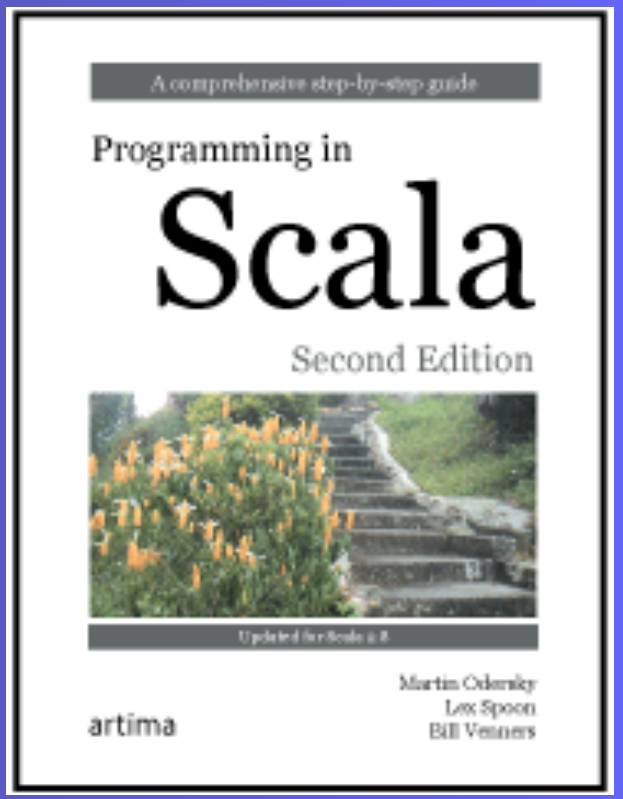
Stairway to Scala - Flight 15

Collections

Bill Venners
Dick Wall

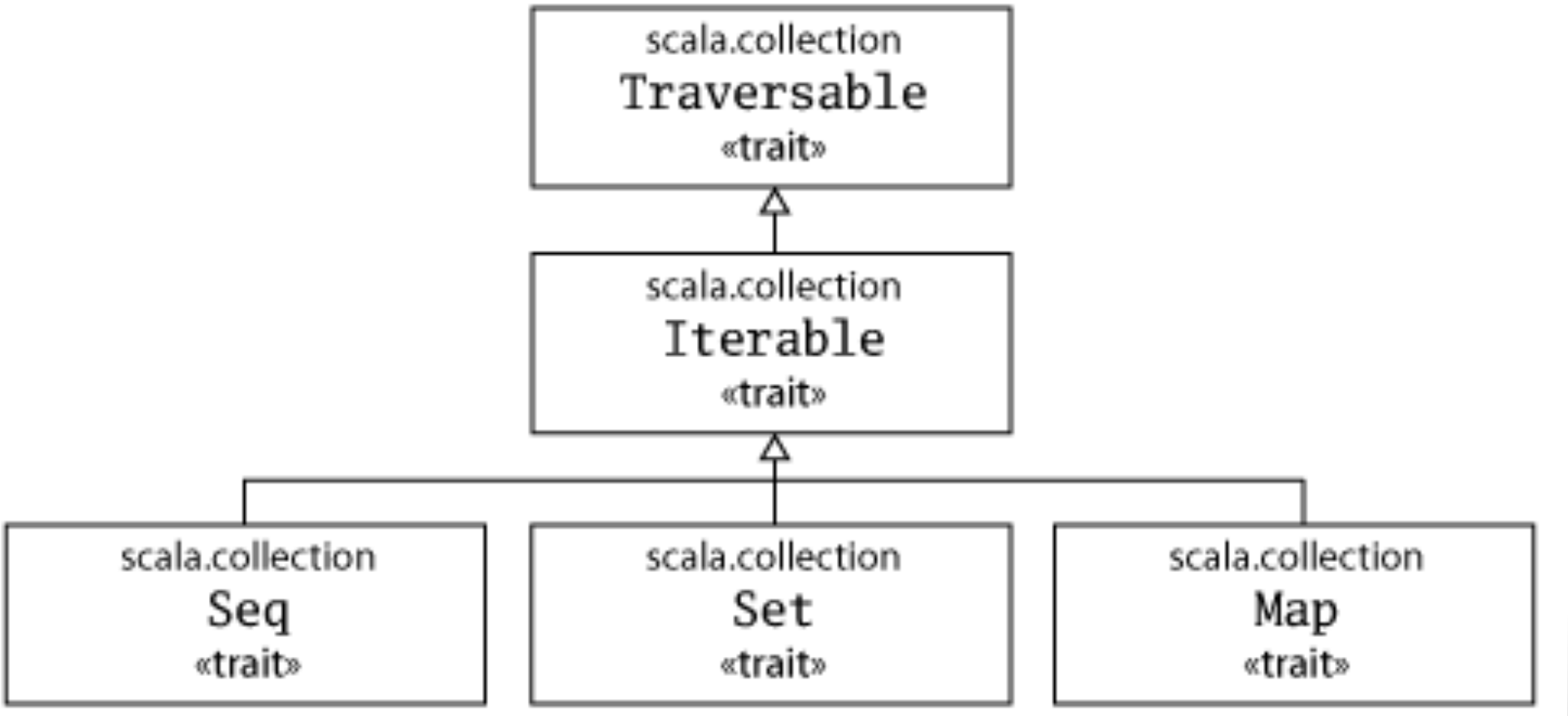


www.artima.com

Flight 15 goal

Get a good overview of Scala collections.

Collections hierarchy



Mutability modeled with types

scala.collection.

- Traversable, Iterable, Seq, Set, Map

scala.collection.immutable.

- Traversable, Iterable, Seq, Set, Map

scala.collection.mutable.

- Traversable, Iterable, Seq, Set, Map

```
Set(1, 2, 3) // immutable is default

import scala.collection._ // easy to use both
mutable.Set(1, 2, 3)
immutable.Set(1, 2, 3)
```

Consistent construction

```
Traversable(1, 2, 3)
Iterable("x", "y", "z")
Map("x" -> 24, "y" -> 25, "z" -> 26)
Set(Color.red, Color.green, Color.blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)

List(1, 2, 3)
HashMap("x" -> 24, "y" -> 25, "z" -> 26)
```

- Can also say List.empty, Buffer.empty, ...

Consistent equality

- Seqs, Sets, and Maps are always unequal to each other
- Within same category, equal if and only if contain same elements (and for Seq, in same order)
- Mutability doesn't matter

```
List(1, 2, 3) == Vector(1, 2, 3)
