ADVANCED FUNCTIONAL THINKING

DR SURBHI SARASWAT
SCHOOL OF COMPUTER SCIENCE

LAZY EVALUATION

- Lazy evaluation is an evaluation strategy which holds the evaluation of an expression until its value is needed. It avoids repeated evaluation.
- Lazy Evaluation – Advantages
  - It allows the language runtime to discard sub-expressions that are not directly linked to the final result of the expression.
  - It reduces the time complexity of an algorithm by discarding the temporary computations and conditionals.
  - It allows the programmer to access components of data structures out-of-order after initializing them, as long as they are free from any circular dependencies.
- One drawback of lazy evaluation is that the performance may not be predictable — because you cannot say exactly when the value will be evaluated.

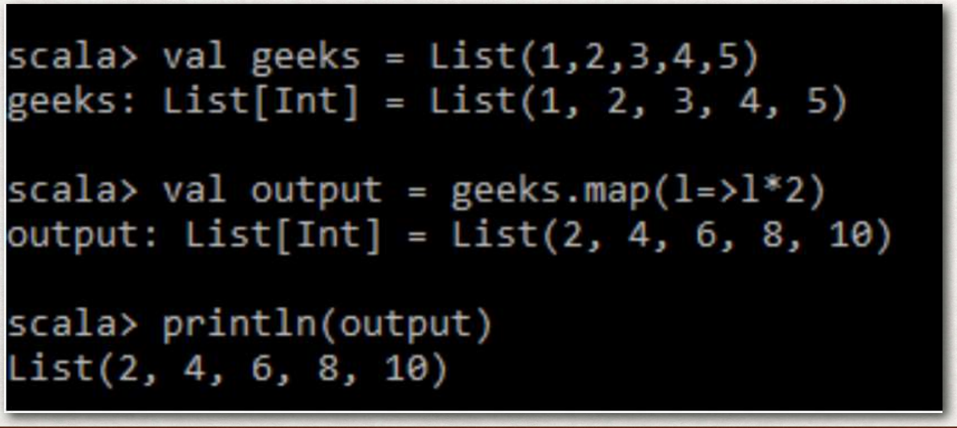
LAZY EVALUATION

- Laziness is mostly used to create data structures to handle large data volumes efficiently. Apache Spark libraries are the most common examples. If you know Spark RDD, they implement all transformations as lazy operations.

- *Strict languages* are those that evaluate the expression as soon as it's declared

- Scala uses strict evaluation by default but allows lazy evaluation of value definitions with the `lazy val`

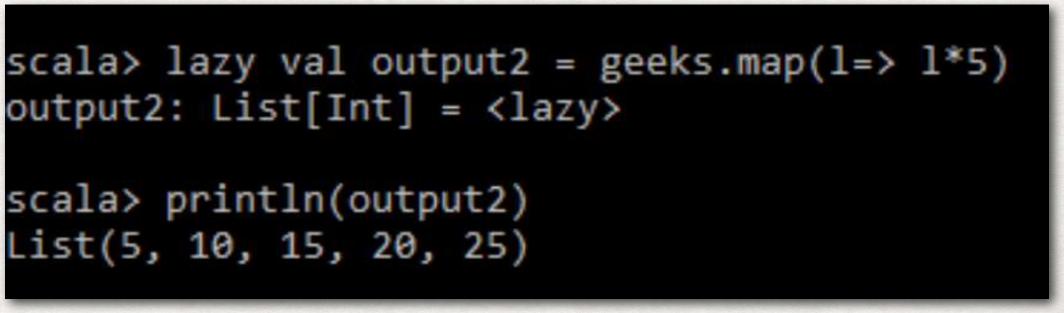
```
lazy val x = expr
```



```
scala> val geeks = List(1,2,3,4,5)
geeks: List[Int] = List(1, 2, 3, 4, 5)

scala> val output = geeks.map(l=>l*2)
output: List[Int] = List(2, 4, 6, 8, 10)

scala> println(output)
List(2, 4, 6, 8, 10)
```



```
scala> lazy val output2 = geeks.map(l=> l*5)
output2: List[Int] = <lazy>

scala> println(output2)
List(5, 10, 15, 20, 25)
```

