Function Composition

Principles of Programming Languages

map() and Lambda Expressions

- 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.

```
val lambda_exp = (variable:Type) => Transformation_Expression
```

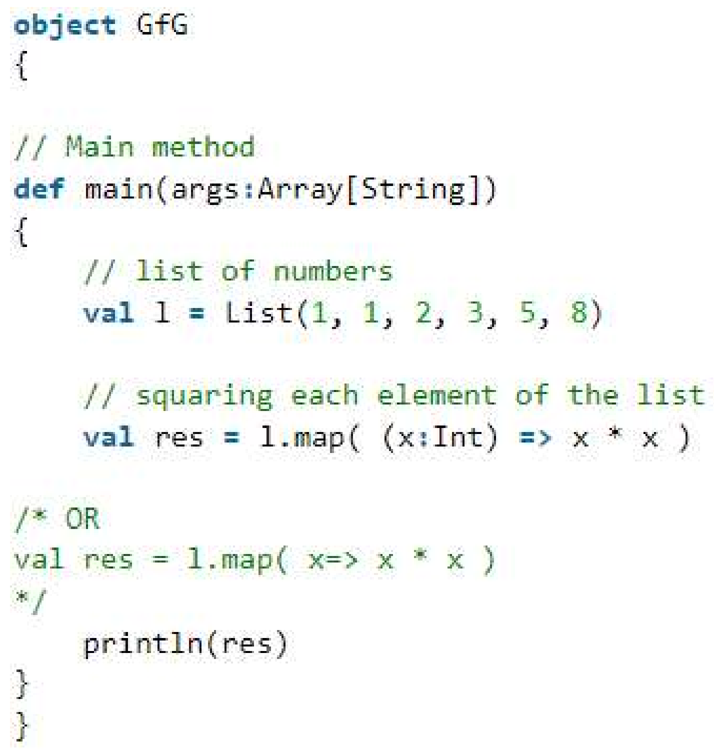
- Example:

```
// lambda expression to find double of x
val ex = (x:Int) => x + x
```

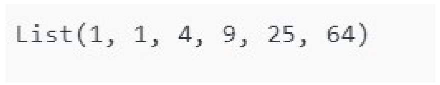
map() and Lambda Expressions

- 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

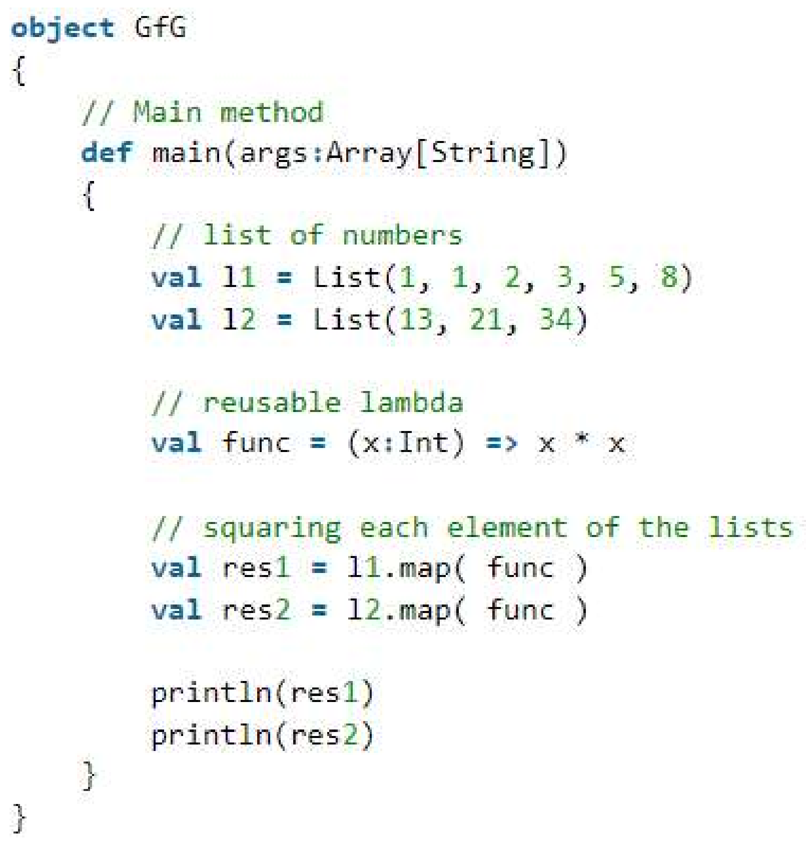


```
object GfG
{

// 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)
}
}
```



