SCALA LISTS

Scala Lists are quite similar to arrays which means, all the elements of a list have the same type but there are two important differences. First, lists are immutable, which means elements of a list cannot be changed by assignment. Second, lists represent a linked list whereas arrays are flat.

The type of a list that has elements of type T is written as **List[T]**. For example, here are few lists defined for various data types:

```
// List of Strings
val fruit: List[String] = List("apples", "oranges", "pears")

// List of Integers
val nums: List[Int] = List(1, 2, 3, 4)

// Empty List.
val empty: List[Nothing] = List()

// Two dimensional list
val dim: List[List[Int]] =
   List(
      List(1, 0, 0),
      List(0, 1, 0),
      List(0, 0, 1)
   )
```

All lists can be defined using two fundamental building blocks, a tail **Nil** and **::**, which is pronounced **cons**. Nil also represents the empty list. All the above lists can be defined as follows:

```
// List of Strings
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

// List of Integers
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

// Empty List.
val empty = Nil

// Two dimensional list
val dim = (1 :: (0 :: (0 :: Nil))) ::
          (0 :: (1 :: (0 :: Nil))) ::
          (0 :: (0 :: (1 :: Nil))) :: Nil
```

Basic Operations on List:

All operations on lists can be expressed in terms of the following three methods:

Methods	Description
head	This method returns the first element of a list.
tail	This method returns a list consisting of all elements except the first.
isEmpty	This method returns true if the list is empty otherwise false.

Following is the example showing usage of the above methods:

```
object Test {
   def main(args: Array[String]) {
      val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
```

```
      val nums = Nil

      println( "Head of fruit : " + fruit.head )
      println( "Tail of fruit : " + fruit.tail )
      println( "Check if fruit is empty : " + fruit.isEmpty )
      println( "Check if nums is empty : " + nums.isEmpty )
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
Head of fruit : apples
Tail of fruit : List(oranges, pears)
Check if fruit is empty : false
Check if nums is empty : true

C:/>
```

Concatenating Lists:

You can use either **:::** operator or **List.:::** method or **List.concat** method to add two or more lists. Following is the example:

```
object Test {
   def main(args: Array[String]) {
      val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))
      val fruit2 = "mangoes" :: ("banana" :: Nil)

      // use two or more lists with ::: operator
      var fruit = fruit1 ::: fruit2
      println( "fruit1 ::: fruit2 : " + fruit )

      // use two lists with Set.:::() method
      fruit = fruit1.:::(fruit2)
      println( "fruit1.:::(fruit2) : " + fruit )

      // pass two or more lists as arguments
      fruit = List.concat(fruit1, fruit2)
      println( "List.concat(fruit1, fruit2) : " + fruit  )


   }
}
```

When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
fruit1 ::: fruit2 : List(apples, oranges, pears, mangoes, banana)
fruit1.:::(fruit2) : List(mangoes, banana, apples, oranges, pears)
List.concat(fruit1, fruit2) : List(apples, oranges, pears, mangoes, banana)

C:/>
```

Creating Uniform Lists:

You can use **List.fill** method creates a list consisting of zero or more copies of the same element as follows:

```
object Test {
   def main(args: Array[String]) {
      val fruit = List.fill(3)("apples") // Repeats apples three times.
      println( "fruit : " + fruit  )
```

```
      val num = List.fill(10)(2)         // Repeats 2, 10 times.
      println( "num : " + num  )
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
fruit : List(apples, apples, apples)
num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

C:/>
```

Tabulating a Function:

You can use a function along with **List.tabulate** method to apply on all the elements of the list before tabulating the list. Its arguments are just like those of List.fill: the first argument list gives the dimensions of the list to create, and the second describes the elements of the list. The only difference is that instead of the elements being fixed, they are computed from a function:

```
object Test {
   def main(args: Array[String]) {
      // Creates 5 elements using the given function.
      val squares = List.tabulate(6)(n => n * n)
      println( "squares : " + squares  )

      //
      val mul = List.tabulate( 4,5 )( _ * _ )
      println( "mul : " + mul  )
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
squares : List(0, 1, 4, 9, 16, 25)
mul : List(List(0, 0, 0, 0, 0), List(0, 1, 2, 3, 4),
           List(0, 2, 4, 6, 8), List(0, 3, 6, 9, 12))

C:/>
```

Reverse List Order:

You can use **List.reverse** method to reverse all elements of the list. Following is the example to show the usage:

