Self Study Topis:

- Scala Traits and Inheritance
- Linear flattening
- Modules as parameters in SCALA
- Implicit Function, Class and Parameters
- Lazy Val

Exercise 1: Write Scala program that implements the function $y = ax^2 + bx + c$, where $a = 3,\ b = 5$ and $c = 7$ (this is similar to what has been done in Listing 10.4). Create a list of integers in the range $-3 \leq x \leq 3$ and use the defined function to map its elements to another list. Print the mapped list and verify the results. (Try to use wildcard wherever possible.)

Exercise 2: Zip the two lists created in Exercise 1 and then zip the resulting list with its index. A list with three elements per pair is created in the following format: $(x, f(x), index)$. Find the mean value of $f(x)$ and store the respective pair to a variable *mean*. Refer to Listings 10.11, 10.16 and 10.21 for possible hints.

Exercise 3: Write a Scala code that takes in a vector as an integer list and calculates its Euclidean norm. Recall that this is also termed as the magnitude of a vector and is evaluated as given below.

$$||\vec{u}||_2 = \left(\sum_{i=1}^{N} |u_i|^2\right)^{\frac{1}{2}} = \sqrt{u_1^2 + u_2^2 + ... + u_N^2}$$

Exercise 4: Refer to the Listings 11.2, 11.3, 11.4, 11.5. Redo all these examples using `wildcard`.

Exercise 1: Design an FSM by using a companion object. Put static functionality in companion object i.e. `enum` will be placed inside the `object` instead of `class` then use this to develop your FSM code inside the class.

Exercise 2: Create a shallow and a deep copy of an object of a case class given in the Listing 11.13. Find out whether the copy method available with case class gives a shallow or a deep copy.[1]

Exercise 3: Refer to the Listing 11.2,11.3,11.4,11.5. Find out which other collections support `map` and `flatMap` methods and go through their illustrations.

Problem 4: Refer to Exp 9, Problem 5. Solve the same problem but using `map` and `flatMap`.

Exercise 1: Write an apply function that when called add a complete list that its called upon and prints out its value.

Exercise 2: Refer to Listing 13.4 and modify it to pass two modules as parameters at the same time and show their output at two different output ports.

Exercise 1: Implicit can be used in different context. In this session we only saw how implicit is used for parameters. Explore and learn how implicit is used for type conversion.

Exercise 2: Write a function which has two inputs of any type and then implicitly converts these input of any type to string and print the addition of these strings.

Appendix

(Listings Attached)

```
// 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 ")
```

Listing 10.2

```
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))
```

Listing 10.11: Zip and unzip methods.

```
// 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)
```

Listing 10.16: Reduce method on paired list.

```
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")
```

Listing 10.21: Processing multiple lists.

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

// map method applied to List
val uList_Twice = uList.map( x -> x*2 )
println(s"List elements doubled - $uList_Twice")

// Applying map to List using user defined method
def f(x: Int) = if (x > 2) x*x else None
val uList_Squared = uList.map(x -> f(x))
println(s"List elements squared selectively - $uList_Squared")

// The output at the terminal is given below
List elements doubled = List(2, 4, 6, 8, 10)
List elements squared selectively = List(None, None, 9, 16, 25)
```

Listing 11.2: Illustration of `map` method.

```
// An example list
val uList: List[Int] = List(1, 2, 3, 4, 5)

def g(v:Int) = List(v-1, v, v+1)
val uList_Extended = uList.map(x -> g(x))
println(s"Extended list using map - $uList_Extended")

val uList_Extended_flatmap = uList.flatMap(x -> g(x))
println(s"Extended list using flatMap - $uList_Extended_flatmap")

// The output at the terminal is
Extended list using map = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4),
    List(3, 4, 5), List(4, 5, 6))

Extended list using flatMap = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5,
     6)
```

Listing 11.3: Illustration of `map` and `flatMap` methods.

```scala
import chisel3._
import chisel3.util._
import chisel3.experimental.{BaseModule}

// Define IO interface as a Trait
trait ModuleIO {
    def in1: UInt
    def in2: UInt
    def out: UInt
}

class Add extends RawModule with ModuleIO {
    val in1 = IO(Input(UInt(8.W)))
    val in2 = IO(Input(UInt(8.W)))
    val out = IO(Output(UInt(8.W)))
    out := in1 + in2
}

class Sub extends RawModule with ModuleIO {
    val in1 = IO(Input(UInt(8.W)))
    val in2 = IO(Input(UInt(8.W)))
    val out = IO(Output(UInt(8.W)))
    out := in1 - in2
}

class Top [T <: BaseModule with ModuleIO] (genT: => T) extends Module {
    val io = IO(new Bundle {
        val in1 = Input(UInt(8.W))
        val in2 = Input(UInt(8.W))
        val out = Output(UInt(8.W))
    })
    val sub_Module = Module(genT)
    io.out := sub_Module.out
    sub_Module.in1 := io.in1
    sub_Module.in2 := io.in2
}

// Generate verilog for two modules, one for addition, second for subtraction
println((new chisel3.stage.ChiselStage).emitVerilog(new Top(new Add)))
println((new chisel3.stage.ChiselStage).emitVerilog(new Top(new Sub)))
```


Listing 13.4: Module as parameter example code.

```
// An example list
val uList: List[Int] = List(1, 2, 3, 4, 5)

// Applying map and flatMap to List with builtin Options class
def f(x: Int) = if (x > 2) Some(x) else None
val uList_selective = uList.map(x => f(x))
println(s"Selective elements of List with .map = $uList_selective")

val uList_selective_flatMap = uList.flatMap(x => f(x))
println(s"Selective elements of List with .flatMap =
    $uList_selective_flatMap")

// Output at the terminal
Selective elements of List using .map = List(None, None, Some(3), Some(4),
    Some(5))
Selective elements of List using .flatMap = List(3, 4, 5)
```

Listing 11.4: Illustration of map and flatMap method.

```
// An example Map using (key, value) pairs
val uMap = Map('a' -> 2, 'b' -> 4, 'c' -> 6)

// Applying .mapValues to Map
val uMap_mapValues = uMap.mapValues(v => v*2)
println(s"Map values doubled using .mapValues = $uMap_mapValues")

def h(k:Int, v:Int) =  Some(k->v*2)

// Applying .map to Map
val uMap_map = uMap.map {
    case (k,v) => h(k,v)
}
println(s"Map values doubled using .map = $uMap_map")

// Applying .flatMap to Map
val uMap_flatMap = uMap.flatMap {
    case (k,v) => h(k,v)
}
println(s"Map values doubled using .flatMap = $uMap_flatMap")

// The output at the terminal
Map values doubled using .mapValues = Map(a -> 4, b -> 8, c -> 12)
Map values doubled using .map = List(Some((97,4)), Some((98,8)), Some
    ((99,12)))
Map values doubled using .flatMap = Map(97 -> 4, 98 -> 8, 99 -> 12)
```

Listing 11.5: Illustration of map and flatMap methods applied to Map collection.