Scala I

Muhammad Tahir

Lecture 10

Contents

_ wildcard

- _ in Scala is similar to * Java
- Handling the default case
- Used as place holder
- Accessing a tuple selectively
- Wildcard imports

```
import scala.collection._
import chisel3._
```

_ as Placeholder

```
// An example List
val uList = List(11, -10, 5, 0, -5, 10)

// Applying .filter to List
val uList_filter1 = uList.filter(x => x > -1)
println(s"Filtered list = $uList_filter1")

// Applying .filter to List using _
val uList_filter2 = uList.filter(_ > -1)
println(s"Filtered list using _ as place holder =
    $uList_filter2")

// The output at the terminal
Filtered list = List(11, 5, 0, 10)
Filtered list using _ as place holder = List(11, 5, 0, 10)
```

_ as Placeholder Cont'd

```
def ALU_Scala(a: Int, b: Int, op: Int): Int = {
    op match {
        case 1 => a + b
        case 2 => a - b
        case 3 => a & b
        case 4 => a | b
        case 5 => a ^ b
        case _ => -999    // This should not happen
    }
}

var result = ALU_Scala(18,11,2)
println(s"The result is: $result")

result = ALU_Scala(12,17,9)
println(s"The result is: $result")

// The output at the terminal
The result is: 7
The result is: -999
```

_ as Placeholder Cont'd

```
// User type definition
type R = Int

// An example List
val uList = List(1, 2, 3, 4, 5)

// functional composition
def compose(g: R => R, h: R => R) =
(x: R) => g(h(x))

// implement y = mx+c ( with m=2 and c =1)
def y1 = compose(x => x + 1, x => x * 2)
def y2 = compose(_ + 1, _ * 2)

val uList_map1 = uList.map(x => y1(x))
val uList_map2 = uList.map(y2(_))

println(s" Linearly mapped list 1 = $uList_map1 ")
println(s" Linearly mapped list 2 = $uList_map2 ")

// The output at the terminal
Linearly mapped list 1 = List (3, 5, 7, 9, 11)
Linearly mapped list 2 = List (3, 5, 7, 9, 11)
```

_ for Function Assignment

```
// assigning a function to val
val f_max = scala.math.max _
val f_abs = scala.math.abs _

// calling the function
val max_value = f_max(1, 5)
val abs_value = f_abs(-123)

println(s"The maximum value is = $max_value")
println(s"The absolute value is = $abs_value")

// The output at the terminal
The maximum value is = 5
The absolute value is = 123
```

Higher Order Functions

Higher-order functions in Scala:

- either take a function as an argument
- or return a function

```
// Define a higher order function
def higherOrder(a: Int, b: Int, c: Int, d: Int, function: (Int, Int) => Int) = {
      val first_inst = function(a, b)
      val second_inst = function(c, d)
      function(first_inst, second_inst)
}

// call to higherOrder function
val result1 = higherOrder(2, 5, 7, 9, (x, y) => x + y)
val result2 = higherOrder(2, 5, 7, 9, _ + _)

println(s"The result1 is = $result1")
println(s"The result2 is same = $result2")

// The output at the terminal
The result1 is = 23
The result2 is same = 23
```

Hiding a Class During Import

- Wildcard to hide a class from being imported

```
import chisel3.util.{MuxCase => _ , _}

io.out := MuxCase(false.B, Array(
    (io.sel === 0.U)  ->  io.in0,
    (io.sel === 1.U)  ->  io.in1,
    (io.sel === 2.U)  ->  io.in2,
    (io.sel === 3.U)  ->  io.in3 )
)
```

```
// We get the following error message
not found: value MuxCase
```

apply Method

1. `apply` method is widely used as one of the default methods supported by traits, classes and objects
2. `apply` method for the three traits Seq, Set and Map
   - Seq: `apply` is positional indexing, element numbering starts from 0 always
   - Set: `apply` is a membership test and is similar to the contains method
   - Map: `apply` is a selection and returns the value associated with the given key

