Lecture 4
Immutable Collections

4.3.2023

Iflaah Salman

LUT University

Collections

- A collection is a single object representing a group of objects (such as a list).
- The collection classes are often used as the basis for data structures and abstract data types.
  - For example, List, Sets.
- A collection should be used when we want to associate some significant behaviour with the data in the collection.
  - For example, SortedList: to maintain some order despite adding and removing elements.
- The Scala collections framework is defined within the **scala.collection** package and its sub-packages, split into *mutable* and ***immutable*** (data) structures
  - **scala.collection.immutable** once the collection is created, we cannot change its contents.

The collection types incorporate higher-order functions!

LUT University

Collections: Key Design Principles

- Ease of Use
  - Most problems can be solved with just a couple of operations
- Concise
  - Use of higher order functions has made operations concise, e.g., **foreach** function.
- Safe
  - The presumption of immutability and the avoidance of side effects make the use of collection types much safer in Scala.

Collections: Key Design Principles

- Fast
  - Operations are tunned and optimised to maximise performance
- Universal
  - A common set of operations exists across all the collection types; easy for developers from the learning perspective.
- Expressive
  - The vocabulary that is used is expressive and semantically meaningful; helping developers to express the intent of their code.

Trait

- Traits can play in developing reusable behaviour that simplifies the development of new types.

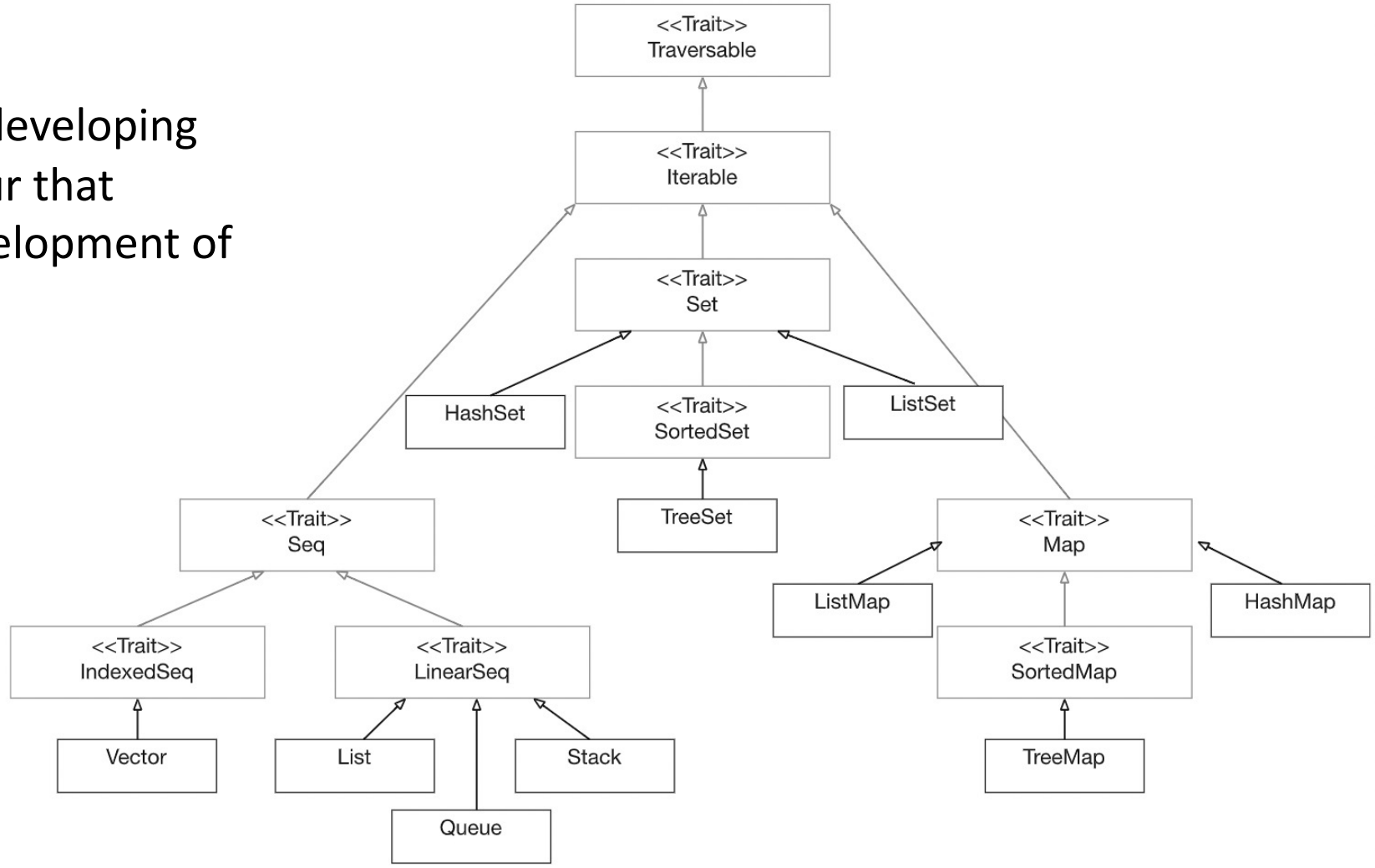


Fig. 27.1 Key classes and traits in the scala.collection.immutable package

LUT University

Immutable Lists

List

- An immutable fixed sequence of elements
- Constant time access to the first element and to the tail elements.
- Other operations take linear time.
- Provides the core Sequence like behaviours
  - A sequence is a collection in which there is a specific order to the elements it holds.

