HIGHER ORDER FUNCTIONS

Higher Order Functions

- Higher order functions take other functions as parameters or return a function as a result.
- This is possible because functions are first-class values in Scala.
- The terminology can get a bit confusing at this point, and we use the phrase "higher order function" for both methods and functions that take
  - functions as parameters or
  - that return a function.
- In a pure Object Oriented world a good practice is to avoid exposing methods parameterized with functions that might leak object's internal state.
- Leaking internal state might break the invariants of the object itself thus violating encapsulation.
- One of the most common examples is the higher-order function map which is available for collections in Scala.

Function As Parameter

- In Scala, you can define functions that take other functions as parameters.
- These functions are referred to as higher-order functions.
- By accepting functions as arguments, higher-order functions can encapsulate generic behavior that can be customized by providing different functions.
- This allows for code reuse and modularity.

Function As Parameter

```
object MainObject{
   def main(args: Array[String])={
      def math(a:Int,b:Int,arithop:(Int,Int)=>Int):Int=arithop(a,b)
                  val result=math(3,5,(a,b) => a+b)
                  println(result)
  def add(a:Int,b:Int):Int =a+b
   println(math(5,6,add))
 }
 }
```

Function As Return Value

- When a function returns another function as its result, it is referred to as a function returning a function or a higher-order function.
- This allows for the creation of functions that generate or customize other functions based on certain conditions or parameters.

Function As Return Value

```
object MainObject{
   def main(args: Array[String])={
         def math(opname:String):(Int,Int)=>Int=(x:Int,y:Int)=>{
              opname match{
                     case "add"=>x+y
                     case "sub"=>x-y
                     case "mul"=>x*y
                   }
}
val addValue=math(opname="add")
println(addValue(3,2))
val subValue=math(opname="sub")
println(subValue(3,2))
val mulValue=math(opname="mul")
println(mulValue(3,2))
}
```

Example

```
object MainObject {
   def main(args: Array[String]) = {
     functionExample(25, multiplyBy2)           // Passing a function as parameter

    }
    def functionExample(a:Int, f:Int=>AnyVal):Unit = {
        println(f(a))                                    // Calling that function
    }
    def multiplyBy2(a:Int):Int = {
        a*2
    }
}
```

Higher Order Functions In Scala

- map()
- flatmap()
- filter()
- reduce()
- fold()

map()

- Map: The map function in Scala is used to transform each element in a collection and create a new collection with the transformed elements.
- It applies a given function to each element and returns the results in a new collection of the same size.
- Example

```
val no=List(1,2,3,4,5)
val nomap=no.map(x=>x+2)
println(nomap)
```

flatmap()

- The flatMap function is similar to map, but it allows each input element to be mapped to zero or more output elements.
- It applies a function to each element and flattens the results into a single collection.
- The flatMap function is used to convert a 2-D array to a 1-D array.
- Example

```
val sentence=List("hello good morning")
val sentenceflatmap=sentence.flatMap(sen=>sen.toCharArray)
println(sentenceflatmap)
```

filter()

- The filter function accepts a function that returns a Boolean value (also known as a predicate) as input, applies the function to every element in a collection of values, and returns the elements that return true from the function as a new collection.
- The predicate should accept a parameter of the same type as the elements in the collection, evaluate the result with the element, and return true to keep the element in the new collection or false to filter it out.
- Example:

```
val no=List(1,2,3,4,5)
val nofilter=no.filter(x=>x%2==0)
println(nofilter)
```

reduce()

- The reduce function accepts a combining function (usually a binary operation) as input, applies the function to successive elements in a collection of values, and returns a single cumulative result.
- It applies the operation in a pairwise manner, starting from the first element.
- The result is then used with the next element, and so on until all elements are combined into a single value.

Example:

```
Val no=List(1,2,3,4,5)
val noreduce=no.reduce((x,y)=>x+y)
println(noreduce)
```

fold()

- The fold function is a more general form of reduction operation that combines the elements of a collection using an initial value and a binary operation.
- It starts with an initial value and applies the operation to the initial value and the first element, then the result and the next element, and so on.
- Example:

```
Val no=List(1,2,3,4,5)
val nofold=no.fold(2)((x,y)=>x+y)
println(nofold )
```

span()

- The "span" operation splits a collection into two parts based on a predicate function.
- It returns a pair of collections: the first part contains the elements that satisfy the predicate until the first element that does not satisfy it, and the second part contains the remaining elements.
  - val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  - val (even, odd) = numbers.span(_ % 2 == 0)
  - println(even)
  - println(odd)

  Output:
  - List()
  - List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

partition()

- The "partition" operation splits a collection into two parts based on a predicate function, similar to "span".
- However, unlike "span", it doesn't stop at the first element that doesn't satisfy the predicate.
- Instead, it partitions all elements into two collections based on whether they satisfy the predicate or not.
  - val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  - val (even, odd) = numbers.partition(_ % 2 == 0)
  - println(even) // Output: List(2, 4, 6, 8, 10)
  - println(odd) // Output: List(1, 3, 5, 7, 9)

Example Partition and span

```
object HelloWorld {    def main(args: Array[String]): Unit =
{val numbers = List(6, 1, 3, 4, 5, 6, 7, 8, 9, 10)
val (evenpart, oddpart) = numbers.partition(x=>x % 2 == 0)
val(evenspan,oddspan)=numbers.span(x=>x%2==0)
println("partition even"+evenpart)
 println("partition odd"+oddpart)
println("span even"+evenspan)
println("span odd"+oddspan)
}
}
```

- Output:

- partition evenList(6, 4, 6, 8, 10)
- partition oddList(1, 3, 5, 7, 9)
- span evenList(6)
- span oddList(1, 3, 4, 5, 6, 7, 8, 9, 10)

```
object MainObject{
  def main(args: Array[String])={
    val no=List(1,2,3,4,5)
    val nomap=no.map(x=>x+2)
    println(nomap)
    val sentence=List("hello good morning")
    val sentenceflatmap=sentence.flatMap(sen=>sen.toCharArray)
    println(sentenceflatmap)
    val nofilter=no.filter(x=>x%2==0)
    println(nofilter)
    val noreduce=no.reduce((x,y)=>x+y)
    println(noreduce)
    val nofold=no.fold(2)((x,y)=>x+y)
    println(nofold)
 }
}
```

- Output:

- List(3, 4, 5, 6, 7))
- List(h, e, l, l, o, , g, o, o, d, , m, o, r, n, i, n, g))
- List(2, 4))
- 15
- 17