Chapter 26
Immutable Lists and Maps

26.1 Introduction

This chapter focuses on the List and Map types from the `scala.collection.immutable` package.

26.2 The Immutable List Collection

As a common approach is applied across all the collection classes in Scala we will first use the `scala.collection.immutable.List` class as a detailed case study to look at the facilities it provides and the operations that can be performed on it.

A `List` is an immutable (fixed) sequence of elements that provides for constant time access to the first element and to the tail (remaining) elements. Many other operations take linear time. Lists are a very widely used collection as they provide all the core sequence like behaviours.

26.2.1 List Creation

The following listing illustrates some of the list creation options available:

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_26

```
object CollectionsApp extends App {
  // List creation options
  val myList0: List[String] =
                    List[String]("One", "Two", "Three")
  val myList1 = List[String]("One", "Two", "Three")
  val myList2 = List("One", "Two", "Three")
}
```

The first option illustrated by `myList0` creates a list containing strings initialized to "One", "Two" and "Three". This is saved into a val `myList0` which is specified to be of type `List[String]`. The second example illustrates how Scala can infer the type of the val `myList1` for us, and the third indicates that Scala can determine the type of the contents of the list so that it is not necessary to explicitly state that the list will contain Strings.

26.2.2 List Concatenation

We can concatenate strings together using the '`:::`' operator. This creates a new list based on two existing lists, for example,

```
// The list concatenation method that creates a new list
// based on existing lists
val longList = myList0 ++ myList1
println(longList)
```

This results in `longList` now containing the list:

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

Note that in the older versions of Scala the '`:::`' operator was available instead of the '`++`'. However, the effect will be the same whichever operation you select to use.

It is also possible to use the *cons* operator ('`::`') to prepend an element to the front of a list (which takes constant time), for example,

```
// The cons method that prepends a new element
// to the beginning of a list - takes constant time
val newList = "Zero" :: myList2
println(newList)
```

Note that this of course produces a new list referenced by `newList` as Lists are immutable. The contents of `newList` is:

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

The cons operator can also be used to construct a list from a set of existing values. For example,

```
val myList3 = "One" :: "Two" :: "Three" :: Nil
println(myList3)
```

This example may look a little strange. This is because the value `Nil` at the end of the statement represents an empty list. It is thus to this list that the strings “One”, “Two” and “Three” are being added. This raises the question, why is Nil at the end of the statement and not at the start? This is because any method or operation that ends with a ‘:’ is right associative. Thus the expression:`"Three" :: Nil`

must be read from the right to the left (and not from the more traditional left to right). Thus the String “Three” is being prepended to the empty list represented by `Nil`. This expression then returns the new list containing the String “Three”. The String “Two” is then prepended to that list by the next ‘::’ cons operation. This result sin a new List containing “Two” and “Three”. The string “One is then prepended to that List to create the final list containing “One”, “Two” and “Three”. As prepending is done in constant time is the preferred way to add an element to the List. The result is

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

There is also the ‘+:’ operator which again prepends an element to the front of the list return a new copy of the list.

We could also append an element to the contents of the existing list using the ‘:+’ operator. However, this is rarely used as the time it takes to append to the list grows linearly as the size of the list grows:

```
val endList = myList2 :+ "End"
println(endList)
```

The val `endList` would now hold a reference to the following list:

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

26.2.3 List Operations

Other operations of interest on the List class are illustrated in the following listing and discussed below:

```
object ListOpsApp extends App {
  // Create a list of numbers
val numbers = List(1, 2, 3, 4, 5)
println(numbers)
// Determine the length of the list
println("length of the list: " + numbers.length)
// Reverse the list
val rv = numbers.reverse
println("Reversed: " + rv)
// returns the list without its first 2 objects
println("Drop first two objects: " + numbers.drop(2))
// Returns the first element
println("The first element: " + numbers.head)
// Returns the last element
println("The last Element: " + numbers.last)
// Returns the list minus the first element
println("The tailed list: " + numbers.tail)
// Returns the list minus the last element
println("The init part of the list: " + numbers.init)
// Tests to see if the list is empty
println("Is the list Empty: " + numbers.isEmpty)
// Convert the list into a String
val s = numbers.mkString(",")
println("String format of list: " + s)
}
```

