 1))._1
```

Traversable methods

```
xs.foldRight(z)(op)
(xs :\ z)(op)
```

Apply binary operation `op` between successive elements of `xs`, going right to left and starting with `z`.

Traversable methods

`xs reduceLeft op` Apply binary operation `op` between successive elements of non-empty collection `xs`, going left to right.

```
> Traversable(1,2,3,4) reduceLeft(_ + _)
res178: Int = 10
```

Traversable methods

`xs reduceRight op` Apply binary operation `op` between successive elements of non-empty collection `xs`, going right to left.

```
> Traversable(1,2,3,4) reduceRight(_ + _)
res179: Int = 10
```

Traversable methods

`xs reduceRight op` Apply binary operation `op` between successive elements of non-empty collection `xs`, going right to left.

```
> Traversable(1,2,3,4) reduceRight(_ + _)
res179: Int = 10
```

Traversable methods

```
def aggregate[B](z: => B)(seqop: (B, Int) => B,combop: (B, B) => B): B
```

```
> Seq(1,2,3,4).aggregate(0)(
     |      (prev,curr) => prev + curr, // addToPrev
     |      (sumA,sumB) => sumA + sumB)  // combineSums
)
res266: Int = 10

> Seq(1,2,3,4)
    .grouped(2) // split into groups of 2 members each
    .map(prevAndCurrList => prevAndCurrList(0) + prevAndCurrList(1))
    .foldLeft(0)(sumA,sumB => sumA + sumB)
res267: Int = 10
```

Traversable methods

```
Seq(1).ensuring

def ensuring(cond: Boolean,msg: => Any): scala.collection.mutable.Seq[Int]
def ensuring(cond: scala.collection.mutable.Seq[Int] => Boolean,msg: => Any): scala.collection.mutable.Seq[Int]
def ensuring(cond: Boolean): scala.collection.mutable.Seq[Int]
def ensuring(cond: scala.collection.mutable.Seq[Int] => Boolean): scala.collection.mutable.Seq[Int]
```

```
> Seq(1,2,3).ensuring(_.size > 2, "Size <= 2!!!")
res281: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)

> Seq(1).ensuring(_.size > 2, "Size <= 2!!!")
java.lang.AssertionError: assertion failed: Size <= 2!!!
  at scala.Predef$Ensuring$.ensuring$extension3(Predef.scala:219)
  ... 29 elided
```

Traversable methods

Specific Folds:

`xs.sum`	The sum of the numeric element values of collection `xs`.
`xs.product`	The product of the numeric element values of collection `xs`.
`xs.min`	The minimum of the ordered element values of collection `xs`.
`xs.max`	The maximum of the ordered element values of collection `xs`.

Traversable methods

```
> Traversable(1,2,3,4).sum
res131: Int = 10
> Traversable(1,2,3,4).product
res132: Int = 24
> Traversable(1,2,3,4).max
res133: Int = 4
> Traversable(1,2,3,4).min
res134: Int = 1
> Traversable("1","2","3","4").min
res135: String = 1
> Traversable("1","2","3","4").sum
<console>:15: error: could not find implicit value for parameter num: Numeric[String]
       Traversable("1","2","3","4").sum
```

Traversable methods

`xs addString (b, start, sep, end)` Adds a string to `StringBuilder b` that shows all elements of `xs` between separators `sep` enclosed in strings `start` and `end`. `start`, `sep`, `end` are all optional.

Traversable methods

`xs mkString (start, sep, end)` Converts the collection to a string that shows all elements of `xs`between separators `sep` enclosed in strings `start` and `end`. `start`, `sep`, `end` are all optional.

```
> Traversable(1,2,3,4) mkString (",", "[", "]")
res184: String = ,1[2[3[4]
> Traversable(1,2,3,4) mkString ("[", ",", "]")
res185: String = [1,2,3,4]
> Traversable(1,2,3,4) mkString (",")
res186: String = 1,2,3,4
```

Traversable methods

`xs.stringPrefix` The collection name at the beginning of the string returned from `xs.toString`.