```
object Test {
   def main(args: Array[String]) {
      val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
      println( "Before reverse fruit : " + fruit )

      println( "After reverse fruit : " + fruit.reverse )
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
Before reverse fruit : List(apples, oranges, pears)
After reverse fruit : List(pears, oranges, apples)

C:/>
```

Scala List Methods:

Following are the important methods, which you can use while playing with Lists. For a complete list of methods available, please check official documentation of Scala.

SN	Methods with Description
1	**def +*elem*: *A*: List[A]** Prepends an element to this list
2	**def ::*x*: *A*: List[A]** Adds an element at the beginning of this list.
3	**def :::*prefix*: *List*[*A*]: List[A]** Adds the elements of a given list in front of this list.
4	**def ::*x*: *A*: List[A]** Adds an element x at the beginning of the list
5	**def addString*b*: *StringBuilder*: StringBuilder** Appends all elements of the list to a string builder.
6	**def addString*b*: *StringBuilder*, *sep*: *String*: StringBuilder** Appends all elements of the list to a string builder using a separator string.
7	**def apply*n*: *Int*: A** Selects an element by its index in the list.
8	**def contains*elem*: *Any*: Boolean** Tests whether the list contains a given value as an element.
9	**def copyToArray*xs*: *Array*[*A*], *start*: *Int*, *len*: *Int*: Unit** Copies elements of the list to an array. Fills the given array xs with at most len elements of this list, starting at position start.
10	**def distinct: List[A]** Builds a new list from the list without any duplicate elements.

11

def drop$n$**:** Int**: List[A]**

Returns all elements except first n ones.

12

def dropRight$n$**:** Int**: List[A]**

Returns all elements except last n ones.

13

def dropWhile$p$**: (**A **=> Boolean): List[A]**

Drops longest prefix of elements that satisfy a predicate.

14

def endsWith[B]$that$**:** $Seq[B]$**: Boolean**

Tests whether the list ends with the given sequence.

15

def equals$that$**:** Any**: Boolean**

The equals method for arbitrary sequences. Compares this sequence to some other object.

16

def exists$p$**: (**A **=> Boolean): Boolean**

Tests whether a predicate holds for some of the elements of the list.

17

def filter$p$**: (**A **=> Boolean): List[A]**

Returns all elements of the list which satisfy a predicate.

18

def forall$p$**: (**A **=> Boolean): Boolean**

Tests whether a predicate holds for all elements of the list.

19

def foreach$f$**: (**A **=> Unit): Unit**

Applies a function f to all elements of the list.

20

def head: A

Selects the first element of the list.

21

def indexOf$elem$**:** A**,** $from$**:** Int**: Int**

Finds index of first occurrence of some value in the list after or at some start index.

22

def init: List[A]

Returns all elements except the last.

23

def intersect*that*: *Seq*[*A*]: **List[A]**

Computes the multiset intersection between the list and another sequence.

24

def isEmpty: Boolean

Tests whether the list is empty.

25

def iterator: Iterator[A]

Creates a new iterator over all elements contained in the iterable object.

26

def last: A

Returns the last element.

27

def lastIndexOf*elem*: *A*, *end*: *Int*: **Int**

Finds index of last occurrence of some value in the list before or at a given end index.

28

def length: Int

Returns the length of the list.

29

def map[B]*f*: (*A* => **B): List[B]**

Builds a new collection by applying a function to all elements of this list.

30

def max: A

Finds the largest element.

31

def min: A

Finds the smallest element.

32

def mkString: String

Displays all elements of the list in a string.

33

def mkString*sep*: *String*: **String**

Displays all elements of the list in a string using a separator string.

34

def reverse: List[A]

Returns new list with elements in reversed order.

35

def sorted[B >: A]: List[A]

Sorts the list according to an Ordering.

36

def startsWith[B](*that*: *Seq*[*B*], *offset*: *Int*): **Boolean**

Tests whether the list contains the given sequence at a given index.

37

def sum: A

Sums up the elements of this collection.

38

def tail: List[A]

Returns all elements except the first.

39

def take(*n*: *Int*): **List[A]**

Returns first n elements.

40

def takeRight(*n*: *Int*): **List[A]**

Returns last n elements.

41

def toArray: Array[A]

Converts the list to an array.

42

def toBuffer[B >: A]: Buffer[B]

Converts the list to a mutable buffer.

43

def toMap[T, U]: Map[T, U]

Converts this list to a map.

44

def toSeq: Seq[A]

Converts the list to a sequence.

45

def toSet[B >: A]: Set[B]

Converts the list to a set.

46

def toString: String

Converts the list to a string.