The method `length` returns the length of the list while the method `reverse` returns the list in reverse order. The `head` of the list returns the first element in the list and the method `last` returns the last element in the list. The method `tail` returns all the elements in the list bar the first element. The method `isEmpty` returns a Boolean `true` of `false` depending upon whether the list is empty or not and the .`mkString` method converts the contents of the list into a string with each element in the list separated by the string passed into the method. The result of executing `Test2` is shown below:

```
List(1, 2, 3, 4, 5)
length of the list: 5
Reversed: List(5, 4, 3, 2, 1)
Drop first two objects: List(3, 4, 5)
The first element: 1
The last Element: 5
```

```
The tailed list: List(2, 3, 4, 5)
The init part of the list: List(1, 2, 3, 4)
Is the list Empty: false
String format of list: 1,2,3,4,5
```

26.2.4 List Processing

You can also process all the elements in the list. This can be done in a number of ways. For example, lists are iterable collections and thus you can iterate over the elements in the list as follows:

However, many of the collection classes also provide the `foreach` higher-order function that can take a function to apply to each of the elements in the List in turn. Therefore you can also write:

```
object ListProcApp extends App {
  val myList = List[String]("One", "Two", "Three")
  myList.foreach((x: String) => {println(x)})
}
```

Of course in Scala this can be written in a number of different (and shorter) ways, including:

```
// More Scala oriented equivalent
myList.foreach((e) => {println(e)})
myList.foreach(e => {println(e)})
myList.foreach(e => println(e))
myList.foreach(println (_))
// Shortest form of above
myList.foreach(println)
```

where we gradually rely in Scala to infer more and more of what is being defined.

26.2.5 Further List Processing

As well as the `foreach` operation, there is a whole range of higher order functions that you can use with Lists. These include `filter`, `map`, `foldLeft` and `foldRight`, `flatMap`.

For example, if you wish to select only certain elements in a list, that meet a specific criterion, you can do that using the filter higher order function. For example,

```
// Create a list of numbers
val numbers = List(1, 2, 3, 4, 5)
println(numbers)
// Apply a function used to filter the members
// of the list and create a new list
val f = numbers.filter(n => n < 3)
println("Filtered: " + f)
```

This would result in the following output:

```
List(1, 2, 3, 4, 5)
Filtered: List(1, 2)
```

As you can see from this example, the result of applying the function literal passed into the filter is that only elements less than three have been included in the filter list 'f'.

We can also apply a function to each of the elements in a list and create a new list of the same size, based on the result returned by the function we applied. For example,

```
// Create a list of numbers
val numbers = List(1, 2, 3, 4, 5)
// Apply a function to each of the members of the list
// and create a new list of the same size
val m = numbers.map(n => n + 10)
println("Modified list: " + m)
```

The output from this is

```
Modified list: List(11, 12, 13, 14, 15)
```

Thus the modified list has five elements, and these elements are each 10 greater than the elements in the original list. Thus the function:

```
n => N + 10
```

has added 10 to each of the elements and collated the results into a new list referenced by 'm'.

We could have written this in a short from as:

```
// Short hand form of the above.
// That is {n=>n+10} can be rewritten as (_+10) which means
// you do not need to declare the input param
val m2 = numbers.map(_ + 10);
println("Modified List (alternative form): " + m2)
```

Which would also result in a list being created containing five elements, where each element is 10 greater than that in the original list:

```
Modified List (alternative form): List(11, 12, 13, 14, 15)
```

We can also apply a function to all the elements in the list and gather up the results into a single value. This can be done using either the `foldLeft` or the `foldRight` methods. The `foldLeft` operation takes an initial value (or state) and propagates that state from one element to the next, with the result of one evaluation being passed as input to the next. It starts from the left most element and processes right towards the end of the list. For example, given our list of numbers we could add them all up using the `foldLeft` operation:

```
// Create a list of numbers
val numbers = List(1, 2, 3, 4, 5)
val sum = numbers.foldLeft(0)((total, element) => total +
   element)
println("Sum of List " + sum)
```

The result of this is:

```
Sum of List 15
```

Note that `foldLeft` is a multi-argument list operation, thus the first argument takes the initial value to use (in this case Zero) and the second argument list takes the function to apply. This function is a two parameter function, one which takes the total already calculated and one parameter that is the current element to process. The result return from this function is then passed to the next invocation of the function. Note due to the alternative syntax available for single parameter lists the above could also be written as the following with the second parameter list being represented by the curly braces '{…}':

```
// Create a list of numbers
val numbers = List(1, 2, 3, 4, 5)
val sum = numbers.foldLeft(0){(total, element) => total +
   element}
println("Sum of List " + sum)
```