List creation options

```
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")
}
```

LUT University

Immutable Lists

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

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

in the older versions of Scala the ‘:::’ operator was available instead of the ‘++’

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

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

LUT University

Immutable Lists

cons operator to construct a list from a set of existing values.

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

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

- 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.
- 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.

- ‘+:’ operator which again prepends an element to the front of the list and returns a new list.
  ‘:+’ operator to append an element to the contents of the existing list – takes linear time.

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

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

Immutable Lists

```scala
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)
```

```
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
```

Immutable Lists

```
val s = numbers.mkString(",")
println("String format of list: " + s)
```

```
String format of list: 1,2,3,4,5
```

- the **.mkString** method converts the contents of the list into a string.
- each element in the list separated by the string passed into the method.

List Processing

- We can process elements in the List because it is an iterable collection.
- **`foreach`** is a higher-order function that can take a function to apply to each element in the List in turn.

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

List Processing via HOFs

```
// 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)
```

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

filter: to select only certain elements in a list that meet a specific criterion.

```
// 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)
```

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

map: apply a function to each of the elements in a list and create a new list of the same size

List Processing via HOFs!

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

```
// 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)
```

foldleft is preferred over **foldright** when the lists are large. Therefore, it is better to **reverse** the list and apply **foldleft**

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

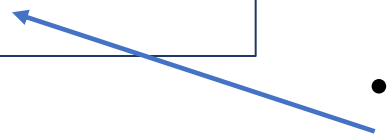
- **foldLeft** or **foldRight:** to apply a function to all the elements in the list and gather the results into a single value.
- **foldLeft** is a multi-argument list operation
- **foldLeft** operation takes an initial value (or state) and propagates with the result of one evaluation being passed as input to the next.
- It starts from the leftmost element and processes towards the right end of the list.
- The first argument takes the initial value to use and the second argument list takes the function to apply.
- The result return from this function is then passed to the next invocation of the function.

List Processing via HOFs!

```
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)
```

- **flatten** can be used to flatten a list.
- given a list of Lists, it can return a single list.

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



- **flatMap** is essentially ***map*** plus ***flatten***
- The function given to **flatMap** is expected to return a list of values.
- **flatMap** is used to convert the array to a list and then to ***flatten*** the two lists into a single list.

Converting to a List

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

Array to a List

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

String to a List

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

Generated sequence to a List

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

Set or Map to List

A *Set* does not allow a duplicate of an element.

Lists of user-defined 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)
```

```
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)
```

We can now also access the properties and methods defined in the class Person within the functions we apply to the elements of the list.

Immutable Map

- A Map is a set of associations, each representing a key-value pair.
- The values can be unordered and can be repeated, but every value has a key and keys are unique.
- Some Map implementations, like TreeMap and ArrayMap, guarantee a specific order.
- **HashMap** implementation does not guarantee a specific order.
  - It is a concrete immutable implementation based on Hash trie.
  - 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 finds the appropriate bucket into which the key-value pair would have been placed.
  - Hash trie strikes a balance between reasonably fast lookup and reasonably efficient inserts and deletions.

hash trie can refer to: Hash tree, a trie used to map hash values to keys.

Source: wikipedia

HashMap

```
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"))
}
```

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

- 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.
- The method get and the access (nth) are often treated as synonymous, but they have different return types. **get** may return *None,* but **access** will throw an exception.
- An alternative to **get** or **access** is **getOrElse.**

Sequences

- **Vector**
  - A general-purpose immutable indexed sequence.
  - you might choose to use a Vector over a List as it provides faster access.

```
object VectorTest extends App {
  val v1 = Vector(3, 2, 1)
  println(v1)
  println(v1(0))
  println(v1.length)
  val v2 = 4 +: v1
  println(v2)
}
```

```
Vector(3, 2, 1)
3
3
Vector(4, 3, 2, 1)
```

Sequences

- Queue
  - A Queue is a first-in first-out (FIFO) type of collection.
  - Primary methods are **enqueue** and **dequeue**.
    - return a copy of the original queue with a new element added or an element removed.

```
import scala.collection.immutable.Queue

object QueueTest extends App {
  val q1 = Queue[Int]()
  val q2 = q1.enqueue(1)
  println(q2)
  val q3 = q2.enqueue(List(2, 3))
  println(q3)
  val (r, q4) = q3.dequeue
  println(r)
  println(q4)
}
```

```
Queue(1)
Queue(1, 2, 3)
1
Queue(2, 3)
```

Sequences

- **Set**
  - **HashSet** is an immutable Set implementation based on a hashing function.
  - Only allows a single instance of an element in the Set.

```
// Create an immutable Set
var teams =
     HashSet("Liverpool", "West Ham",
         "Newcastle", "Everton", "West Ham")
println(teams)
```

```
HashSet(Liverpool, West Ham, Newcastle, Everton)
```

  - **ListSet**
    - set using a list-based structure internally
    - Or a List that restricts the occurrences of some element to a single occurrence.

```
var t2 = ListSet("Liverpool", "West Ham",
"Newcastle", "Everton", "West Ham")
  println(t2)
```

```
ListSet(Everton, Newcastle, West Ham, Liverpool)
```

Sequences

Maps

- ListMap
  - a map that uses a linked list-based structure to internally represent the key-value pairs
  - Operations on a list map take linear time relative to the size of the map.
    - hashMap is a better choice.
- HashMap
  - collection of associated keys and values that are organized based on the hash code of the key.
- TreeMap
  - Implements the map as a tree structure.

References

- Hunt, J. (2018). A Beginner's Guide to Scala, Object Orientation and Functional Programming. In *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Springer International Publishing. https://doi.org/10.1007/978-3-319-75771-1