Collections Standard Scala

Jean-Luc Falcone

4 juin 2013

Array

Java

```
String[] greetings = new String[2];
greetings[0] = "Hello";
greetings[1] = "Ciao";

int[] is = { 0, 1, 2, 3 };
int j = is[1] + is[2];
```

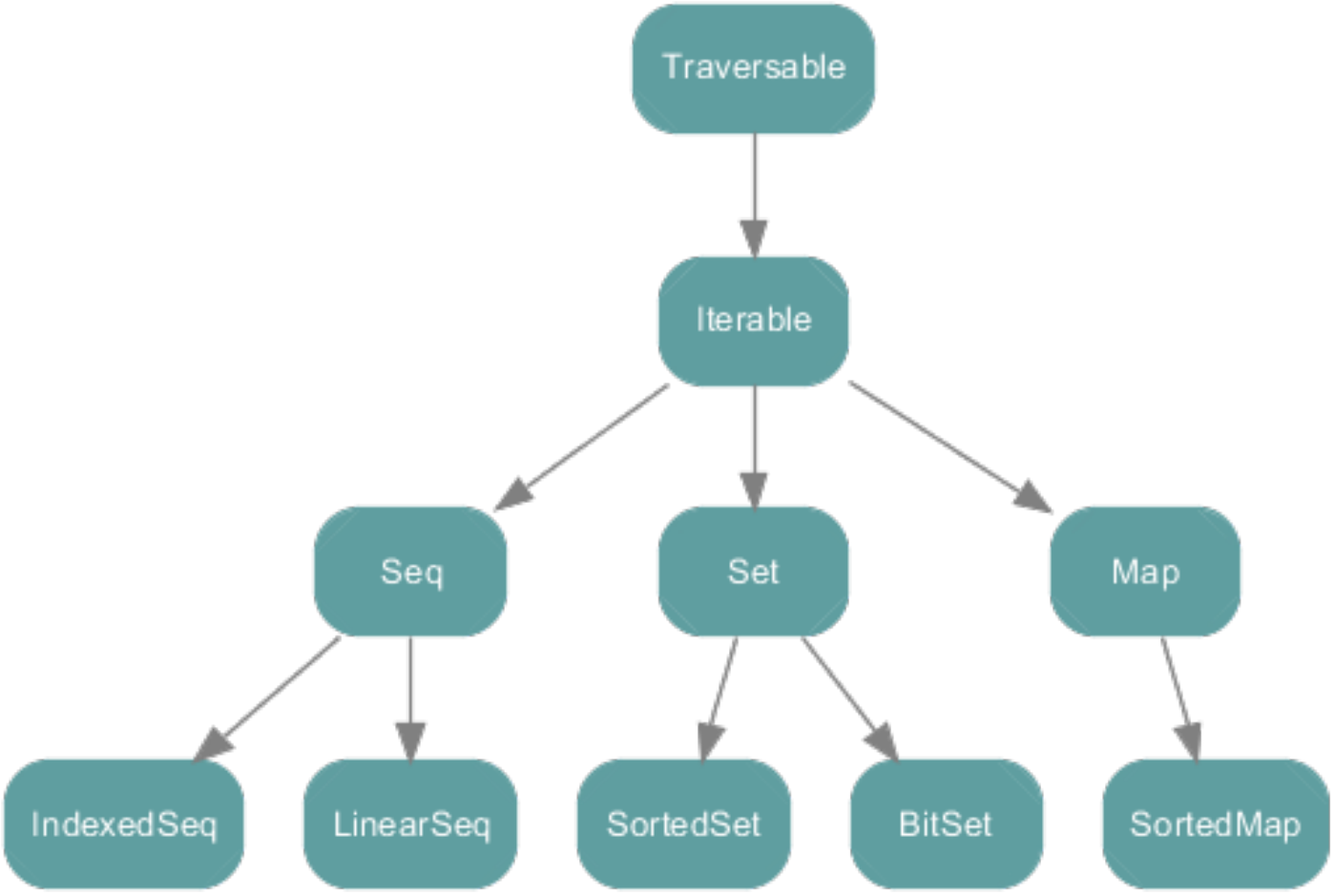
Array

Scala

```
val greetings = Array.ofDim[String](2)
greetings(0) = "Hello"
greetings(1) = "Ciao"

val is = Array( 0, 1, 2, 3 )
val j = is(1) + is(2)
```

Hiérarchie (scala.collection._)



Implémentation

Immutables (`scala.collection.immutable._`)

`Seq` List, String, Range, Vector, ...