apply Method: Example

```
// Illustration of built-in apply functions for different types of
    collections
val uList = List (11 , 22 , 33 , 44 , 55)
val uSet = Set (11 , 22 , 33 , 44 , 55)
val uMap = Map (1 -> 'a', 2 -> 'b', 3 -> 'c', 4 -> 'd')

println (s" Apply method for the List with .apply = ${uList.apply
    (1)}")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for the List without .apply = ${uList(1)}")

println (s" Apply method for the Set with .apply = ${uSet.apply(22)}
    ")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for the Set without .apply = ${uSet(22)}")

println (s" Apply method for Map with .apply = ${uMap.apply(2)}")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for Map without .apply = ${uMap(2)}")
```

zip and unzip

- The zip takes two lists and creates a list of pairs
- If the lengths of two lists are different then unmatched elements are dropped
- zip and unzip methods allow to operate on multiple lists together

zip and unzip Cont'd

```
val uList1: List[(Char)] = List('a', 'b', 'c', 'd', 'e')
val uList2: List[(Int)] = List(20, 40, 100)

val uList_Zipped = uList1.zip(uList2)
println(s"The zipped list is: $uList_Zipped")

val uList_unZipped = uList_Zipped.unzip
println(s"The unzipped list is: $uList_unZipped")

val uList_indexZip = uList1.zipWithIndex
println(s"The list zipped with its index: $uList_indexZip")

// The output at the terminal
The zipped list is: List((a,20), (b,40), (c,100))
The unzipped list is: (List(a, b, c),List(20, 40, 100))
The list zipped with its index: List((a,0), (b,1), (c,2), (d,3), (e
    ,4))
```

reduce Method

- reduce is a higher order method
- Traverses all the elements in a collection
- It combines them in *some way* to produce a the result
- Uses a binary operation to combine the elements
- Can not be applied to empty collection
- The order matters

reduce Method: Examples

```
// Illustration of reduce method for Lists
val uList = List(1, 2, 3, 4, 5)

val uSum_Explicit = uList.reduce((a, b) => a + b)
println(s"Sum of elements using reduce function explicitly=
    $uSum_Explicit")

val uSum: Double = uList.reduce(_ + _)
println(s"Sum of elements using reduce function with
    wildcard = $uSum")

// The output at the terminal is given below
Sum of elements using reduce function explicitly = 15
Sum of elements using reduce function with wildcard = 15.0
```

reduce Method: Examples Cont'd

```
// source collection
val uList = List(1, 5, 7, 8)

// converting every element to a pair of the form (x,1)
val uList_Modified = uList.map(x => (x, 1))

// adding elements at correspnding positions
val result = uList_Modified.reduce((a, b) => (a._1 + b._1, a
    ._2 + b._2))
val average = (result._1).toFloat / (result._2).toFloat

println("(sum, no_of_elements) = " + result)
println("Average = " + average)

// The result at the terminal
(sum, no_of_elements) = (21,4)
Average = 5.25
```

Order of reduce Method

```
// Illustration of the order of reduce method
val uList = List(1, 2, 3, 4, 5)

val uSum: Double = uList.reduce(_ - _)
println(s"Difference of elements using reduce function =
    $uSum")

val uSum_Explicit = uList.reduceLeft(_ - _)
println(s"Difference of elements using reduceLeft =
    $uSum_Explicit")

val uSum_Explicit1 = uList.reduceRight(_ - _)
println(s"Difference of elements using reduceRight =
    $uSum_Explicit1")

// The output at the terminal
Difference of elements using reduce function = -13.0
Difference of elements using reduceLeft function = -13
Difference of elements using reduceRight function = 3
```

Processing Multiple Lists

- Multiple lists are mapped to a Tuple
- The zipped method on tuples allows processing multiple lists together

```
val uList1: List[(Int)] = List(3, 2)
val uList2: List[(Int)] = List(20, 40, 100)

// Processing multiple lists
val resultProduct = (uList1, uList2).zipped.map(_ * _)
val resultCount = (uList1, uList2).zipped.max

println(s"The product of two lists: $resultProduct")
println(s"The max element: $resultCount")

// The output at the terminal
The product of two lists: List(60, 80)
The max element: (3,20)
```

Reading List I

- Read Chapters 16 and 25 of [Odersky et al., 2016] for in-depth understanding of Scala Collections
- The tutorial available at [Tutorial, 2020] is good resource for quick reference

References

Odersky, M., Spoon, L., and Venners, B. (2016).
Programming in Scala.
Artima Incorporation.

Tutorial (2020).
Scala tutorial.
https://www.tutorialspoint.com/scala/index.htm.