```
> Traversable(1,2,3,4).stringPrefix
res188: String = ArrayBuffer
```

Traversable methods

`xs.view` Produces a view over `xs`.

```
> Traversable(1,2,3,4).view
res189: scala.collection.TraversableView[Int,scala.collection.mutable.Traversable[Int]] = SeqView(...)
```

Traversable methods

`xs view (from, to)` Produces a view that represents the elements in some index range of `xs`.

```
> Traversable(1,2,3,4).view(2,3)
res190: scala.collection.TraversableView[Int,scala.collection.mutable.Traversable[Int]] = SeqViewS(...)
> Traversable(1,2,3,4).view(2,3).toList
res191: List[Int] = List(3)
```

Iterable methods

`xs grouped size` An iterator that yields fixed-sized “chunks” of this collection.

```
> Iterable(1,2,3,4) grouped(2)
res195: Iterator[scala.collection.mutable.Iterable[Int]] = non-empty iterator
> (Iterable(1,2,3,4) grouped 2).toList
res196: List[scala.collection.mutable.Iterable[Int]] = List(ArrayBuffer(1, 2), ArrayBuffer(3, 4))
```

Iterable methods

`xs sliding size` An iterator that yields a sliding fixed-sized window of elements in this collection.

```
> (Iterable(1,2,3,4) sliding 2).toList
res197: List[scala.collection.mutable.Iterable[Int]] = List(ArrayBuffer(1, 2), ArrayBuffer(2, 3), ArrayBuffer(3, 4))
> (Iterable(1,2,3,4) sliding 3).toList
res198: List[scala.collection.mutable.Iterable[Int]] = List(ArrayBuffer(1, 2, 3), ArrayBuffer(2, 3, 4))
```

Iterable methods

`xs zip ys` An iterable of pairs of corresponding elements from `xs` and `ys`.

```
> Seq(1,2,3) zip (Seq(3,4,5))
res199: scala.collection.mutable.Seq[(Int, Int)] = ArrayBuffer((1,3), (2,4), (3,5))
```

Iterable methods

`xs zipAll (ys, x, y)` An iterable of pairs of corresponding elements from `xs` and `ys`, where the shorter sequence is extended to match the longer one by appending elements `x` or `y`.

```
> Seq(1,2,3,4,5) zipAll (Seq(3,4), 1,2)
res204: scala.collection.mutable.Seq[(Int, Int)] = ArrayBuffer((1,3), (2,4), (3,2), (4,2), (5,2))
> Seq(1,2) zipAll (Seq(3,4,5,6,7), 1,2)
res205: scala.collection.mutable.Seq[(Int, Int)] = ArrayBuffer((1,3), (2,4), (1,5), (1,6), (1,7))
```

Iterable methods

`xs.zipWithIndex` An iterable of pairs of elements from `xs` with their indices.

```
> Seq(1,2,3).zipWithIndex
res206: scala.collection.mutable.Seq[(Int, Int)] = ArrayBuffer((1,0), (2,1), (3,2))
```

Iterable methods

`xs sameElements ys` A test whether `xs` and `ys` contain the same elements in the same order

```
> Seq(1,2,3) sameElements Seq(1,2,3)
res207: Boolean = true
> Seq(1,2,3) sameElements Seq(2,1,3)
res208: Boolean = false
> Set(1,2,3) sameElements Set(2,1,3)
res209: Boolean = true
> Set(1,2,3) sameElements Set(2,1)
res210: Boolean = false
```

Seq methods

```
xs.inits
def inits: Iterator[scala.collection.mutable.Seq[Int]]
```

```
> Traversable(1,2,3,4).inits
res118: Iterator[scala.collection.mutable.Traversable[Int]] = non-empty iterator

> Traversable(1,2,3,4).inits.toList
res119: List[scala.collection.mutable.Traversable[Int]] = List(ArrayBuffer(1, 2, 3, 4), ArrayBuffer(1, 2, 3),
ArrayBuffer(1, 2), ArrayBuffer(1), ArrayBuffer())
```

Seq methods

