Chapter 25
Scala Collections Framework

25.1 Introduction

The collections framework is one of the main categories within the set of Scala libraries that you will work with. It provides types (Traits, Objects and Classes) that support various types of data structures (such as lists and maps) and ways to process elements within those structures. This chapter introduces the Scala collections framework.

25.2 What Are Collections?

A *collection* is a single object representing a group of objects (such as a list or dictionary). That is, they are a *collection* of other objects. Collections may also be referred to as containers (as they contain other objects). These collection classes are often used as the basis for data structures and abstract data types. In general a collection should be used wherever some significant behaviour is associated with the data in the collection (arrays can be used elsewhere). For example, a `SortedList` may need to support the idea of adding and removing elements from the list but also maintaining some *sort* order etc. Collections are thus the Scala mechanism for building data structures of various sorts; it is therefore important to become familiar with the collection API and its functionality.

Some of the collection classes, for example, `ListBuffer`, provide functionality similar to existing data structure classes such as Arrays, but are more flexible to use (at a small performance cost). For example, `ListBuffers` are growable; that is, they can grow in size as new elements are added to them, whereas Scala Arrays are of fixed size.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_25

25.3 Scala Collections

The Scala collections framework is defined within the `scala.collection` package and the subpackages associated with it. This package provides a collections framework for holding references to objects, instances and values.

The collection types incorporate higher-order functions, such as the Scala List type that includes a `foreach` function that can be used to apply an operation to each of the elements held in a collection.

The Scala collection framework is split into mutable and immutable structures that are defined in the `scala.collection.mutable` and the `scala.collection.immutable` packages. Thus all the collections in the mutable package can be updated, added to, removed from, etc. In practice this means that you can change the contents of the mutable collection after it has been created. All immutable collections, however, once created cannot change their content. Thus when you create an immutable list then the element that comprised that list cannot be removed, added to, etc. This does not mean that these collections do not include operations that appear to add, remove or otherwise alter their contents; rather that these operations produce a copy of the original collection which reflects the changes being made.

Note that by default the contents of the `scala.collection.immutable` package is imported into all Scala source files and thus you need to explicitly import the collection types in the `scala.collection.mutable` package.

All Scala collections have type parameters which can be used to specify the type contained within the collection. For example

```
object CollectionsApp extends App {
  val myList = List[String]("One", "Two", "Three")
  println(myList)
}
```

This specifies that we wish to create a new List that will hold references to Strings and is initialised with the strings “One”, “Two” and “Three”. We can often get Scala to infer the type of the collection for us but such type parameterisation of collections is very common. The output of this application is

```
List(One, Two, Three)
```

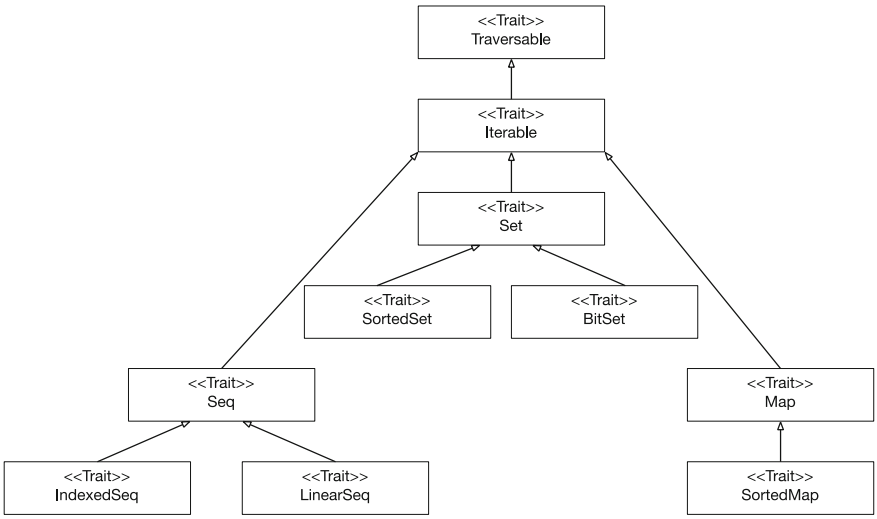
The Scala collections framework was heavily revised in Scala 2.8 to provide a common, uniform framework for collection types. A number of key design principles underpin the framework, and these are:

- **Ease of Use**. Most problems can be solved with just a couple of operations, which are common across the framework.
- **Concise**. Due to the presence of higher-order functions within the framework, such as the *foreach* function, activities that would have taken several statements in other collection frameworks can be achieved in a single statement.
- **Safe**. The presumption of immutability and the avoidance of side effects make the use of collection types much safer in Scala.
- **Fast**. The operations within the collections framework have been heavily tuned and optimised to maximise performance.
- **Universal**. A common set of operation exists across all the collection types. Thus one collection type has similar operations to another collection type that is similar in nature. This means that developers only need to learn a fairly small vocabulary to be able to use a wide range of types.
- **Expressive**. The vocabulary that is used is expressive and semantically meaningful helping developers to clearly express the intent of their code. For example, val (minors, adults) = people.partition(_.age < 18).

The packages that comprise the Scala collection framework are:

- scala.collection. This package contains the root collection types. These root types may be mutable or immutable. For example, scala.collection.IndexedSeq is a parent of both collection.immutable.IndexedSeq and of scala.collection.-mutable.IndexedSeq. In general the operations defined on the type in the scala.collection package are repeated in the immutable type and extended on in the mutable type.
- scala.collection.mutable. This package contains the mutable versions of the collection types.
- scala.collection.immutable. This package contains the immutable versions of the collection types.
- scala.collection.generic. This package provides building blocks for constructing collection types; you would not normally need to work this package.

