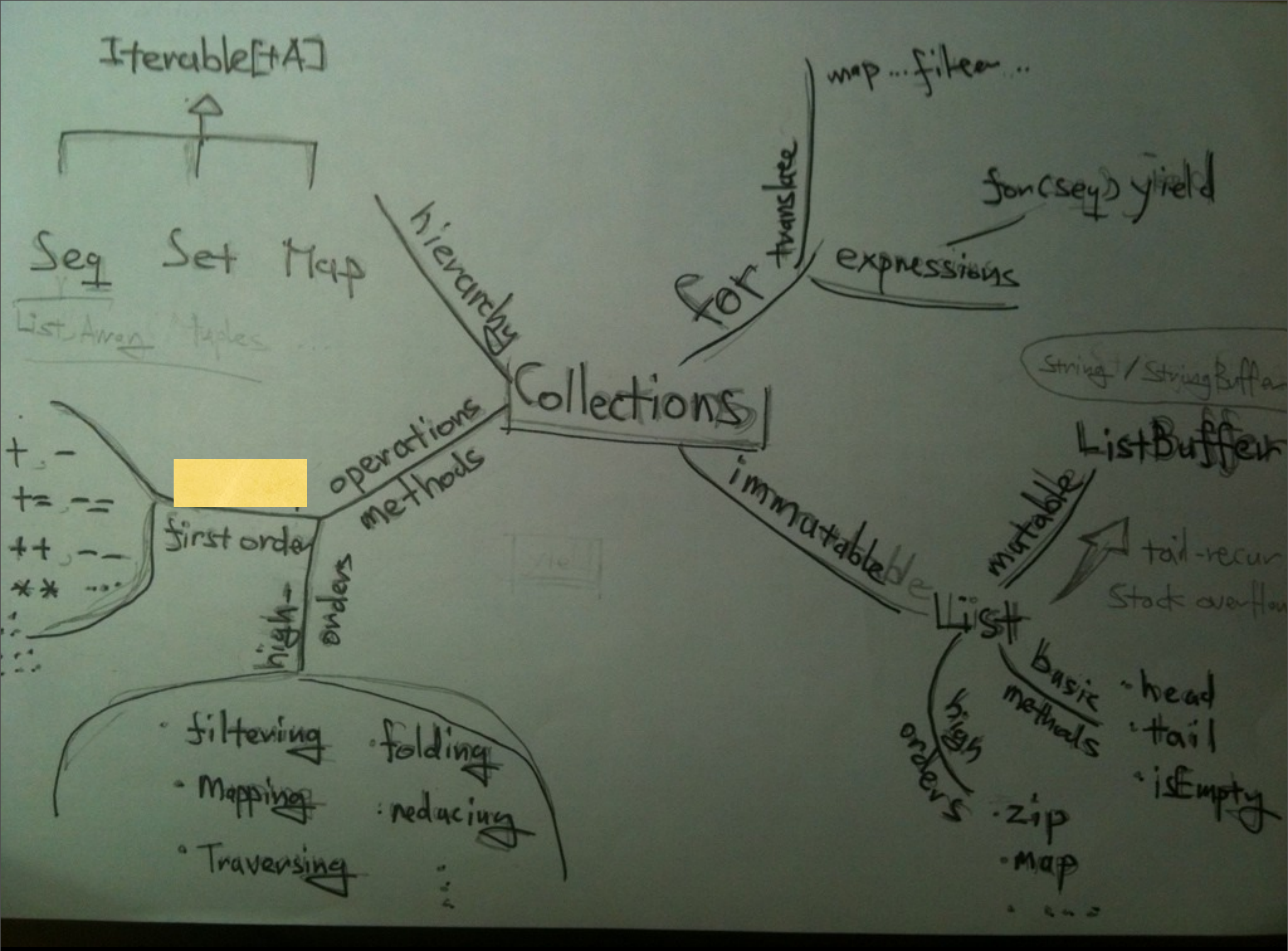
Using Collections

2010.07.02

anarcher@gmail.com


Iterable[+A]
Seq
Set
Map
List, Array, Tuples ...
hierarchy
Collections
for
translate
map ... filter ...
expressions
for(seq) yield
String / StringBuffer
ListBuffer
mutable
tail-recur
Stack overflow
immutable
List
basic methods
• head
• tail
• isEmpty
high orders
• zip
• map
• ...
operations
methods
first order
+, -
+=, -=
++, --
**, ...
high-orders
• filtering
• Mapping
• Traversing
• folding
• reducing