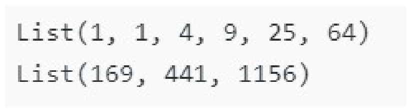
```
List(1, 1, 4, 9, 25, 64)
```

Example



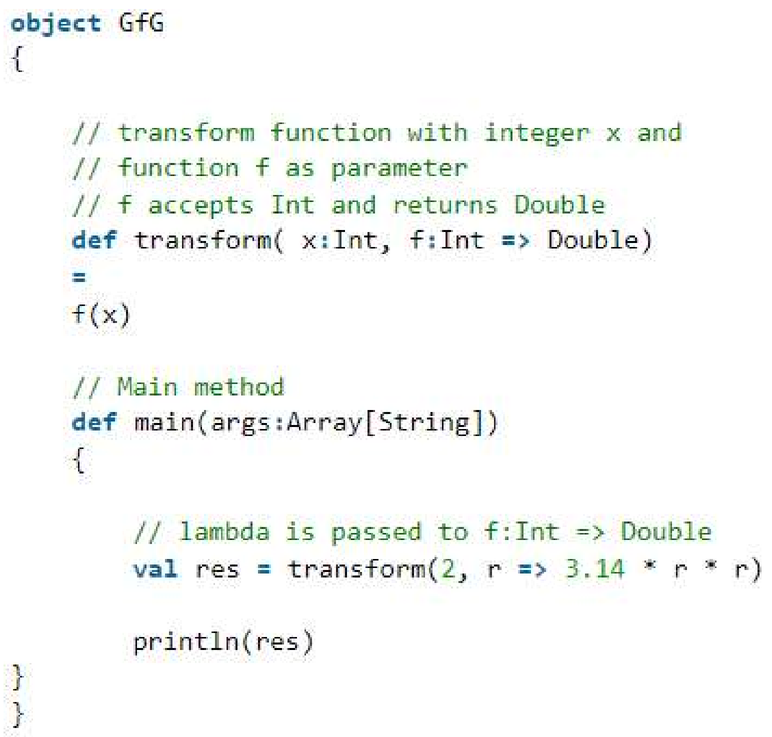
```scala
object GfG
{
    // Main method
    def main(args:Array[String])
    {
        // list of numbers
        val l1 = List(1, 1, 2, 3, 5, 8)
        val l2 = List(13, 21, 34)

        // 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)
    }
}
```

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



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

Example



```
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)
}
}
```

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

Here, 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.

```
12.56
```

Example

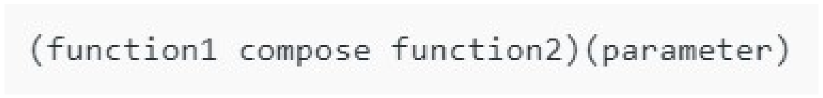
```
object GfG
{

    // 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)
    }
}
```

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.