There is also a `foldRight` operation which starts at the end of the list with the last element and processes towards the start of the list. For example,

```
// Also an option to start from the right
// and process towards the start
val alternativeSum = numbers.foldRight(0)
     {(total, element) => total + element }
println("Alternative Sum of List " + alternativeSum)
```

The result (in this case) is exactly the same, `alternativeSum` contains the value 15:

```
Alternative Sum of List 15
```

However, there are some issues with `foldRight` when it comes to processing very large lists and thus it is often better to reverse a list and then call `foldLeft`, for example,

```
myList.reverse.foldLeft(0){(t, e) => t + e}
```

Another operation `flatten` can be used to flatten a list. That is, given a list of lists it can return a single list constructed from the contents of the original sublists. For example,

```
scala > val nested = List(List(1, 2, 3), List(4, 5))
nested: List[List[Int]] = List(List(1, 2, 3), List(4, 5))
scala > nested flatten
res0: List[Int] = List(1, 2, 3, 4, 5)
```

In the above example, the first sublist containing 1, 2 and 3, and the second sublist containing 4 and 5 have been flattened into a single list containing 1, 2, 3, 4 and 5.

The operation `flatMap` is essentially map plus `flattern` combined together. The function given to the `flatMap` is expected to return a list of values. The resultant list of lists is then flattened into a single list. For example, given a val

'contents' containing a list of Arrays of Integers. In the following example, we use `flatMap` to convert the array to a list and then to flattern the two lists into a single list:

```
val contents = List(Array(1, 2, 3), Array(4, 5, 6))
val result = contents.flatMap(x => x.toList)
println(result)
```

The output from this program is:

```
List(1, 2, 3, 4, 5, 6)
```

26.2.6 Pattern Matching

It is possible to extract the contents of a list into a set of variables as follows:

```
val myList = List("a", "b", "c")
val List(x, y, z) = myList
```

with the result that `x`, `y` and `z` are variables containing the contents of the `myList` (rather than being the members of a new list):

```
x:    java.lang.String = a
y:    java.lang.String = b
z:    java.lang.String = c
```

However, this has limited utility unless you know the number of elements in the list. A generally more useful approach is to extract the head and the tail of a list in this way:

```
val hd :: tl = myList
```

which results in two vals being created one called `hd` and containing the string "a" and one called `tl` containing the list of strings ("b" and "c"):

```
hd: java.lang.String = a
tl: List[java.lang.String] = List(b, c)
```

26.2.7 Converting to a List

There are numerous ways in which you can convert other sequence like constructs into lists. For example, you can convert an array to a list:

```
scala > Array(1, 2, 3, 4) toList
List[Int] = List(1, 2, 3, 4)
```

Or a string to a list (which results in a list of characters);

```
scala > "abc" toList
List[Char] = List(a, b, c)
```

Or a generated sequence into a list

```
var shortList = 1 to 10 toList
```

And indeed other collection types into a list, such as a Set or a Map:

```
scala > Set("abc", 123) toList
List[Any] = List(abc, 123)
scala > Map("apple" -> "red", "banana" -> "yellow") toList
List((apple,red), (banana,yellow))
```

Of course the reverse is also true. That is it is possible to convert a list to a range of sequence like structures using various to <Type> methods, such as `toArray`, `toString`, `toSet`, `toMap`.

26.2.8 Lists of User Defined Types

All of the examples we have looked at so far with Lists have used strings or Integers. We can of course also make lists of user-defined types. For example, given the class `Person` as can create lists which hold `Person` types:

```
val dad = new Person("John", 49)
val mum = new Person("Denise", 46)
var adam = new Person("Adam", 14)
var phoebe = new Person("Phoebe", 16)

val family = List[Person](dad, mum, adam, phoebe)
```

This defines four vals holding references to four istances of a class Person. These four vals are then used to initialise a List of Persons referenced by the val `family`. We have explicitly stated the type of the contents of the List as `Person` in this case, but that is of course optional, and we could also have written:

```
val family = List(dad, mum, adam, phoebe)
```

In the above example, Scala will infer that this is a list of Person types.