25.3.1 Package Scala.Collection



There is a relatively small set of collection types in the scala.collection package. Some of these types are presented above and described briefly below. This hierarchy was introduced in Scala 2.8 and forms the foundation of the current (including Scala 2.12) collections types (however the Scala 2.13 version is exploring a rework of the collections hierarchy if you are using this version then check the online reference material).

The major of the types are traits although some also have objects that support utility type operations such as the Map object with methods such as apply, empty.

- **Traversable**. This trait is the root trait of all traversable collections. It is the base trait for all kinds of Scala collections. It provides the common behaviour common to all collections (such as the foreach method). Thus all collection types that mix in this trait must implement a foreach operation.
- **Iterable**. A base trait for iterable collections. An iterable collection is one that it is possible to iterate over. This means that it is possible to process each element in an iterable collection one-by-one. Implementations of this trait need to provide a concrete method with the signature def iterator: Iterator[A].
- **Seq**. This is a base trait for all sequence like collections. A sequence is a collection in which there is a specific order to the elements it holds. Sequences provide an apply method for indexing, operations such as indexOf, lastIndexOf, startsWith, endsWith and reverse.
- **IndexedSeq**. This is a base trait for indexed sequences. An indexed sequence is a sequence where each element is accessible by an index and where element

access is supported in constant time (or near constant time) for elements across the collection. An IndexedSeq provides fast random access to elements and a fast length operation.
- **LinearSeq**. The base trait for linear sequences such as Lists and Streams. A LinearSeq provides fast access only to the first element, via the head, but also fast tail operation.
- **Set**. The base trait for all sets (both mutable and immutable). A Set does not allow a duplicate of en element.
- **SortedSet**. The SortedSet trait represents a Set that has been sorted.
- **BitSet**. A base trait for BitSets. BitSets are sets of nonnegative integers that can be used to represent variable-size arrays of bits packed into 64-bit words.
- **Map**. This is a trait that represents the key concepts implemented by all Maps. A map is a collection that maintains relationships between keys and values.
- **SortedMap**. A SortedMap trait is a Map whose keys are sorted.

25.3.2 Common Seq Behaviour

The following lists some of the common behaviour shared by all Seq types.

- seq(n) Return the element n in the sequence seq. Note that Scala collections are Zero based and therefore the first element in the sequence will be the element '0'. This means that if the sequence holds 5 elements they will be accessed by the indexes 0–4.
- seq.length Return the length of the sequence.
- seq.lengthCompare(s2) Returns −1 if seq is shorter than s2 and +1 it is longer, with Zero indicating that they have the same length.
- seq.indexOf(x) The index of the first element in seq equal to x.
- seq.lastIndexOf(x) Return the index of the last element in seq that equals x.
- seq.indexOfSlice(s2) The first index of seq such that the successive elements starting from that index from the sequence s2.
- seq +: x A new sequence that consists of x prepended to seq.
- seq :+ x A new sequence that consists of x appended to seq.
- seq.padTo(length, x) The sequence resulting from appending the value x to seq until the length is reached.
- seq(i) = x Changes the element of seq at index I to be x.
- seq.sorted A new sequence obtained by sorting the elements of seq using the standard ordering of the element type contained within seq.
- seq.sortWith(test) A new sequence obtained by sorting the elements of seq using the 'test' as the comparison operation.
- seq.reverse Returns a sequence with the elements of seq in reverse order.
- seq.iterator Returns an iterator yielding each of the elements of seq in order.
- seq.reverseiterator Returns an iterator yielding each of the elements of seq in reverse order.

- seq.contains(x) Tests whether seq contains an element equal to x.
- seq.startsWith(s2) Returns true if seq starts with the sequence contained in s2.
- seq.endsWith(s2) Returns true if seq ends with the sequence contained in s2.
- seq.intersect(s2) The multi-set intersection of sequences seq and s2 that preserves the order of the elements in seq.
- seq.diff(s2) The multi-set difference of sequences seq and s2 that preserves the order of the elements in seq.

25.3.3 Common Set Behaviour

The following lists some of the common behaviour shared by all Set types.

- set.contains(x) Tests whether set contains an element equal to x.
- set(x) Same as set.contains(x).
- set.subsetOf(s2) Tests whether set is a subset of s2.
- set + x Returns the set containing all the elements of set plus x.
- set ++ s2 A set containing the union of all the elements of set and s2.
- set − x A set containing all the elements of set except x.
- set − s2 A set containing all the elements of set except those elements also in s2.
- set.empty Test to see if the set is empty.
- Set interest s2 The set intersection of set and s2.
- Set diff s2 The set difference between set and s2.

25.3.4 Common Map Behaviour

The following lists some of the common behaviour shared by all Map types.

- m.contains(k) Tests whether m contains a mapping for key k.
- m.get(k) Retrieves the value associated with the key k as an Option or None if there is no mapping.
- m(k) Retrieves the value associated with the key k or an exception if not mapping is present.
- m.getOrElse(k, d) Retrieves the value associated with the key k or the default value d if no key is found.
- m + (k –> v) Add the mapping k –> v to the map.
- M − k Remove the key k (and its associated value) from the map m.
- m.keys Return an iterable containing each of the keys in the map m.
- m.keySet Return a set containing all the keys in the map m.
- m.keyIterator Return an iterator for each of the keys in the map m.
- m.values An iterable containing each of the values in the map m.
- m.valuesIterator An iterator over the values in the map m.