Sets

Unordered collections of unique elements

```
> val words = Set("As","Soon","As","Possible")
words: scala.collection.immutable.Set[String] =
Set(As,Soon,Possible)

 scala> words("As")
 res2: Boolean = true

 scala> words("A")
 res3: Boolean = false
```

Sets (Cont.)

Creating ,initializing, and using an immutable set.

```
var words = Set("As","Soon","As")
words += "Possible"
println(words.contains("Possible"))
```

Creating ,initializing, and using an mutable set.

```
import scala.collection.mutable.Set
val words = Set("As","Soon","As")
words += "Possible"
println(words("Possible"))
```

Maps

A dictionary of key-value pairs.

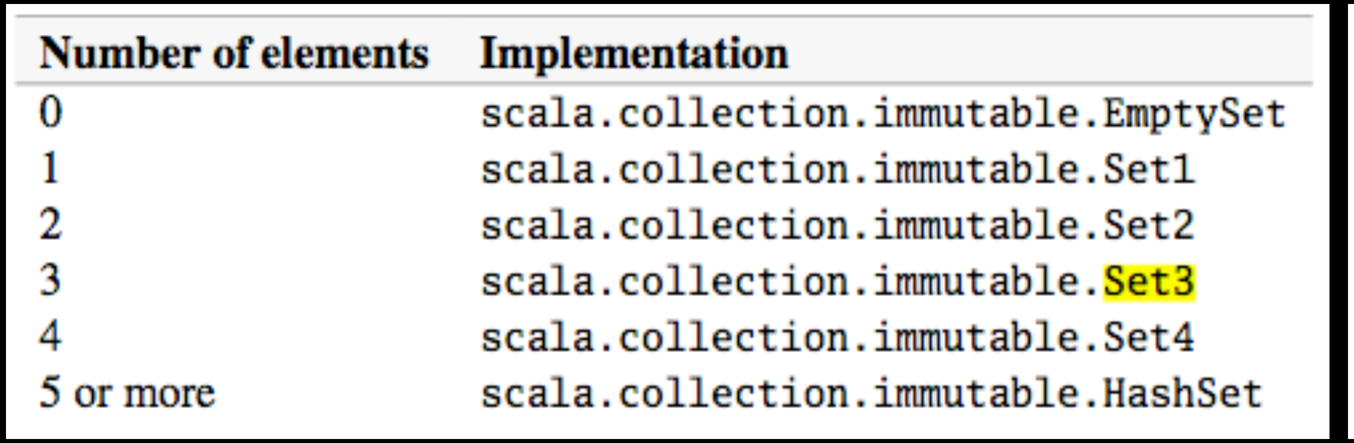
```
var romanNum = Map(1 -> "I" , 2 -> "II", 3 -> "III", 4 ->
"IV")
romanNum += (5 -> "V")

romanNum(5)
res12: java.lang.String = V

romanNum.get(5)
res15: Option[java.lang.String] = Some(V)

romanNum(6)
java.util.NoSuchElementException: key not found: 6
```

Maps (Cont.)