We can now apply the operations discussed earlier in this section with this list of persons. Of course we can now access the properties and methods defined on the class `Person` within the functions we apply to the elements of the list. Thus assuming the class `Person` is defined as follows:

```
case class Person (var name: String, var age:Int)
```

We can write:

```
val family = List(dad, mum, adam, phoebe)

// Note Scala can infer the parameter:
family.foreach{println("Family Member: " + _) }

// get everyone over the age of 21
val over21 = family.filter { _.age > 21 }
println(over21)

// Extract the ages and find the average
val ages = family.map(_.age)
println(ages)
val averageAge = ages.sum / ages.size
println("Average age: " + averageAge)
```

The examples illustrate the use of `foreach`, `filter` and `map`. The `foreach` example prints out each member of the family with a prefix of “Family member:”. The second filters the family members to find those over 21 using the age property. The final example uses the map function to obtain a list containing the ages of each

of the members of the family which is then used to calculate the average of all the ages. The output of this program is:

```
Family Member: Person(John,49)
Family Member: Person(Denise,46)
Family Member: Person(Adam,14)
Family Member: Person(Phoebe,16)
List(Person(John,49), Person(Denise,46))
List(49, 46, 14, 16)
Average age: 31
```

26.3 The Immutable Map Type

A `Map` is a set of associations, each representing a key-value pair. The elements in a `Map` may be unordered, but each has a definite name or *key*. Although the values may be duplicated, keys cannot. In turn a key can *map* to at most one value. Some Map implementations, like `TreeMap` and `ArrayMap`, make specific guarantees as to their order; others, like `HashMap`, do not. The `Map` protocol allows either the keys to be viewed, the values to be viewed or for the keys to be used to access the values.

A Concrete implementation of the `Map` trait is the `scala.collection.immutable.HashMap` class. It provides an immutable implementation of a Map based on the use of a hashing trie. A hash trie is a standard way to implement immutable sets and maps efficiently within Scala. To find a given key in a map, the code first takes the hash code of the key and based on information held in the hash find the appropriate *bucket* into which the key value pair will have been placed. The advantage of hash tries are that they strike a balance between reasonably fast lookup and reasonably efficient inserts and deletions.

The following listing illustrates some of the operations available on the `Map` trait that is implemented by the `HashMap` class.

```
import scala.collection.immutable.HashMap

object HashMapTest extends App {
   val capitalCitys =
            HashMap("UK" -> "London",
                    "FRANCE" -> "Paris",
                    "Spain" -> "Madrid",
                    "USA" -> "Wasington. DC")
   println(capitalCitys.size)
   println(capitalCitys.keys)
   println(capitalCitys.values)
   println(capitalCitys.isEmpty)
   println(capitalCitys.get("UK"))
   println(capitalCitys("UK"))
   println(capitalCitys.contains("UK"))
   println(capitalCitys.getOrElse("Ireland",
                                  "Not known"))
   val newCapitalCitys =
             capitalCitys +("Ireland" -> "Dublin")
   println(newCapitalCitys("Ireland"))
}
```

The result of executing this program is shown below:

```
4
Set(USA, Spain, UK, FRANCE)
MapLike(Wasington. DC, Madrid, London, Paris)
false
Some(London)
London
true
Not known
Dublin
```

The points to note about this listing include that a map contains key to value pairs. Thus the size of the map is four elements just after it is instantiated. Also note that you can obtain the keys and values in the map separately should you need to do so. The keys are returned as a `Set` as they will be unique. The Values are returned as a type of sequence as there may be duplicates amongst them. You can also test a map to see if it is empty. The method `get` and the access (nth) are often treated as synonymous, but they actually have different return types, for example,

```
println(capitalCitys.get("UK"))
```

Returns

```
Some(London)
```

Where `as`

```
println(capitalCitys("UK"))
```

Returns

```
London
```

The difference being that get will return None if the key does not exist in the `HashMap` were as the *accessor* will throw an exception if the key is not present.

An alternative is to use the `getOrElse` method:

```
println(capitalCitys.getOrElse("Ireland", "Not known"))
```

This will return the value associated with the specified key (in this case Ireland) or if the key is not present it will return the value passed to the method as the second parameter.

Finally note that act of adding a new key-value pair to the map results in a new map being created containing the same set of key-value pairs as the original map with the addition of the new key-value pair (the original map is unaffected):

```
val newCapitalCitys = capitalCitys +("Ireland" -> "Dublin")
```