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

Function composition

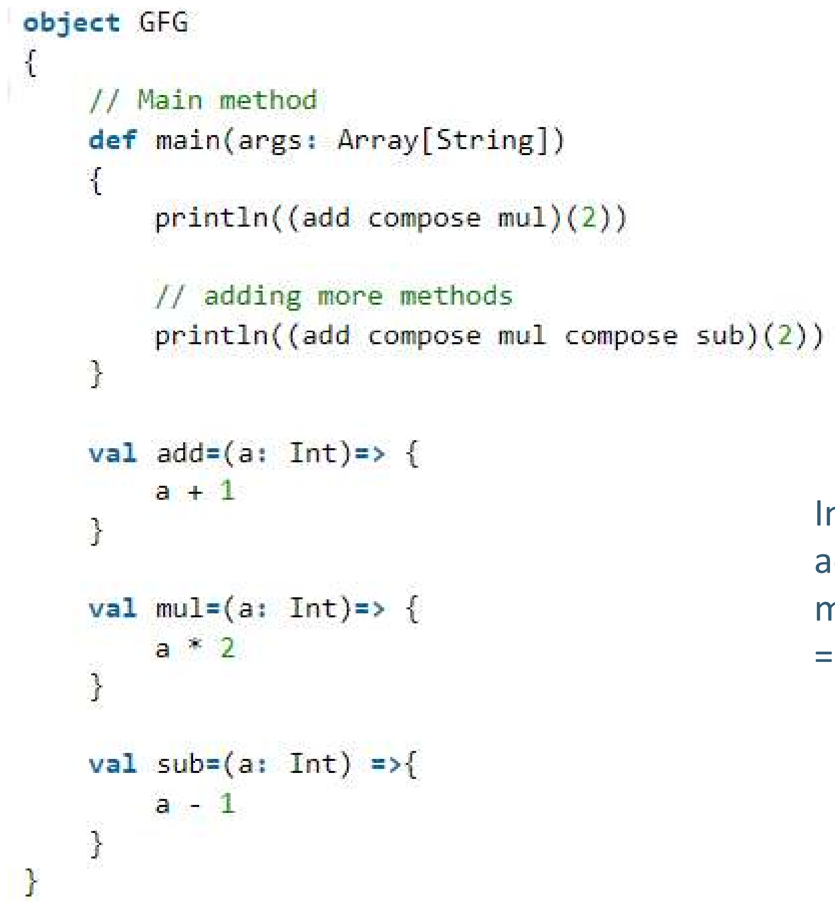
- Function composition is a way in which a function is mixed with other functions. During the composition, one function holds the reference to another function in order to fulfill it’s mission.
- compose : Composing method works with val functions



```
(function1 compose function2)(parameter)
```

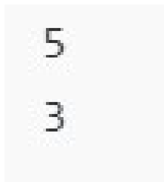
Here, function2 works first with the parameter passed & then passes then returns a value to be passed to function1.

Example



```
object GFG
{
    // Main method
    def main(args: Array[String])
    {
        println((add compose mul)(2))

        // adding more methods
        println((add compose mul compose sub)(2))
    }

    val add=(a: Int)=> {
        a + 1
    }

    val mul=(a: Int)=> {
        a * 2
    }

    val sub=(a: Int) =>{
        a - 1
    }
}
```

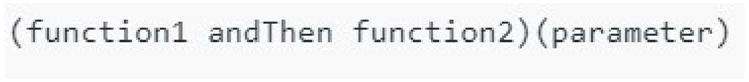
In this example, firstly mul function called we got 4(2 * 2) then add function called and we got 5(4 + 1). similarly (add compose mul compose sub)(2) will print 3 (step1 : 2 – 1 = 1, step2 : 1 * 2 = 2, step3 : 2 + 1 = 3).



```
5
3
```

Function Composition

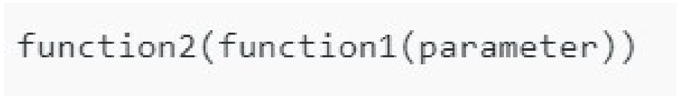
- andThen : andThen method also works with val functions.



```
(function1 andThen function2)(parameter)
```