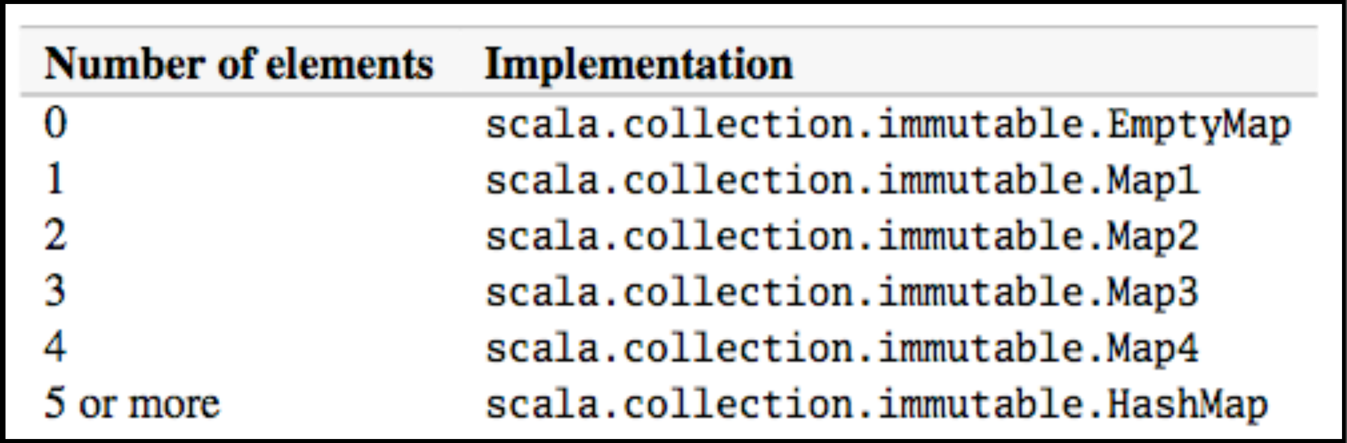
```
romanNum(5) = "V(Five)"
warning: there were deprecation warnings; re-run with -
deprecation for details
scala.collection.immutable.Map[Int,java.lang.String] = Map
((1,I), (2,II), (3,III), (5,V(Five)))


romanNum += ( 5 -> "V" )
scala.collection.immutable.Map[Int,java.lang.String] = Map
((1,I), (2,II), (3,III), (5,V))


romanNum.update(5,"V(5)")
warning: there were deprecation warnings; re-run with -
deprecation for details
scala.collection.immutable.Map[Int,java.lang.String] = Map
((1,I), (2,II), (3,III), (5,V(5)))
```

Default map and set definitions in Predef

```scala
object Predef {
    type Set[T] = scala.collection.immutable.Set[T]
    type Map[K,V] = scala.collection.immutable.Map[K,V]
    val Set = scala.collection.immutable.Set
    val Map = scala.collection.immutable.Map
}
```



Number of elements	Implementation
0	scala.collection.immutable.EmptySet
1	scala.collection.immutable.Set1
2	scala.collection.immutable.Set2
3	scala.collection.immutable.Set3
4	scala.collection.immutable.Set4
5 or more	scala.collection.immutable.HashSet



Number of elements	Implementation
0	scala.collection.immutable.EmptyMap
1	scala.collection.immutable.Map1
2	scala.collection.immutable.Map2
3	scala.collection.immutable.Map3
4	scala.collection.immutable.Map4
5 or more	scala.collection.immutable.HashMap

Lists

Ordered collections of elements of same type.

```
val fruit = List("apple","fineapple","oranges")
val nums = 1 :: 2 :: 3 :: Nil
val empty = List()
val empty = Nil

nums(0)
Int = 1
```

Iterable [+A]

represents collection objects
that can product an Iterator via method elements.

```
def elements : Iterator[A]
```

provides dozens of useful concrete methods. (and higher-order methods)

map filter findall exists foldLeft mkString...

