Scala Collections

Using List Collection

 It is an immutable singly linked list.

**Creating lists :**

1. val numbers = List(32, 95, 24, 21, 17)

val colors = List("red", "green", "blue")

println(s"I have ${colors.size} colors: $colors")

1. //LISP style approach

val list = 1 :: 2 :: 3 :: Nil

1. //Creating Empty List.

val empty: List[Nothing] = List()

1. //Using fill method

val fruit = List.fill(3)("apples") // Repeats apples three times.

println( "fruit : " + fruit )

val num = List.fill(10)(2) // Repeats 2, 10 times.

1. // Creates 5 elements using the given function.

val squares = List.tabulate(6)(n => n \* n)

1. //Using range method

val x = List.range(1,10) //adds all the elements from 1 to 10

val x = List.range(0,10,2) //third argument is step value

**Iterating lists :**

1. for (c <- colors) { println(c) }
2. colors.foreach( (c: String) => println(c) )
3. colors.foreach(println)

foreach function takes an argument and invokes each element with that function

**Map functions :**

map() takes a function that converts a single list element to another value and/or type. This is mainly used for conversion and getting a new collection e.g

colors list is iterated and for every element size is found(length) out and new collection is returned

**val sizes = colors.map( (c: String) => c.size )**

**sizes: List[Int] = List(3, 5, 4)**

Some built in mapping functions :

|  |  |  |
| --- | --- | --- |
| collect | List(0, 1, 0) collect {case 1 => "ok"} | Transforms each element using a partial function, retaining applicable elements. |
| flatMap | List("milk,tea")flatMap (\_.split(',')) | Transforms each element using the given function and “flattens” the list of results into this list. |
| map | List("milk","tea")map (\_.toUpperCase) | Transforms each element using the given function. |

**Reducing Lists :**

List reduction is a common operation for working with collections. Need to sum up a list of grades, or calculate the average duration of several benchmarks. reduce() takes a function that combines two list elements into a single element.

val total = numbers.reduce( (a: Int, b: Int) => a + b )

There are some built in mathematical reduction operations available

1. max

List(41, 59, 26).max

Finds the maximum value in the list.

1. min

List(10.9, 32.5, 4.23, 5.67).min

Finds the minimum value in the list.

1. product

List(5, 6, 7).product

Multiplies the numbers in the list.

1. sum

List(11.3, 23.5, 7.2).sum

Sums up the numbers in the list.

**Some more operations on the list :**

|  |  |  |
| --- | --- | --- |
| :: | 1 :: 2 :: Nil | Appends individual elements to this list. |
| ::: | List(1, 2) ::: List(2, 3) | Prepends another list to this one.  A right-associative operator. |
| ++ | List(1, 2) ++ Set(3, 4, 3) | Appends another collection to this list. |
| == | List(1, 2) == List(1, 2) | Returns true if the collection types and contents are equal. |
| distinct | List(3, 5, 4, 3, 4).distinct | Returns a version of the list without duplicate elements. |
| drop | List('a', 'b', 'c', 'd') drop 2 | Subtracts the first *n* elements from the list. |
| filter | List(23, 8, 14, 21) filter (\_ > 18) | Returns elements from the list that pass a true/false function. |
| flatten | List(List(1, 2), List(3, 4)).flatten | Converts a list of lists into a single list of elements. |
| partition | List(1, 2, 3, 4, 5) partition (\_ < 3) | Groups elements into a tuple of two lists based on the result of a true/false function. |
| reverse | List(1, 2, 3).reverse | Reverses the list. |
| slice | List(2, 3, 5, 7) slice (1, 3) | Returns a segment of the list from the first index up to but not including the second index. |
| sortBy | List("apple", "to") sortBy (\_.size) | Orders the list by the value returned from the given function. |
| sorted | List("apple", "to").sorted | Orders a list of core Scala types by their natural value. |
| splitAt | List(2, 3, 5, 7) splitAt 2 | Groups elements into a tuple of two lists based on if they fall before or after the given index. |
| take | List(2, 3, 5, 7, 11, 13) take 3 | Extracts the first *n* elements from the list. |
|  |  |  |

**Set :**

A Set is an immutable and unordered collection of unique elements, but works similarly to List. Here is an example of creating a Set with duplicate items. As another subtype of Iterable, a Set instance supports the same operations as a List instance does.

val unique = Set(10, 20, 30, 20, 20, 10)

Outcome : unique: scala.collection.immutable.Set[Int] = Set(10, 20, 30)

**Map :**

A Map is an immutable key-value store, also known as a hashmap, dictionary, or associative array in other languages. Values stored in a Map with a given unique key may be retrieved using that key. The key and the value are type-parameterized.

When creating a Map, specify the key-value pairs as tuples.  You can use the relation operator (->) to specify the key and value tuple.

e.g

val colorMap = Map("red" -> 0xFF0000, "green" -> 0xFF00, "blue" -> 0xFF)

outcome : colorMap: scala.collection.immutable.Map[String,Int] =

Map(red -> 16711680, green -> 65280, blue -> 255)

1. To access element inside the collection

val redRGB = colorMap("red")

1. Iterating the map :
2. for (pairs <- colorMap) { println(pairs) }
3. for ((k,v) <- m1) printf("key: %s, value: %s\n", k, v) //where m1 is the map

c. m1 foreach {case (key, value) => println (key + "-->" + value)}

1. Adding, removing, and updating mutable Map elements

To create a mutable Map, import it first:

var states = scala.collection.mutable.Map("AL" -> "Alabama")

// add elements with +=

states += ("AZ" -> "Arizona")

states += ("CO" -> "Colorado", "KY" -> "Kentucky")

// remove elements with -=

states -= "KY"

states -= ("AZ", "CO")

// update elements by reassigning them

states("AK") = "Alaska, The Big State"