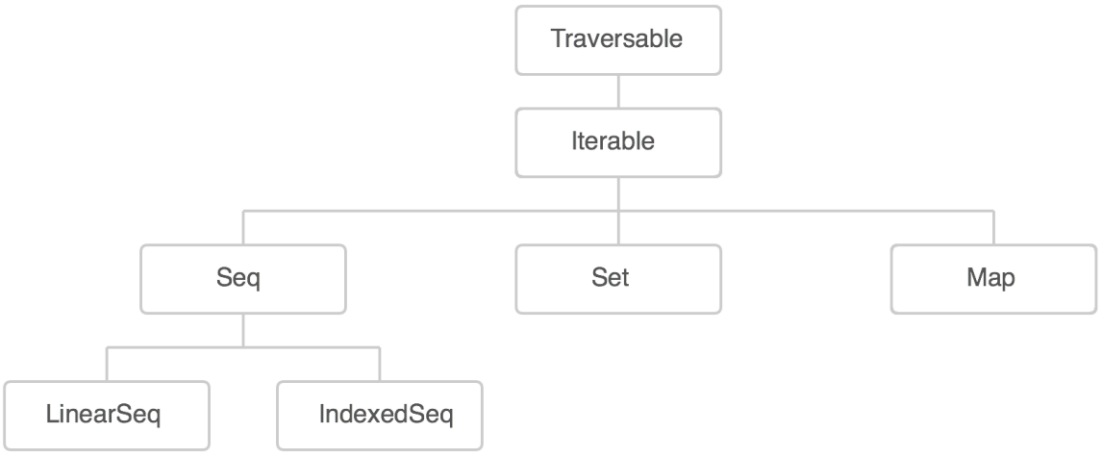
The Scala Collection hierarchy



Traversable:

Traversable is the top of the Collection hierarchy.

It defines a lot of the great operations that the Scala Collections has to offer. The only abstract method Traversable contains, is a foreach method.

```
def foreach[U](f: Elem => U): Unit
```

This method is essential to the operations in Traversable. The foreach method should traverse all the elements of the collection, while executing the function f on each of the elements. The actual implementation on the other hand, can differ a lot to make sure it's optimized for the different collections.

Iterable:

Contains an abstract iterator method that all sub-collections must define. Iterable also implements the abstract foreach method from Traversable using the iterator. But as I mentioned earlier, a lot of the sub-collections override this implementation to optimize.

Sequence:

Sequences differ a bit from normal Iterables. Sequences has a defined order and length. Seq has two sub-traits.

LinearSeq:

LinearSeq is defined in terms of three abstract methods.

- isEmpty - tells if the list is empty
- head - first element in the sequence
- tail - all elements in the sequence except the first one

It's assumed that sub-collections of LinearSeq will have good performance on their implementations of these methods. Typical collections that extends LinearSeq are Lists and Streams.

IndexedSeq:

IndexedSeq is defined in terms of two abstract methods.

- apply - Find element by its index
- length - The length of the sequence

It's assumed that sub-collections of IndexedSeq will have good performance on random access patterns. Typical indexed sequences are ArrayBuffer and Vectors. Vectors are a bit special, because their performance is good both on indexed and linear access.

Strings and Arrays also fit into this group, but it's worth mentioning that they are not subclasses of Seq. They are actually taken from Java. But by the help of different conversions, Scala offers the same operations on Strings and Arrays as on other sequence collections.

Set

Sets are Iterables without duplicate elements.

Map

Maps are Iterables that contains pairs of keys and values.

Mutable and Immutable

Scala distinguishes between mutable and immutable collections. For those collections that comes in both variants, Scala chooses immutable by default. So if you want a mutable variant, you have to explicitly write it.

