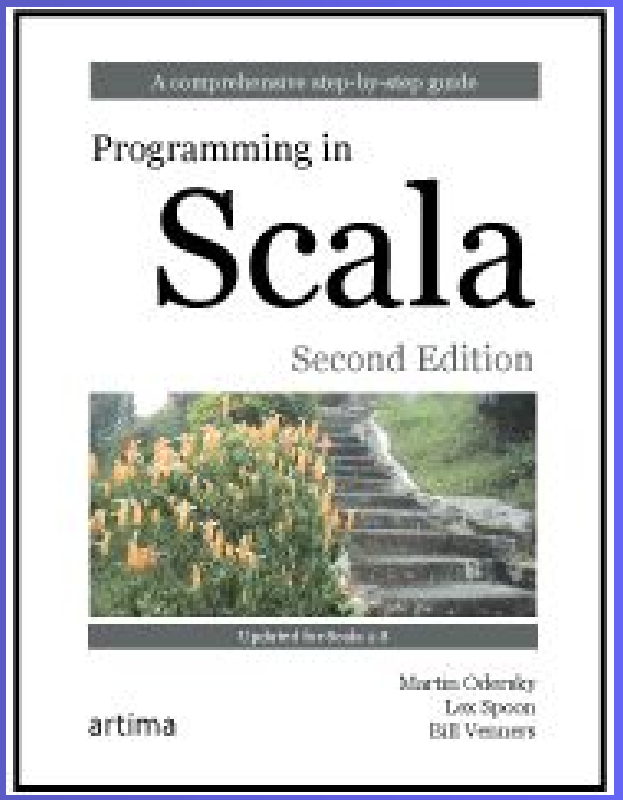
Stairway to Scala - Flight 15

Collections

Bill Venners
Dick Wall

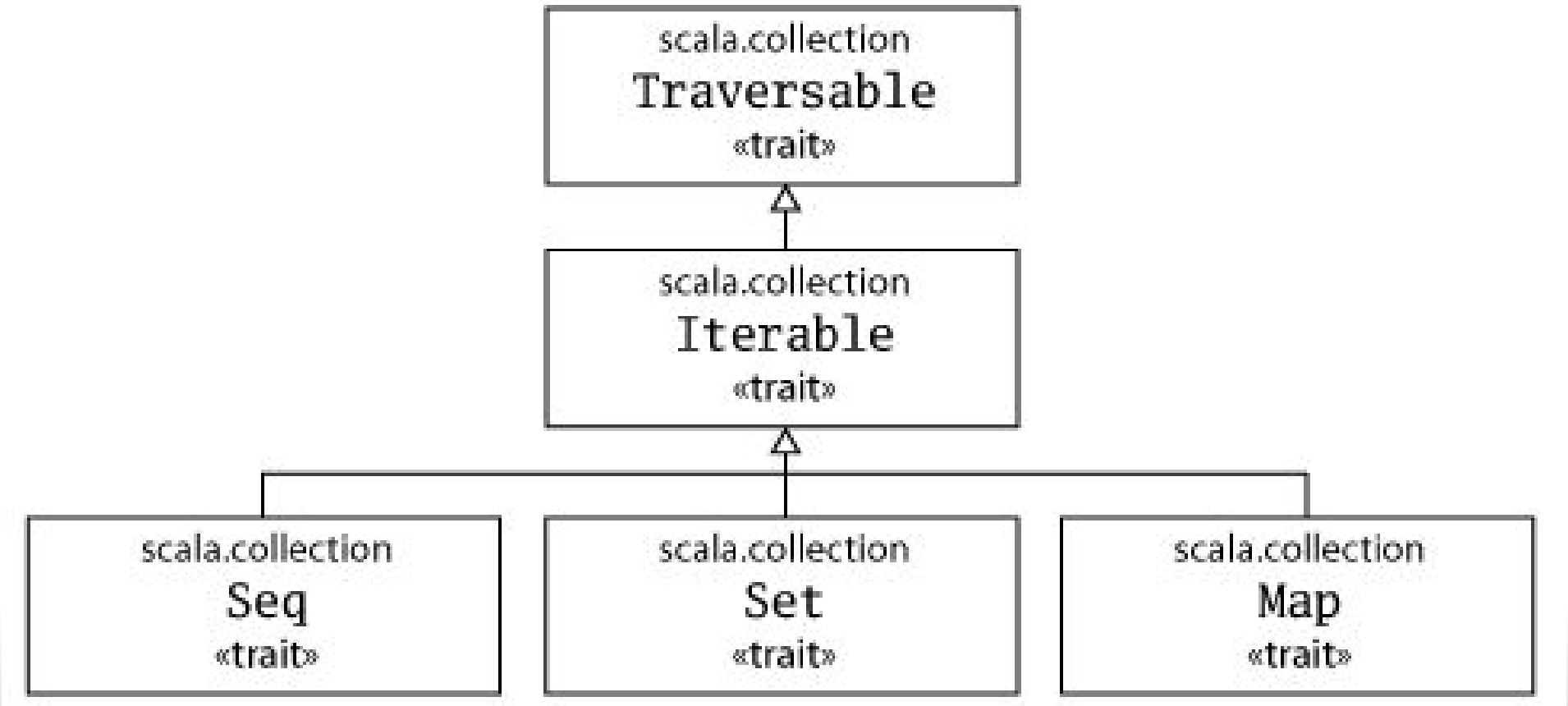


escalatesoft.com

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.mutable
import scala.collection.immutable
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)
```

- But Array says, "I'm not! (unless you make me deep)"

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

import scala.collection.mutable.HashSet
HashSet('A', 'B', 'C').toString == "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> mutable.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: easy conversion

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

Traversable: easy concatenation

```
scala> val xs = List(1, 2, 3, 3, 4, 5, 5, 5)
xs: List[Int] = List(1, 2, 3, 3, 4, 5, 5, 5)

scala> Set.empty ++ xs
res0: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

Traversable: How do I collect elements mapped by a *partial* function?

```
scala> val romNum = Map(
        "I" -> 1, "II" -> 2, "III" -> 3, "IV" -> 4, "V" -> 5
      )
romNum: scala.collection.immutable.Map[java.lang.String,Int]
= Map(II -> 2, IV -> 4, I -> 1, V -> 5, III -> 3)

scala> romNum collect {
      case (roman, arabic) if arabic % 2 == 0 => roman