HashSet(1, 2, 3) == TreeSet(1, 2, 3)
```

Consistent toString

- toString returns a string that looks similar to the construction expression

```
List(1, 2, 3).toString == "List(1, 2, 3)"

Set('A', 'B', 'C') == "Set(A, B, C)"

import scala.collection.mutable.HashSet
HashSet('A', 'B', 'C') == "Set(A, B, C)"
```

Consistent return types

- All collection types support Traversable's methods, but with their own type as the return type

```
scala> List(1, 2, 3) map (_ * 2)
res7: List[Int] = List(2, 4, 6)

scala> Set(1, 2, 3) map (_ * 2)
res8: scala.collection.Set[Int] = Set(2, 4, 6)

scala> import scala.collection.mutable._
import scala.collection.mutable._

scala> HashSet(1, 2, 3) map (_ * 2)
res9: scala.collection.mutable.HashSet[Int] = Set(6, 4, 2)

scala> Vector(1, 2, 3) map (_ * 2)
res10: scala.collection.immutable.Vector[Int] = Vector(2, 4, 6)
```

Traversable methods (1)

Abstract method:

xs foreach f

Addition:

xs ++ ys

Mapping:

xs map f
xs flatMap f
xs collect pf

Conversions:

xs.toArray
xs.toList
xs.toIterable
xs.toSeq
xs.toIndexedSeq
xs.toStream
xs.toSet
xs.toMap

Traversable methods (2)

Copying:

xs copyToBuffer buf
xs copyToArray(arr, s, len)