`Map` HashMap, TreeMap, ListMap, ...

`Set` HashSet, ListSet, BitSet, ...

Mutables (`scala.collection.mutable._`)

`Seq` ArrayBuffer, StringBuilder, Stack, ...

`Map` HashMap, ObservableMap, ListMap, ...

`Set` HashSet, ListSet, BitSet, ...

Listes

```
val list = List( 1, 2, 4, 8, 16 )
val h = list.head
val t = list.tail

val list2 = 0 :: list
val list3 = Nil
val list4 = 1 :: 2 :: 3 :: Nil
```

Listes (pattern matching)

```
def max( is: List[Int] ): Int = {
  def maxRec( rem: List[Int], max: Int ): Int =
    rem match {
      case Nil => max
      case i :: rest if i>max => maxRec( rest,i )
      case _ :: rest => maxRec( rest, max )
    }
  maxRec( is.tail, is.head )
}

def even[A]( lst: List[A] ): Boolean =
  lst match {
    case _ :: _ :: rest => even(rest)
    case _ :: Nil => false
    case Nil => true
  }
```

Modèle: Gestion de Bibliothèques

```
case class Reader( id: Long, name: String, ... )

trait BookState
case object Available extends BookState
case class Borrowed( reader: Reader, maxDate: Date )
  extends BookState

case class Book( id: Long, state: BookState,
                                    title: String, ...)
```

Appliquer un effet de bord (foreach)

Afficher les livres empruntés.

```
books.foreach { b=>
  b.state match {
    case Borrowed( r, _ ) =>
      println( r + " a emprunte' " + b.title )
    case _ =>
  }
}
```

for comprehension

```
for( b <- books ) {
 b.state match {
    case Borrowed( r, _ ) =>
      println( r.name + " a emprunte' " + b.title )
    case _ =>
  }
}
```

for comprehension (2)

```
var sum = 0
for( i <- 0 to 99 ) {
  sum += i
}

var sum = 0
( 0.to(99) ).foreach{ i => sum += i }
```

Appliquer une fonction à chaque élément (map)

Extraire la liste des titres de tous les livres.

```
val titles = books.map( b => b.title )

val titles = for( b <- book ) yield {
  b.title
}
```

Filtrer les éléments (filter)

Obtenir la liste des livres disponibles.

```
val available = books.filter {
  b => b.state == Available
}
```

```
val available = for(
  b <- books if b.state == Available
) yield b
```

Résumé

- `F[A].foreach( f: A=>Unit ): Unit`
- `F[A].map( f: A=>B ): F[B]`
- `F[A].filter( f: A=>Boolean ): F[A]`

Exemple complexe

Une liste des noms des utilisateurs avec plus de 3 livres:

```
def booksOf( books: List[Book], userID: Long ): List[Book] =
  books.filter {
    b.state match {
      case Borrowed( r, _ ) => r.id == userID
      case _ => false
    }

val moreThan3Books = users.filter {
  u => booksOf(books,u.id).size > 3
}.map( _.name )
```

Version plus rapide

```
lazy val borrowed: List[(Book,User)] = books.collect{
  case b @ Book( _, Borrowed( u, _ ), _ ) => (b,u)
}

lazy val booksOf: Map[User,List[Book]] =
  borrowed.groupBy( _._2 ).mapValues( _.map(_._1) )

lazy val borrowers: List[User] =
  booksOf.keySet.toList

val moreThan3Books = borrowers.collect {
  case u if booksOf(u.id).size > 3 => u.name
}
```