LAZY EVALUATION

- Strict evaluation

```
def factorial(i: Int): Int = {
 println(s"Starting Factorial for $i")
 def tFactorial(n: Int, f: Int): Int = {
   if (n <= 0) f
   else tFactorial(n - 1, n * f)
 }
 return tFactorial(i, 1)
}
val s = factorial(15) / factorial(11)
/*Output:-
   Starting Factorial for 15
   Starting Factorial for 11
   s: Int = 50
 */
println(s)
//50
```

- Lazy variable assignment

```
lazy val l = factorial(15) / factorial(11)
//Output:- l: Int = <lazy>
//now use the value l
println(l)
/* Output:-
   Starting Factorial for 15
   Starting Factorial for 11
   50
```

LAZY EVALUATION

- Lazy variable assignment

```
def factorial(i: Int): Int = {
  println(s"Starting Factorial for $i")
  def tFactorial(n: Int, f: Int): Int = {
    if (n <= 0) f
    else tFactorial(n - 1, n * f)
  }
  return tFactorial(i, 1)
}
lazy val l = factorial(15) / factorial(11)
//Output:- l: Int = <lazy>
//now use the value l
println(l)
/* Output:-
    Starting Factorial for 15
    Starting Factorial for 11
    50
```

We used the same expression earlier without the keyword lazy.

- However, this time, Scala does not evaluate the value of '*l*' because we made it lazy by using the lazy keyword.
- Scala will not evaluate the expression until we use the *l*.
- So, when we print the *l*, it evaluates the expression.
- So, what is a lazy evaluation?
- Evaluate the values at the time of its first use.

LAZY EVALUATION

- Strict function values in Higher Order functions

```
//define a HO function
def twice(f: => Int) = {
  println("We have not used f yet")
  f + f
}
//call the function
twice(factorial(15) / factorial(11))
/* Output:-
   We have not used f yet
   Starting Factorial for 15
   Starting Factorial for 11
   Starting Factorial for 15
   Starting Factorial for 11
   res8: Int = 100
 */
```

- In case of a function parameter, Scala defers the evaluation for later, and it evaluates the function on every use.
- If we use f ten times in our function, Scala will evaluate it ten times.
- As the parameter function is pure and it always returns the same value, this repeated execution is unnecessary.
- How can we avoid this thing?
- Cache the parameters in a *val*.

LAZY EVALUATION

- Strict function values in Higher Order functions

```
//define a HO function
def twice(f: => Int) = {
  println("We have not used f yet")
  f + f
}
//call the function
twice(factorial(15) / factorial(11))
/* Output:-
   We have not used f yet
   Starting Factorial for 15
   Starting Factorial for 11
   Starting Factorial for 15
   Starting Factorial for 11
   res8: Int = 100
 */
```

```
//define a function
def twice(f: => Int) = {
  val i = f
  println("We have not used i yet")
  i + i
}
//Call the function
twice(factorial(15) / factorial(11))
/*Output: -
   Starting Factorial for 15
   Starting Factorial for 11
   We have not used i yet
   res9: Int = 100
 */
```

LAZY EVALUATION

- Lazy function values in Higher Order functions

```
//define the function
def twice(f: => Int) = {
 lazy val i = f
 println("We did not use i yet")
 i + i
}
//call the function
twice(factorial(15) / factorial(11))
/* Output:-
   We did not use i yet
   Starting Factorial for 15
   Starting Factorial for 11
   res1: Int = 100
 */
```

- In the previous example, Scala evaluates the function immediately as we cache it.
- In that scenario, even if we do not use the cached value, the function gets evaluated at the time of caching.
- The following example uses a lazy val to cache the function value.
- This simple technique makes the function value lazy, and instead of immediate evaluation, the function value is evaluated once at the time of its first use.