    }
res2: scala.collection.immutable.Iterable[java.lang.String] =
List(II, IV)
```

Traversable: How do you partition into two collections according to a predicate?

```
scala> romNum partition {
     case (_, arabic) => arabic % 2 == 0
   }
res5: (scala.collection.immutable.Map[java.lang.String,Int],
scala.collection.immutable.Map[java.lang.String,Int]) = (Map
(II -> 2, IV -> 4),Map(I -> 1, V -> 5, III -> 3))
```

Or: romNum partition { _._2 % 2 == 0 }

(Returns a tuple of two collections)

Traversable: How do you group elements according to a function?

```
case class Person(first: String, last: String, age: Int)
val people = Set(
  Person("Fred", "Jones", 33),
  Person("Bob", "Smith", 49),
  Person("Sally", "Ames", 33),
  Person("Cindy", "Smith", 29),
  Person("Jim", "Roberts", 49)
)
```

(Returns a map of result to collection)

```
scala> people groupBy { _.age }
res8: immutable.Map[Int,immutable.Set[Person]] = Map(
49 -> Set(Person(Jim,Roberts,49), Person(Bob,Smith,49)),
29 -> Set(Person(Cindy,Smith,29)),
33 -> Set(Person(Fred,Jones,33), Person(Sally,Ames,33)))
```

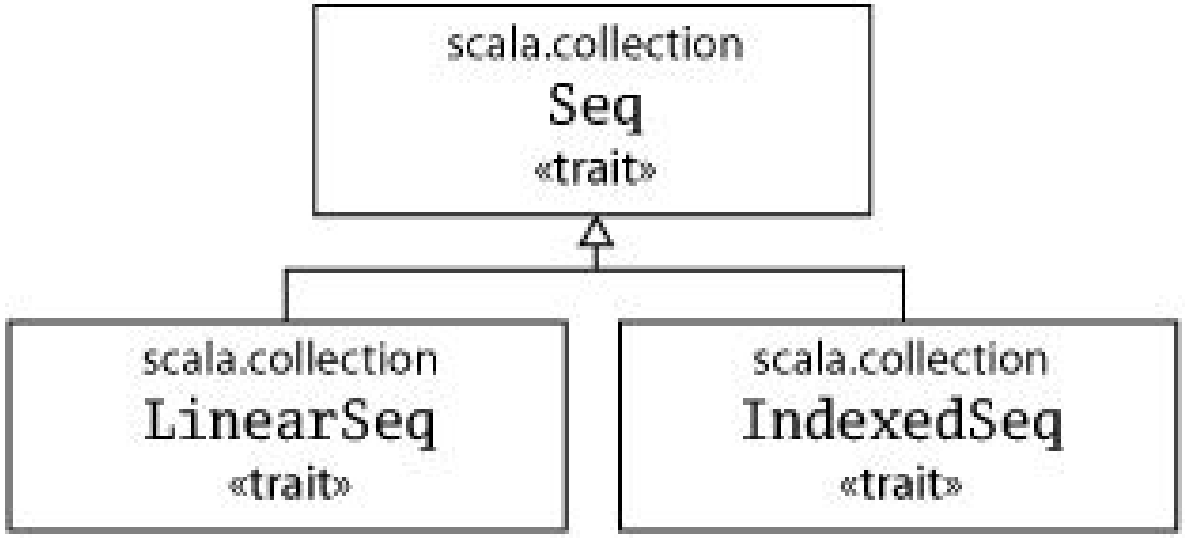
Iterable: grouping, sliding, zipping

```
xs grouped size
xs sliding size
xs zip ys
xs zipWithIndex
```