Size info:

xs.isEmpty
xs.nonEmpty
xs.size
xs.hasDefiniteSize

Element retrieval:

xs.head
xs.headOption
xs.last
xs.lastOption
xs find p

Traversable methods (3)

Subcollections:

xs.tail
xs.init
xs slice (from, to)
xs take n
xs drop n
xs takeWhile p
xs dropWhile p
xs filter p
xs withFilter p
xs filterNot p

Subdivisions:

xs splitAt n
xs span n
xs partition p
xs groupBy f

Element Conditions:

xs forall p
xs exists p
xs count p

Traversable methods (4)

Folds:

(z /: xs)(op)
(xs :\ z)(op)

xs.foldLeft(z)(op)
xs.foldRight(z)(op)

xs reduceLeft op
xs reduceRight op

Views:

xs.view
xs view (from, to)

Specific folds:

xs.sum
xs.product
xs.min
xs.max

Strings:

xs.mkString(start, sep, end)
xs.addString(b, start, sep, end)
xs.stringPrefix

Iterable methods

Abstract method:

xs.iterator

Other iterators:

xs grouped size
xs sliding size

Subcollections:

xs takeRight n
xs dropRight n

Zippers:

xs zip ys
xs zipAll (ys, x, y)
xs zipWithIndex

Comparison:

xs sameElements ys

Seq methods (1)

Indexing and length:

xs(i)
xs isDefinedAt i
xs.length
xs.lengthCompare ys
xs.indices

Additions:

x +: xs // prepend
xs :+ x // append
xs padTo (len, x)

Index search:

xs indexOf x
xs lastIndexOf x
xs indexOfSlice ys
xs lastIndexOfSlice ys
xs indexWhere p
xs segmentLength (p, i)
xs prefixLength p

Updates

xs patch (i, ys, r)
xs updated (i, x)
xs(i) = x // xs.update(i, x)

Seq methods (2)

Sorting:

xs.sorted
xs sortWith lessThan
xs sortBy f

Reversals:

xs.reverse
xs.reverseIterator
s reverseMap f

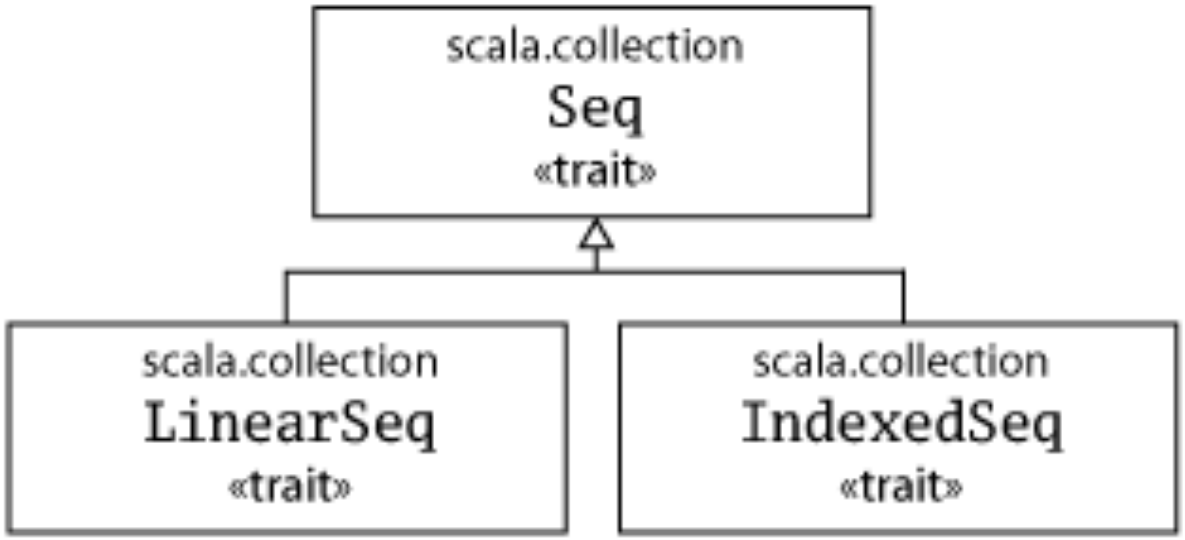
Comparisons:

xs startsWith ys
xs endsWith ys
xs contains x
xs containsSlice ys
(xs corresponds ys)(p)

