

Scala Collections



cerenode.io

Collections in Scala

- Collections are the container of things which contains random number of elements
- Nothing but Classes found in the package `scala.collection`
- There are mutable and immutable collections
- Mainly there are three types of collection classes
 - Sequence
 - Set
 - Map




Mutable Collection

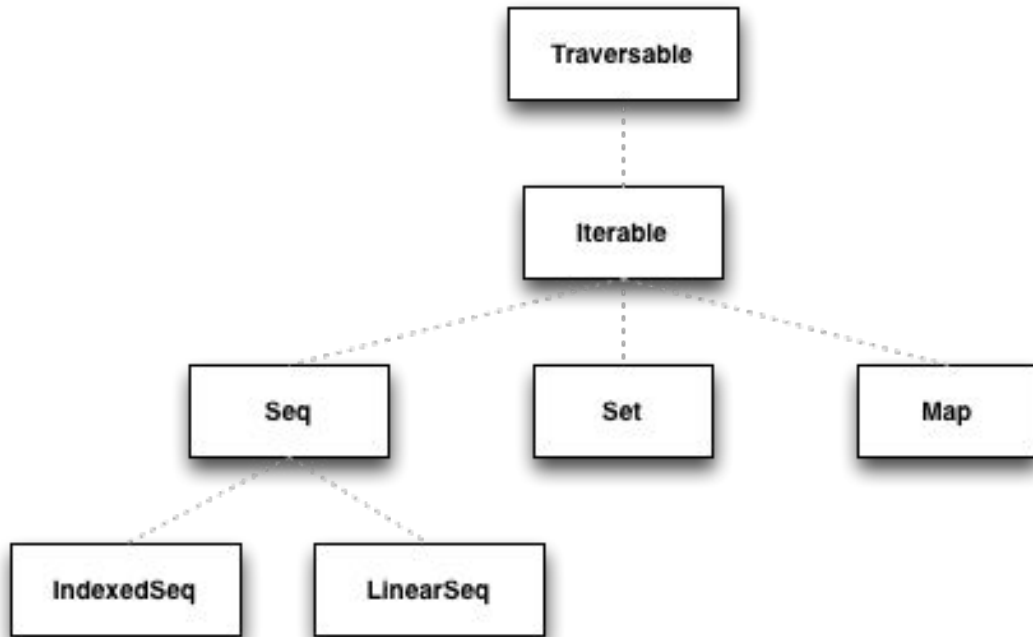
This type of collection is changed after it is created. All Mutable collection classes are found in the package `scala.collection.mutable`.

Immutable Collection

The elements in **most** immutable collections will not change after it is created. All immutable collection classes are found in the package `scala.collection.immutable`.

A decorative graphic in the bottom right corner consisting of several overlapping triangles in shades of pink and magenta.