Seq: apply and updated

```
xs(i) // xs.apply(i)
xs updated (i, x)
xs(i) = x // xs.update(i, x)
```

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)

Buffers: updates, insertions, removals, appends

```
buf += x
buf insert (i, x)
```

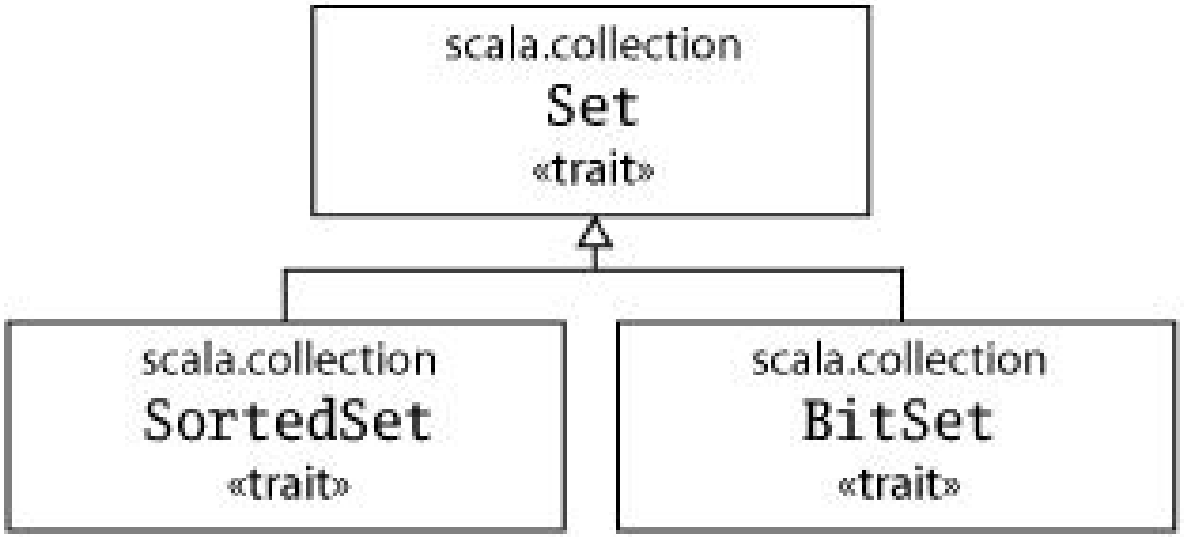
Sets

```
xs intersect ys
xs union ys
xs diff ys
```

mutable.Set

xs(x) = b

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

Maps

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

mutable.Map

```
ms getOrElseUpdate(k, d)
ms transform f
```

Concrete immutable collections

- List
- Stream - potentially infinite
- Vector - persistent immutable data structure with constant access time
- 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 = (1 to 10).toVector
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 strict 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>
```

Seq Performance Characteristics

	head	tail	apply	update	prepend	append	insert
immutable							
List	C	C	L	L	C	L	-
Stream	C	C	L	L	C	L	-
Vector	eC	eC	eC	eC	eC	eC	-
Stack	C	C	L	L	C	L	-
Queue	aC	aC	L	L	L	C	-
Range	C	C	C	-	-	-	-
String	C	L	C	L	L	L	-
mutable							
ArrayBuffer	C	L	C	C	L	aC	L
ListBuffer	C	L	L	L	C	C	L
StringBuilder	C	L	C	C	L	aC	L
MutableList	C	L	L	L	C	C	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C	-	-	-
Stack	C	L	L	L	C	L	L
ArrayStack	C	L	C	C	aC	L	L
Array	C	L	C	C	-	-	-

Set/Map Performance Characteristics

	lookup	add	remove	min
immutable				
`HashSet/HashMap`	eC	eC	eC	L
`TreeSet/TreeMap`	Log	Log	Log	Log
`BitSet`	C	L	L	eC^a
`ListMap`	L	L	L	L
mutable				
`HashSet/HashMap`	eC	eC	eC	L
`WeakHashMap`	eC	eC	eC	L
`BitSet`	C	aC	C	eC^a

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.JavaConverters._
import collection.JavaConverters._

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

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

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

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

Exercises for Flight 15