Multiset operations:

xs intersect ys
xs diff ys
xs union ys
xs.distinct

Seq hierarchy



LinearSeq: efficient head and tail
examples: List and Stream
IndexedSeq: efficient apply, length, and (if mutable) update
examples: Array and ArrayBuffer (Buffers allow element insertions, removals, and efficient appending)

Buffer methods

Additions:

```
buf += x
buf += (x, y, z)
buf ++= xs
x +=: buf
xs ++=:  buf
buf insert (i, x)
buf insertAll (i, xs)
```

Removals:

```
buf -= x
buf remove i
buf remove (i n)
buf trimStart n
buf trimEnd n
buf.clear()
```

Cloning:

```
buf.clone
```

Set methods

Tests:

xs contains x
xs(x)
xs subsetOf ys

Additions:

xs + x
xs + (x, y, z)
xs ++ ys

Removals:

xs - x
xs - (x, y, z)
xs -- ys
xs.empty

Binary operations:

xs & ys
xs intersect ys
xs | ys
xs union ys
xs &~ ys
xs diff ys

mutable.Set methods

Additions:

```
xs += x
xs += (x, y, z)
xs ++= ys
xs add x
```

Removals:

```
xs -= x
xs -= (x, y, z)
xs --= ys
xs remove x
xs retain p
xs.clear()
```

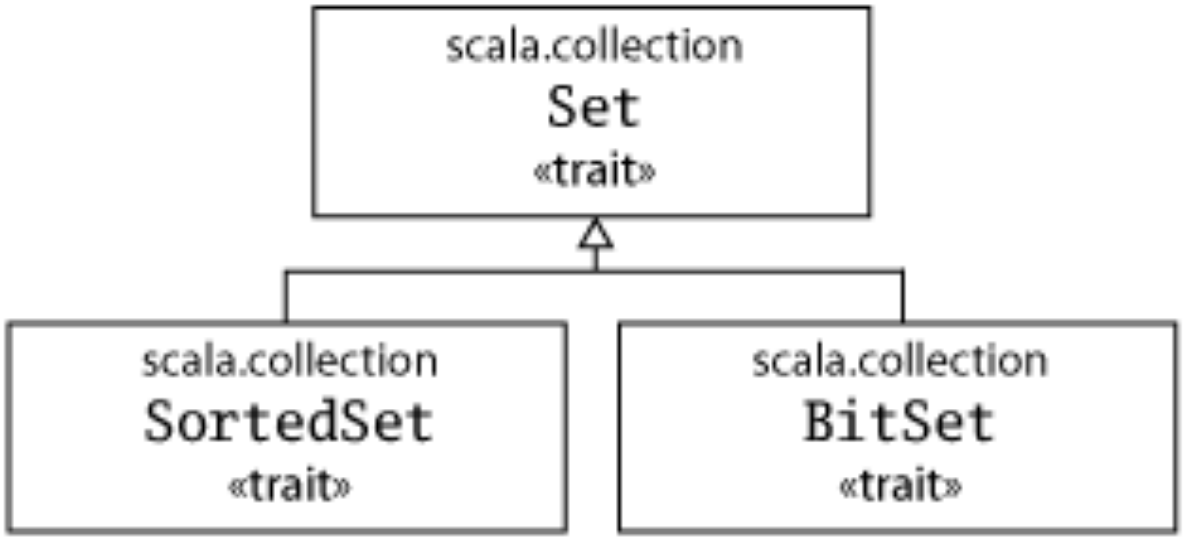
Update:

```
xs(x) = b
```

Cloning:

```
xs.clone
```

Set hierarchy



SortedSet: traversed in sorted order, no matter the order of addition

BitSet: sets of non-negative integer elements represented by bits in longs

Map methods

Lookups:

```
ms get k
ms(k)
ms getOrElse (k, d)
ms contains k
ms isDefinedAt k
```