foreach

The standard traversal method is foreach.

```
List(1,2,3,4) foreach { i => println ("Int:" + i) }

val romanNums = Map( 1 -> "I",
                     2 -> "II" ,
                     3 -> "III",
                    4 -> "IV" )
romanNum foreach { kv => println (kv._1 + " is " + kv._2 )



trait Iterable[+A] {
  ...
  def foreach(f : (A) => Unit) : Unit = ...
  ...
}
```

map

the map method returns a new collection of the same size as the original collection.

```
scala> val romanNums = Map ( 1 -> "I",
                             2 -> "II",
                             3 -> "III",
                             4 -> "IV",
                             5 -> "V")
res27: Map[Int,java.lang.String] = Map((5,V), (1,I), (2,II),
(3,III), (4,IV))

scala> romanNums map { kv => (kv._1,kv._2.length) }
res28: Map[Int,Int] = Map((5,1), (1,1), (2,2), (3,3), (4,2))

scala> List(1,2,3,4) map ( _ + 1 )
res29: List[Int] = List(2,3,4,5)
```

map(Cont.)

```
trait Iterable[+A] {
    ...
    def map[B](f: (A) => B ) : Iterable[B] = ...
    ...
}
```

filter

It is common to traverse a collection and extract a new collection from it with elements that match certain condition.

```
scala> List(1,2,3,4,5) filter ( _ > 3)
res30: List[Int] = List(4, 5)

scala> romanNums filter { kv => kv._1 > 3 }
res31: Map[Int,java.lang.String] = Map((5,V), (4,IV))


trait Iterable[+A] {
    ...
    def filter (p: (A) => Boolean ) : Boolean = ...
    ...
}
```

Folding & Reducing

“Shrinking” a collection down to a smaller collection or a single value.

Folding starts with an initial “seed” value.
Reducing doesn’t start with a user-supplied value.

Folding & Reducing

```
scala> List(2,3).foldLeft(10) ( _ * _ )
res4: Int = 60


scala> List(2,3).foldLeft(1) ( _ * _ )
res7: Int = 6

scala> List(2,3) reduceLeft(_ * _ )
res5: Int = 6

scala> val total = feeds.foldLeft(0) { (total, feed) => total +
feed.length }
scala> println("Total length of feed urls: " + total )
res2: Total length of feed urls: 61

scala> ( 10 /: List(2,3)) { _ * _ }
res22: Int = 60

scala> (List(2,3) :\ 10) ( _ * _ )
res26: Int = 60
```

Operator notation

Any method can be operator

```
s.indexOf('o') // is not an operator.
s indexOf '0'  // is an operator.
```

Infix operator

```
s indexOf '0',5 // (+ x y)
x :: xs // a special case of an infix operator.
::(x,xs)
```

Prefix operator (unary)

The only identifiers that can be used as prefix operators are +, -, !, and ~.

```
scala> -2.0 // Scala invokes (2.0).unary_-
res2: Double = -2.0
scala> (2.0).unary_-
res3: Double = -2.0
```

For expression

```
for ([pattern <- generator ; definition*]+ ; filter*)
 [yield] expression
```

```
val result = for (i <- 1 to 10) yield i * 2
val result2 = (1 to 10).map(_ * 2)
```

Scala translates the for expression into an expression that uses a combination of methods like map( ) and filter( ) depend- ing on the complexity of the expression.

```
val doubleEven = for (i <- 1 to 10; if i % 2 == 0)
    yield i * 2
```

list comprehension

For expression (cont.)

```
class Person(val firstName: String, val lastName: String)
object Person {
  def apply(firstName: String,
lastName: String) : Person =
    new Person(firstName, lastName)
}

val friends = List(Person("Brian", "Sletten"), Person
("Neal", "Ford"),
  Person("Scott", "Davis"), Person("Stuart", "Halloway"))

val lastNames = for (friend <- friends; lastName =
friend.lastName) yield lastName

println(lastNames.mkString(", "))
```

For expression (cont.)

```
for (i <- 1 to 3; j <- 4 to 6) {
  print("[" + i + "," + j + "] ")
}

// [1,4] [1,5] [1,6] [2,4] [2,5] [2,6] [3,4] [3,5] [3,6]
```

Q & A