```
xs.combinations
def combinations(n: Int): Iterator[scala.collection.mutable.Seq[Int]]
```

```
> Seq(1,2,3,4,5).combinations(2).toList
res258: List[scala.collection.mutable.Seq[Int]] = List(ArrayBuffer(1, 2), ArrayBuffer(1, 3), ArrayBuffer(1,
4), ArrayBuffer(1, 5), ArrayBuffer(2, 3), ArrayBuffer(2, 4), ArrayBuffer(2, 5), ArrayBuffer(3, 4),
ArrayBuffer(3, 5), ArrayBuffer(4, 5))
> Seq(1,2,3,4,5).combinations(3).toList
res259: List[scala.collection.mutable.Seq[Int]] = List(ArrayBuffer(1, 2, 3), ArrayBuffer(1, 2, 4),
ArrayBuffer(1, 2, 5), ArrayBuffer(1, 3, 4), ArrayBuffer(1, 3, 5), ArrayBuffer(1, 4, 5), ArrayBuffer(2, 3, 4),
ArrayBuffer(2, 3, 5), ArrayBuffer(2, 4, 5), ArrayBuffer(3, 4, 5))
> Seq(1,2,3,4,5).combinations(4).toList
res260: List[scala.collection.mutable.Seq[Int]] = List(ArrayBuffer(1, 2, 3, 4), ArrayBuffer(1, 2, 3, 5),
ArrayBuffer(1, 2, 4, 5), ArrayBuffer(1, 3, 4, 5), ArrayBuffer(2, 3, 4, 5))
```

Seq methods

```
xs.permutations
def permutations: Iterator[scala.collection.mutable.Seq[Int]]
```

```
> Seq(1,2,3).permutations.toList
res288: List[scala.collection.mutable.Seq[Int]] = List(ArrayBuffer(1, 2, 3), ArrayBuffer(1, 3, 2),
ArrayBuffer(2, 1, 3), ArrayBuffer(2, 3, 1), ArrayBuffer(3, 1, 2), ArrayBuffer(3, 2, 1))
```

Seq methods

```
xs.combinations
def combinations(n: Int): Iterator[scala.collection.mutable.Seq[Int]]
```

```
> Seq(1,2,3,4,5).combinations(2).toList
res258: List[scala.collection.mutable.Seq[Int]] = List(ArrayBuffer(1, 2), ArrayBuffer(1, 3), ArrayBuffer(1,
4), ArrayBuffer(1, 5), ArrayBuffer(2, 3), ArrayBuffer(2, 4), ArrayBuffer(2, 5), ArrayBuffer(3, 4),
ArrayBuffer(3, 5), ArrayBuffer(4, 5))
> Seq(1,2,3,4,5).combinations(3).toList
res259: List[scala.collection.mutable.Seq[Int]] = List(ArrayBuffer(1, 2, 3), ArrayBuffer(1, 2, 4),
ArrayBuffer(1, 2, 5), ArrayBuffer(1, 3, 4), ArrayBuffer(1, 3, 5), ArrayBuffer(1, 4, 5), ArrayBuffer(2, 3, 4),
ArrayBuffer(2, 3, 5), ArrayBuffer(2, 4, 5), ArrayBuffer(3, 4, 5))
> Seq(1,2,3,4,5).combinations(4).toList
res260: List[scala.collection.mutable.Seq[Int]] = List(ArrayBuffer(1, 2, 3, 4), ArrayBuffer(1, 2, 3, 5),
ArrayBuffer(1, 2, 4, 5), ArrayBuffer(1, 3, 4, 5), ArrayBuffer(2, 3, 4, 5))
```

Seq methods

`xs indexOf x` The index of the first element in xs equal to x (several variants exist).

```
> Seq(1,2,3) indexOf 2
res211: Int = 1
```

Seq methods

`xs lastIndexOf x` The index of the last element in xs equal to x (several variants exist).

```
> Seq(1,2,3,3) lastIndexOf 3
res212: Int = 3
> Seq(1,2,3,3) lastIndexOf 2
res213: Int = 1
```

Seq methods

`xs indexOfSlice ys` The first index of xs such that successive elements starting from that index form the sequence ys.