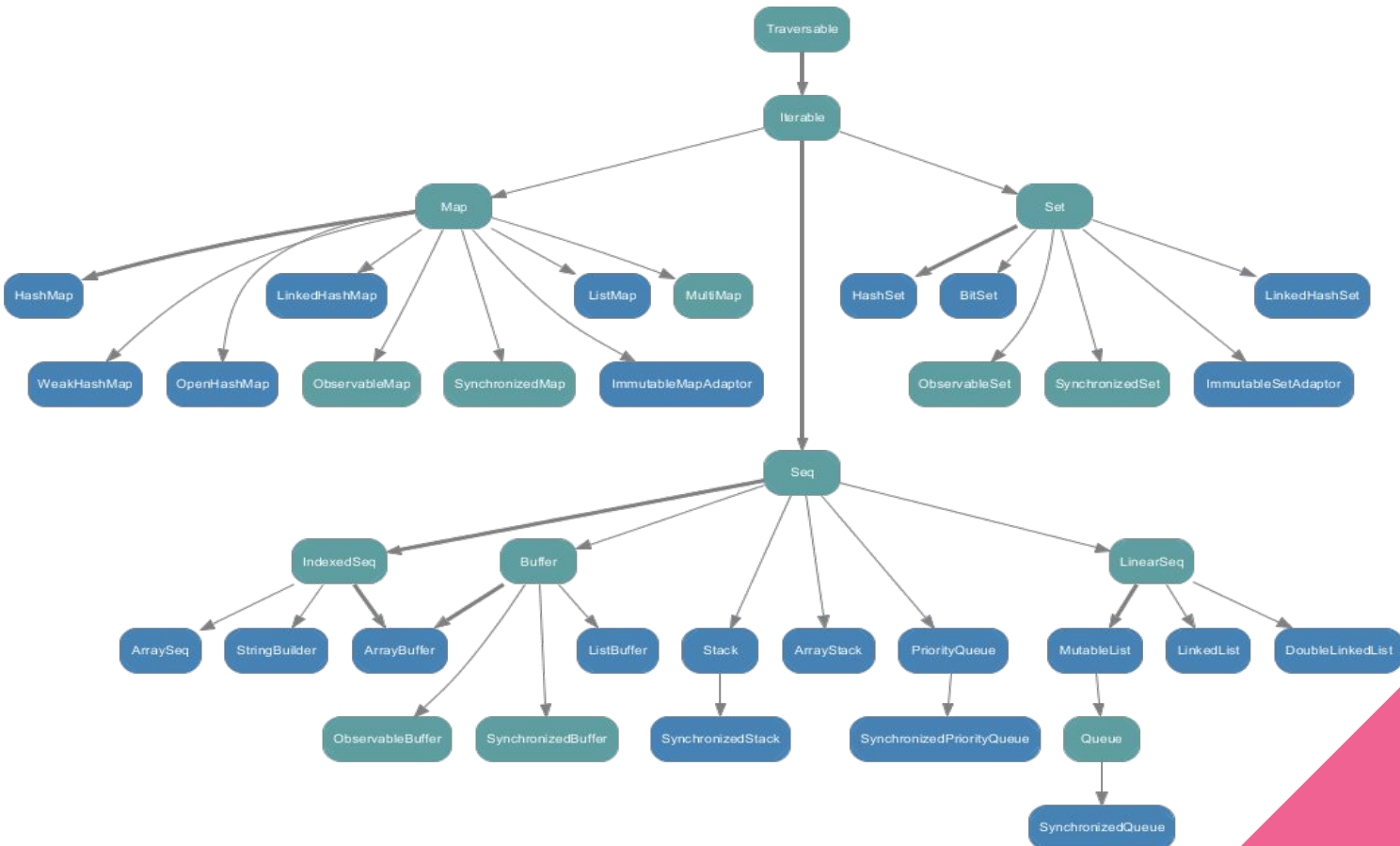
Scala Collections Hierarchy



All collections in package `scala.collection.immutable`



Collections in package `scala.collection.mutable`



Important Methods of Traversable Trait



```
def head: A
```

It returns the first element of collection.

```
def init: Traversable[A]
```

It returns all elements except last one.

```
def isEmpty: Boolean
```

It checks whether the collection is empty or not. It returns either true or false.

```
def last: A
```

It returns the last element of this collection.

```
def max: A
```

It returns the largest element of this collection.

```
def min: A
```

It returns smallest element of this collection



Important Methods of Traversable Trait (contd.)



```
def size: Int
```

It is used to get size of this traversable and returns a number of elements present in this traversable.

```
def sum: A
```

It returns sum of all elements of this collection.

```
def tail: Traversable[A]
```

It returns all elements except first.

```
def toArray: Array[A]
```

It converts this collection to an array.

```
def toList: List[A]
```

It converts this collection to a list.

```
def toSeq: Seq[A]
```

It converts this collection to a sequence.

```
def toSet[B >: A]: immutable.Set[B]
```

It converts this collection to a set.



Methods In Trait Iterable

`xs` **iterator**

An iterator that yields every element in `xs`, in the same order as `foreach` traverses elements.

`xs` **grouped** `size`

An iterator that yields fixed-sized “chunks” of this collection.