Additions and updates:

```
ms + (k -> v)
ms + (k -> v, l -> w)
ms ++ kvs
ms updated (k, v)
```

Removals:

```
ms - k
ms - (k, l, m)
ms -- ks
```

Subcollections:

```
ms.keys
ms.keySet
ms.keysIterator
ms.values
ms.valuesIterator
```

Transformation:

```
ms filterKeys p
ms mapValues f
```

mutable.Map methods

Additions and Updates:

```
ms(k) = v
ms += (k -> v)
ms += (k -> v, l -> w)
ms ++= jvs
ms put (k, v)
ms getOrElseUpdate(k, d)
```

Removals:

```
ms -= k
ms -= (k, l, m)
ms --= ks
ms remove k
ms retain p
ms.clear()
```

Transformation:

```
ms transform f
```

Cloning:

```
ms.clone
```

Concrete immutable collections

- List
- Stream
- Vector
- Stack
- Queue
- Range
- String

- Hash tries (HashSet, HashMap, Set1..4, Map1..4)
- TreeSet/TreeMap
- BitSet
- ListMap

Concrete mutable collections

- ArrayBuffer
- ListBuffer
- StringBuilder
- MutableList
- Queue
- ArraySeq
- Stack
- ArrayStack
- Array

- Hash tables (HashSet, HashMap)
- WeakHashMap
- BitSet

Mutable to immutable and back

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> treeSet
res52: scala.collection.immutable.TreeSet[String] =
  TreeSet(blue, green, red, yellow)

scala> val mutaSet = mutable.Set.empty ++= treeSet
mutaSet: scala.collection.mutable.Set[String] =
  Set(yellow, blue, red, green)

scala> val immutaSet = Set.empty ++ mutaSet
immutaSet: scala.collection.immutable.Set[String] =
  Set(yellow, blue, red, green)
```

Views

- Transformer methods (map, filter, ++) can be strict or non-strict (lazy)
- All concrete collection implementations except Stream are strict

```
scala> val v = Vector(1 to 10: _*)
v: scala.collection.immutable.Vector[Int] =
Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> v map (_ + 1) map (_ * 2)
res5: scala.collection.immutable.Vector[Int] =
Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Views

- Can get a lazy collection with .view
- Can get back a strick collection with .force

```
scala> val vv = v.view
vv: scala.collection.SeqView[Int,Vector[Int]] =
SeqView(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> vv map (_ + 1)
res13: scala.collection.SeqView[Int,Seq[_]] = SeqViewM(...)

scala> res13 map (_ * 2)
res14: scala.collection.SeqView[Int,Seq[_]] = SeqViewMM(...)

scala> res14.force
res15: Seq[Int] = Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

it.next()
it.hasNext

Iterators

Iterators behave like collections *if you never access an iterator again after invoking a method on it*:

```
scala> val it = List(1, 2, 3).iterator
it: Iterator[Int] = non-empty iterator

scala> it.mkString
res12: String = 123

scala> it.mkString
res13: String =

scala>
```

Java and Scala collections

Iterator	<=>	java.util.Iterator
Iterator	<=>	java.util.Enumeration
Iterable	<=>	java.lang.Iterable
Iterable	<=>	java.util.Collection
mutable.Buffer	<=>	java.util.List
mutable.Set	<=>	java.util.Set
mutable.Map	<=>	java.util.Map

- Wrapping, no elements copied. Can "round trip."

Java and Scala collections

```
scala> import collection.JavaConversions._
import collection.JavaConversions._

scala> import collection.mutable._
import collection.mutable._

scala> val jul: java.util.List[Int] = ArrayBuffer(1, 2, 3)
jul: java.util.List[Int] = [1, 2, 3]

scala> val buf: Seq[Int] = jul
buf: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)

scala> val m: java.util.Map[String, Int] = HashMap("abc" -> 1,
"hello" -> 2)
m: java.util.Map[String,Int] = {hello=2, abc=1}
```

Scala-to-Java only conversions

Seq	<=>	java.util.List
mutable.Seq	<=>	java.util.List
Set	<=>	java.lang.Set
Map	<=>	java.util.Map