```
> Seq(1,2,3,3) indexOfSlice Seq(2,3)
res214: Int = 1
> Seq(1,2,3,3) indexOfSlice Seq(3,3)
res215: Int = 2
```

Seq methods

`xs lastIndexOfSlice ys` The last index of xs such that successive elements starting from that index form the sequence ys.

```
> Seq(1,2,3,3,3,3,3) lastIndexOfSlice Seq(3,3)
res216: Int = 5
> Seq(1,2,3,3,3,3,3) lastIndexOfSlice Seq(2,3)
res217: Int = 1
```

Seq methods

`xs indexWhere p` The index of the first element in xs that satisfies p (several variants exist).

```
> Seq(1,2,3,3,3,3,3) indexWhere (_ > 3)
res218: Int = -1
> Seq(1,2,3,3,3,3,3) indexWhere (_ > 2)
res219: Int = 2
> Seq(1,2,3,3,3,3,3) indexWhere (_ > 1)
res220: Int = 1
```

Seq methods

`xs segmentLength (p, i)` The length of the longest uninterrupted segment of elements in xs, starting with xs(i), that all satisfy the predicate p.

```
> Seq(1,2,3,3,3,3,3) segmentLength (_ == 3, 1)
res224: Int = 0
> Seq(1,2,3,3,3,3,3) segmentLength (_ == 3, 2)
res225: Int = 5
```

Seq methods

`xs prefixLength p` The length of the longest prefix of elements in xs that all satisfy the predicate p.

```
> Seq(1,2,3,3,3,3,3) prefixLength (_ > 1)
res226: Int = 0
> Seq(1,2,3,3,3,3,3) prefixLength (_ > 0)
res227: Int = 7
```

Seq methods

`xs padTo (len, x)` The sequence resulting from appending the value x to xs until length len is reached.

```
> Seq(1,2,3) padTo (10, 0)
res228: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3, 0, 0, 0, 0, 0, 0, 0)
```

Seq methods

`xs patch (i, ys, r)` The sequence resulting from replacing r elements of xs starting with i by the patch ys.

```
> Seq(1,2,3,4,5) patch (2, Seq(6,7,8),0)
res249: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 6, 7, 8, 3, 4, 5)
> Seq(1,2,3,4,5) patch (2, Seq(6,7,8),3)
res250: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 6, 7, 8)
> Seq(1,2,3,4,5) patch (2, Seq(6,7,8),1)
res252: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 6, 7, 8, 4, 5)
```

Seq methods

`xs updated (i, x)` A copy of xs with the element at index i replaced by x.

```
> Seq(1,2,3) updated (0,2)
res229: scala.collection.mutable.Seq[Int] = ArrayBuffer(2, 2, 3)
```

Seq methods

`(xs corresponds ys)(p)` Tests whether corresponding elements of xs and ys satisfy the binary predicate p.

```
> (Seq(1,2,3) corresponds Seq(3,4,5))(_ > _)
res231: Boolean = false
> (Seq(1,2,3) corresponds Seq(3,4,5))(_ < _)
res233: Boolean = true
```

Buffer methods

`buf trimStart n` Removes first n elements from buffer.

```
> val b = Buffer(1,2,3)
b: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3)
> b trimStart (2)
> b
res237: scala.collection.mutable.Buffer[Int] = ArrayBuffer(3)
```

Buffer methods

`buf trimEnd n` Removes last n elements from buffer.

```
> val b = Buffer(1,2,3)
b: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3)
> b trimEnd (1)
> b
res239: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2)
```

Seq methods

`xs.distinct` A subsequence of xs that contains no duplicated element.

```
> Seq(1,2,3,4,4,4,5,6,7).toSet
res240: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 7, 3, 4)
> Seq(1,2,3,4,4,4,5,6,7).toSet.toSeq
res241: Seq[Int] = Vector(5, 1, 6, 2, 7, 3, 4)
> Seq(1,2,3,4,4,4,5,6,7).distinct
res242: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3, 4, 5, 6, 7)
```