`xs` **sliding** `size`

An iterator that yields a sliding fixed-sized window of elements in this collection.

`xs` **takeRight** `n`

A collection consisting of the last `n` elements of `xs` (or, some arbitrary `n` elements, if no order is defined).



`xs dropRight n`

The rest of the collection except `xs takeRight n`.

`xs zip ys`

An iterable of pairs of corresponding elements from `xs` and `ys`.

`xs zipAll (ys, x, y)`

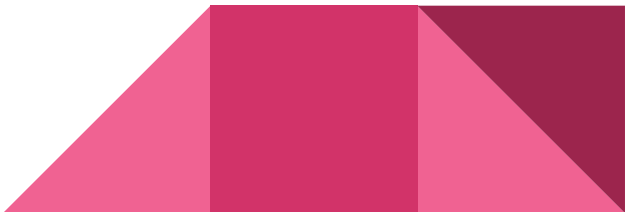
An iterable of pairs of corresponding elements from `xs` and `ys`, where the shorter sequence is extended to match the longer one by appending elements `x` or `y`.

`xs zipWithIndex`

An iterable of pairs of elements from `xs` with their indices.

`xs sameElements ys`

A test whether `xs` and `ys` contain the same elements in the same order



Sequence

- Collections of elements may be indexed or linear (as linked list)
- An IndexedSeq indicates that random access of elements is efficient

```
scala> val x = IndexedSeq(1, 2, 3)
x: IndexedSeq[Int] = Vector(1, 2, 3)
```

- A LinearSeq implies that the collection can be efficiently split into head and tail components, and it's common to work with them using the head, tail and isEmpty methods

```
scala> val seq = scala.collection.immutable.LinearSeq(1, 2, 3)
seq: scala.collection.immutable.LinearSeq[Int] = List(1, 2, 3)
```



Range

- Represent integer values in a range with non-zero step value
- Two ways to implement range
 - `val k: Range[Char] = 'a' to 'c' // elements a, b, c`
 - `val m: Range[Int] = 1 until 3 // elements 1, 2`
- Commonly used in for loops



List

- Data structure to represent a collection of values of the same type
- Immutable linked lists representing ordered collections of elements of type A
 - `val k: List[Int] = List(1, 2, 3, 4)`
- Optimal for stack-like access patterns
- Most commonly used data structure in Scala



Vector

- Provides random access and updates in constant time
- Fast append and prepend
 - `val k: Vector[Int] = Vector(1, 2, 3, 4)`
- Addresses the inefficiency for random access on Lists
- Implemented as 32-way trees



Array

- Array is a special kind of collection in Scala
- Even though it's an immutable collection, we can change the value of its element
- We can create up to five dimensional array in scala
- They store elements of a single type in a linear order
- “class” Array is Java’s Array, which is more of a memory storage method than a class



Methods In Trait Seq



Indexing and length apply, isDefinedAt, length, indices, and lengthCompare

Index search operations indexOf, lastIndexOf, indexOfSlice, lastIndexOfSlice, indexWhere, lastIndexWhere, segmentLength, prefixLength, which return the index of an element equal to a given value or matching some predicate.

Addition operations +:, :+, padTo, which return new sequences obtained by adding elements at the front or the end of a sequence.

Update operations updated, patch, which return a new sequence obtained by replacing some elements of the original sequence.

Sorting operations sorted, sortWith, sortBy, which sort sequence elements according to various criteria.

Reversal operations reverse, reverseliterator, reverseMap, which yield or process sequence elements in reverse order.

Comparisons startsWith, endsWith, contains, containsSlice, corresponds, which relate two sequences or search an element in a sequence.

Multiset operations intersect, diff, union, distinct, which perform set-like operations on the elements of two sequences or remove duplicates.



