Lambda Expression in Scala

Last Updated : 28 Feb, 2019

Lambda Expression refers to an expression that uses an anonymous function instead of variable or value. Lambda expressions are more convenient when we have a simple function to be used in one place. These expressions are faster and more expressive than defining a whole function. We can make our lambda expressions reusable for any kind of transformations. It can iterate over a collection of objects and perform some kind of transformation to them.

Syntax:

val lambda_exp = (variable:Type) => Transformation_Expression

Example:

// lambda expression to find double of x

val ex = (x:Int) => x + x

Working With Lambda Expressions

We can pass values to a lambda just like a normal function call.

Example :

// Scala program to show

// working of lambda expression

// Creating object

object GfG

{

// Main method

def main(args:Array[String])

{

  // lambda expression

```
    val ex1 = (x:Int) => x + 2


    // with multiple parameters
    val ex2 = (x:Int, y:Int) => x * y


    println(ex1(7))
    println(ex2(2, 3))
}
}
```

Output:

9

6

To apply transformation to any collection, we generally use map() function. It is a higher-order function where we can pass our lambda as a parameter in order to transform every element of the collection according to the definition of our lambda expression.

Example :

```
// Scala program to apply
// transformation on collection


// Creating object
object GfG
{
// Main method
def main(args:Array[String])
{
    // list of numbers
```

```
    val l = List(1, 1, 2, 3, 5, 8)


    // squaring each element of the list
    val res = l.map( (x:Int) => x * x )


/* OR
val res = l.map( x=> x * x )
*/
    println(res)
}
}
```

Output:

```
List(1, 1, 4, 9, 25, 64)
```

We can see that the defined anonymous function to perform the square operation is not reusable.

We are passing it as an argument. However, we can make it reusable and may use it with different collections.

Example :

```
// Scala program to apply
// transformation on collection


// Creating object
object GfG
{
    // Main method
```

```
    def main(args:Array[String])
    {
        // list of numbers
        val l1 = List(1, 1, 2, 3, 5, 8)
        val l2 = List(13, 21, 34)

        // reusable lambda
        val func = (x:Int) => x * x

        // squaring each element of the lists
        val res1 = l1.map( func )
        val res2 = l2.map( func )

        println(res1)
        println(res2)
    }
}
```

Output:

```
List(1, 1, 4, 9, 25, 64)
List(169, 441, 1156)
```

A lambda can also be used as a parameter to a function.

Example :

```
// Scala program to pass lambda
// as parameter to a function
```

```
// Creating object
object GfG
{

   // transform function with integer x and
   // function f as parameter
   // f accepts Int and returns Double
   def transform( x:Int, f:Int => Double)
   =
   f(x)

   // Main method
   def main(args:Array[String])
   {

      // lambda is passed to f:Int => Double
      val res = transform(2, r => 3.14 * r * r)

      println(res)
}
}
```

Output:

12.56

In above example, transform function accepts integer x and function f, applies the transformation to x defined by f. Lambda passed as the parameter in function call returns

Double type. Therefore, parameter f must obey the lambda definition.

We can perform the same task on any collection as well. In case of collections, the only change we need to make in transform function is using map function to apply transformation defined by f to every element of the list l.

Example :

```
// Scala program to pass lambda
// as parameter to a function

// Creating object
object GfG
{

  // transform function with integer list l and
  // function f as parameter
  // f accepts Int and returns Double
  def transform( l:List[Int], f:Int => Double)
  =
  l.map(f)

  // Main method
  def main(args:Array[String])
  {
    // lambda is passed to f:Int => Double
    val res = transform(List(1, 2, 3), r => 3.14 * r * r)
    println(res)
```

```
    }
}
```

Output:

```
List(3.14, 12.56, 28.259999999999998)
```