BASIC OPERATIONS

Exists - Tests whether a predicate holds for some of the elements of this traversable collection.

```
// Given a list of programming languages.
// Check if "Scala" exists in the list.
List("Scala", "Java", "Haskell") exists (x => x == "Scala")


// Given a map of languages and their creators.
// Check if Odersky created a language.
Map("Scala" -> "Odersky", "Prolog" -> "Colmerauer") exists (x => x._2 ==
"Odersky")
```

Forall- Tests whether a predicate holds for all elements of this iterable collection.

```
// Given a list of integers.
// Check if vector only contains positive integers.
Vector(1, 2, -4, 3) forall (x => x >= 0)
```

Filter - Selects all elements of this collection which satisfy a predicate.

```
// Given a list of tuples (language, released).
// Get all languages created after 2000.
List(("Clojure", 2007), ("Haskell", 1990), ("Scala", 2003)) filter (x => x._2 >=
2000)
```

Map- Builds a new collection by applying a function to all elements of this collection.

```
// Given a list of integers.
// Get a list with all the elements squared.
List (1,2,3,4) map ( x => x * x)
```

Take - Selects first n elements.

```
// Given an infinite recursive method creating a stream of even numbers.
 def even: Stream[Int] = {
    def even0(n: Int): Stream[Int] =
      if (n%2 == 0) n #:: even0(n+1)
      else even0(n+1)
   even0(1)
 }

// Get a list of the 4 first even numbers.
even take (4) toList
```

GroupBy - Partitions this traversable collection into a map of traversable collections according to some discriminator function.

```
// Given a list of numbers.
// Group them as even and odd numbers.
List(1,2,3,4) groupBy (x => if (x%2 == 0) "even" else "odd")
```

Init - Selects all elements except the last.

```
List (1,2,3,4) init
```

Drop - Selects all elements except first n ones.

```
List (1,2,3,4) drop 2
```

TakeWhile - Takes the longest prefix of elements that satisfy a predicate.

This operation takes the ideas from the take operation and gives it more flexibility. Instead of having a static number of elements to return, we can now give a predicate to determine when to stop fetching elements.

Example

```
// Given an infinite recursive method creating a stream of even numbers.
 def even: Stream[Int] = {
    def even0(n: Int): Stream[Int] =
      if (n%2 == 0) n #:: even0(n+1)
      else even0(n+1)
    even0(1)
 }

// Return a list of all the elements until an element isn't less then 5.
even takeWhile (x => x < 5) toList
```

DropWhile - Drops the longest prefix of elements that satisfy a predicate.
We see the same changes to dropWhile as with takeWhile. It uses a predicate to determine how many elements to drop.

Example

```
List (1,3,6,5,4,2) dropWhile (x => x < 4)
```

FlatMap - flatMap combines the concepts of mapping and flattening into one operation.

```
List(List(1, 2), List(3, 4)) flatMap(x => x.map(x => x * x))
```

Fold, FoldLeft, FoldRight - These three operations are pretty similar.

They all take a start value and a function. The function takes two arguments. The first one is the accumulated value and the second is the value of the current element. Starting with the start value, they iterate the elements of the collection using the function given as an argument to fold the collection.

```
List(1,2,3,4,5).fold(0)(_ + _)
List(1,2,3,4,5).foldLeft(0)(_ + _)
List(1,2,3,4,5).foldRight(0)(_ + _)
```

The main one is that foldLeft iterates the list from left to right, while foldRight iterates the list from right to left. fold however, doesn't guarantee the order of function executions.

Let's create a final example, where we use foldLeft to solve a problem we looked at in the previous post.

```
// Given a list of tuples (language, released).
// Get all languages created after 2000.
List(("Clojure", 2007), ("Haskell", 1990), ("Scala",
2003)).foldLeft(List[String]())((x, y) => if(y._2 >= 2000) y._1 :: x else x)
```

Partition - Partitions this traversable collection in two traversable collections according to a predicate.
As you may imagine, this can be very useful sometimes. Let's look at an example where we want to split a list into two lists where one contains the even numbers and the other one the odd numbers.

```
List(1,2,3,4) partition (x => x % 2 == 0)
```

Zip - Returns a sequence formed from this sequence and another iterable collection by combining corresponding elements in pairs.
Let's just go straight to the example to see what this means.

```
List(1,2) zip (List(3,4))
```

As you can see, it returned a list of tuples where each tuple contains the n-th element from each of the lists.

Now let's look at an operation that can revert this result.

Unzip - Converts this collection of pairs into two collections of the first and second half of each pair.
As an example, let's unzip the zipped collection from our previous example.

```
List(1,2) zip (List(3,4)) unzip
```

Great! The unzip operation gave us a tuple back containing the two original lists.