Sets

- Scala Set is a collection of unique elements

```
scala> val set = Set(1, 2, 3)  
set: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```



Methods In Sets

- **Lookup** operations contains, apply, subsetOf
- **Additions** + and ++, which add one or more elements to a set, yielding a new set.
- **Removals** -, --, which remove one or more elements from a set, yielding a new set.
- **Set operations** for union, intersection, and set difference. Each of these operations exists in two forms: alphabetic and symbolic. The alphabetic versions are intersect, union, and diff



Maps

- Scala Map is a collection of key/value pairs, where all the keys must be unique.

```
scala> val week = Map(1-> "Sun", 2 -> "Mon",3 ->"Tue" , 4 -> "Wed", 5 ->
"Thu" , 6 -> "Fri", 7-> "Sat")
week: scala.collection.immutable.Map[Int,String] = Map(5 -> Thu, 1 -> Sun,
6 -> Fri, 2 -> Mon, 7 -> Sat, 3 -> Tue, 4 -> Wed)
```

- Values can be retrieved using keys
 - `map(1)` // gives the String "Sun"



Methods In Maps

- **Lookup** operations `apply`, `get`, `getOrElse`, `contains`, and `isDefinedAt`. These turn maps into partial functions from keys to values. The fundamental lookup method for a map is: `def get(key): Option[Value]`. The operation “`m get key`” tests whether the map contains an association for the given key. If so, it returns the associated value in a `Some`. If no key is defined in the map, `get` returns `None`.
- Maps also define an `apply` method that returns the value associated with a given key directly, without wrapping it in an `Option`. If the key is not defined in the map, an exception is raised.
- **Additions and updates** `+`, `++`, `updated`, which let you add new bindings to a map or change existing bindings.
- **Removals** `-`, `--`, which remove bindings from a map.
- **Subcollection producers** `keys`, `keySet`, `keysIterator`, `values`, `valuesIterator`, which return a map's keys and values separately in various forms.
- **Transformations** `filterKeys` and `mapValues`, which produce a new map by filtering and transforming bindings of an existing map.

Special Collections In Scala

There are a few other classes that act like collections

- Tuples
- Enumerations
- Option
 - Some/None
- Try
 - Success/Failure



Tuple

- A Scala Tuple is a class that can contain a heterogeneous collection of elements
- A tuple can hold objects with different types and they are immutable
- There are 22 types of Tuple classes available

Tuple1, Tuple2,, Tuple22

```
scala> val Things = new Tuple3(1, "a", 2.5)  
Things: (Int, String, Double) = (1, a, 2.5)
```



Enumeration

- Defines a finite set of values specific to the enumeration
- Provide a lightweight alternative to case classes

```
object Weekday extends Enumeration {  
  val Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday = Value  
}
```



Option

- Scala `Option[T]` is a container for zero or one element of a given type `T`.
- `Option` is unimplemented
- Relies on two subtypes for the implementation
 - `Some`, a type-parameterized collection of one element
 - `None`, an empty collection.



Try

- Represents a computation that may either result in an exception, or return a successfully computed value
- Use when something exceptional could happen that cannot be handled in the function.
- Try is unimplemented but has two implemented subtypes
 - The Success type contains the return value of the attempted expression if no exception was thrown
 - Failure type contains the thrown Exception



Complexities of operations

	head	tail	append
List	C	C	L
Vector	eC	eC	eC
Range	C	C	-

	Lookup	add	remove
HashMap	eC	eC	eC

C - Constant time

eC - Effectively constant time, but
might depend on maximum length
Vector or distribution of hash keys

L - Operation is linear



Mutable Collections

- Similar to immutable collections but the collections are mutable
- Methods available for immutable are also used for mutable



Choosing a collection

- Most commonly used collections are :
 - Map, Set, List, ArrayBuffer, Vector
- If the requirement is to have a key-value lookup, we use Maps.
 - HashMap when order is of no importance
 - ListMap when the key-value pairs are to be stored in a sequence
 - Takes linear time, as the number of elements increases.
- Sets, contain no duplicate elements, so to remove duplicates use a Set, or convert the collection to a Set.
- If you need to store finite elements, traverse through them, or perform some operation on them, choose List
- ArrayBuffer is also a good choice in case you need it to be mutable



Choosing a collection

- If requirement is random access, and traversal is not of much importance, an indexed sequence is recommended
- If you want faster random access and a persistent sequence, use `Vector`
 - Persistent, because it preserves the previous version of itself
 - Not possible with an `ArrayBuffer`, because of mutability.
- Finally, the immutables. We can create a `Range` on the go with some collection's size, or something of that sort. It's easy to create a `Range` with `in`, `until`, and `by` methods.



Important Transformer Methods In Collections

- Constructs a new collection from an existing collection, typically by applying an algorithm to the input collection
- This includes methods like map, filter, reverse, etc..



map Method

- Used to iterate over a collection, performing an operation on each element and adding the result to a new collection
- map is a higher order function
- Behaves identically in Sequence and Set classes
 - `scala> List(1, 2, 3).map { x => x * 2 }`
`List[Int] = List(2, 4, 6)`
 - `scala> Array(1, 2, 3).map(_ * 2) //using the underscore(_) shorthand`
`Array[Int] = Array(2, 4, 6)`
 - `Set(1, 2, 2, 3).map(_ * 2)`
`Set[Int] = Set(2, 4, 6)`

map Method (contd....)

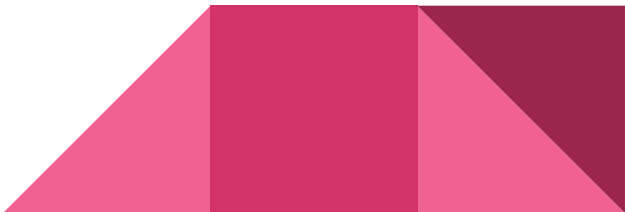
- The Map collection also has a map method, but it converts each key-value pair into a tuple for submission to the mapping function

```
scala> Map("a"-> 1, "b" -> 2).map { case (key , value) => (key, value*2) }  
res0: scala.collection.immutable.Map[String,Int] = Map(a -> 2, b -> 4)
```

- Can also be mapped to other collection types

```
scala> Map("a"-> 1, "b" -> 2).map { case (key , value) => key }  
res0: scala.collection.immutable.Iterable[String] = List(a, b)
```

```
scala> Map("a"-> 1, "b" -> 2).map { case (key , value) => value }  
res1: scala.collection.immutable.Iterable[Int] = List(1, 2)
```



flatten Method

- Used to eliminate undesired collection nesting

```
scala> List(List(1, 2, 3), List(4, 5, 6)).flatten  
res2: List[Int] = List(1, 2, 3, 4, 5, 6)
```

```
scala> Set(List(1, 2, 3), Set(3, 4, 5, 6)).flatten  
res8: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 3, 4)
```

```
scala> Set(List(1, 2, 3), Map(3 -> 5, 6 -> 7)).flatten  
res9: scala.collection.immutable.Set[Any] = Set((6, 7), 1, (3, 5), 2, 3)
```

flatMap Method

- Acts as a shorthand to map a collection and then immediately flatten it

```
scala> val listofLists = List(1, 3).map(x => List(x, x+1))  
listofLists: List[List[Int]] = List(List(1, 2), List(3, 4))  
scala> listofLists.flatten  
res19: List[Int] = List(1, 2, 3, 4)
```

```
// this can achieved with flatMap as follows  
scala> List(1, 3).flatMap(x => List(x, x + 1))  
res20: List[Int] = List(1, 2, 3, 4)
```

map Over Options

- When mapping an `Option` with an inner value (`Some(value)`), the mapping function acts on that value. However, when mapping an `Option` without a value (`None`), the mapping function will just return another `None`.

```
scala> val opt: Option[Int] = Some(1)
opt: Option[Int] = Some(1)
scala> opt.map(_*2)
res29: Option[Int] = Some(2)
```

```
scala> val opt: Option[Int] = None
opt: Option[Int] = None
scala> opt.map(_*2)
res30: Option[Int] = None
```



flatten & flatMap Over Options

- flatten will eliminate nested Option
- flattening a collection of Options will eliminate Nones and reduce Somes to their inner values:

```
scala> Some(Some(1)).flatten
```

```
Option[Int] = Some(1)
```

```
scala> List(Some(1),Some(2),None,Some(4),None).flatten
```

```
List[Int] = List(1, 2, 4)
```

```
scala> List( 1, -1, 2, -2).flatMap(x => if(x > 0) Some(x) else None)
```

```
List[Int] = List(1, 2)
```

A decorative graphic in the bottom right corner consisting of several overlapping triangles and rectangles in shades of pink and red.

reduceLeft & foldLeft Method

- Use reduceLeft to walk through a sequence from left to right
- reduceLeft starts by comparing the first two elements in the collection and returns a result
- The result is compared with the third element, and that comparison yields a new result and so on
- The foldLeft method works like reduceLeft, but lets you set a seed value to be used for the first element

```
scala> (1 to 10).reduceLeft((a,b) => a + b)
```

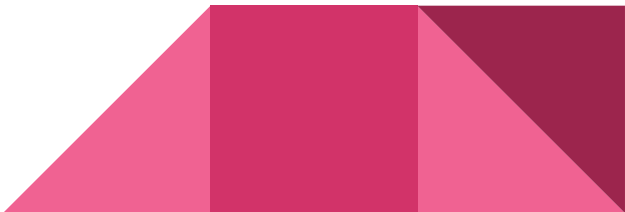
```
Int = 55
```

```
scala> (1 to 10).reduceLeft(_ + _) // shorthand
```

```
Int = 55
```

```
scala> (1 to 10).foldLeft(100)(_ + _)
```

```
Int = 155
```



reduceRight & foldRight Method

- Works same as reduceLeft and foldLeft
- Work from right to left, i.e., from the end of the collection back to the beginning.

```
scala> List(4, 8, 3).reduceRight(_ - _)  
Res4: Int = -1
```



Views

- Types of transformation
 - Strict transformation
 - Non-strict or lazy
- Whenever you create an instance you're creating a strict version of the collection
- A view makes the result non-strict, or lazy
- When it's used with a transformer method, the elements will only be calculated as they are accessed
- A view is a special kind of collection that represents some base collection, but implements all transformers lazily.



Parallel Collections

- To facilitate parallel programming by sparing users from low-level parallelization details
- Provides familiar and simple high-level abstraction.
- Provides a parallel counterpart to a number of important data structures from Scala's (sequential) collection library, including:

`ParArray`

`ParVector`

`mutable.ParHashMap`

`mutable.ParHashSet`

`immutable.ParHashMap`

`immutable.ParHashSet`

`ParRange`

`ParTrieMap`



Creating Parallel Collections

- Parallel collections are meant to be used in exactly the same way as sequential collections
- Two choices for creating a parallel collection:
 - First, by using new keyword and a proper import statement:
 - `import scala.collection.parallel.immutable.ParVector`
`val pv = new ParVector[Int]`
 - Second, by converting from a sequential collection:
 - `val pv = Vector(1,2,3,4,5,6,7,8,9).par`
- Collections that are inherently sequential, like lists, queues, and streams, are converted to their parallel counterparts by copying the elements into a similar parallel collection.
 - An example is List– it's converted into a standard immutable parallel sequence, which is a ParVector.
 - the copying required for these collection types introduces an overhead not incurred by any other collection types, like Array, Vector, HashMap, etc.

Loops In Scala

- The loop statements execute a block of expressions several times and are executed sequentially
- Loop control statements include **while**, **do while** and **for** loop



while Loop

A while loop statement is executed repeatedly until the condition is false.

The syntax for while loop is

```
while(condition) {  
  
    statements  
  
}
```



do while Loop

The do while loop statements executes at least once and then the while loop is executed till the condition is true.

The syntax for do-while loop is;

```
do {  
  
    statements  
  
} while( condition );
```



for Loop

It offer the ability to iterate over a collection, and it also provides filtering options and the ability to generate new collections.

Syntax

```
for( <variable> <- collection ){  
  statement  
}
```