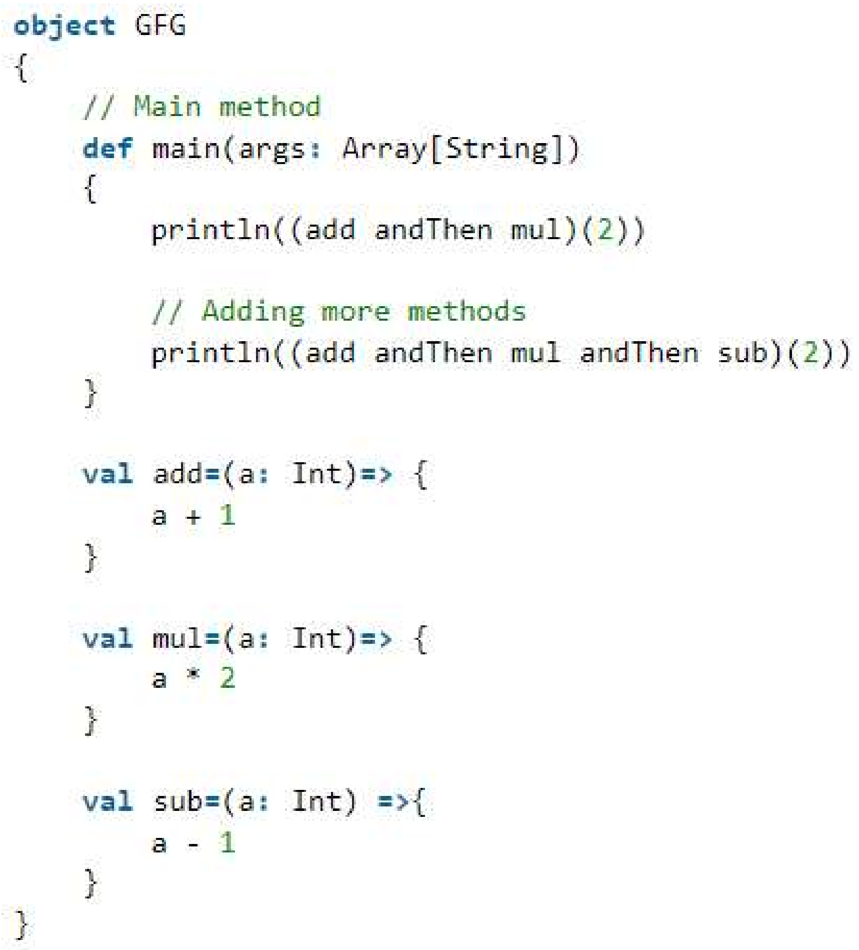
Here, function1 works first with the parameter passed & then passes then returns a value to be passed to function2.

- Also, similar to



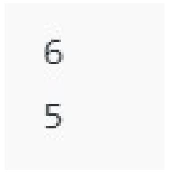
```
function2(function1(parameter))
```

Example



```
object GFG
{
    // Main method
    def main(args: Array[String])
    {
        println((add andThen mul)(2))

        // Adding more methods
        println((add andThen mul andThen sub)(2))
    }

    val add=(a: Int)=> {
        a + 1
    }

    val mul=(a: Int)=> {
        a * 2
    }

    val sub=(a: Int) =>{
        a - 1
    }
}
```

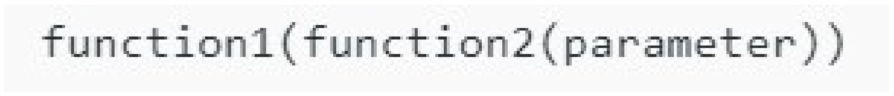
Here, firstly add function called we got 3(2 + 1) than mul function called and we got 6(3 * 2). similarly add (andThen mul andThen sub)(2)) will print 5 (step1 : 2 + 1 = 3, step2 : 3 * 2 = 6, step3 : 6 – 1 = 5).



```
6
5
```

Function Composition

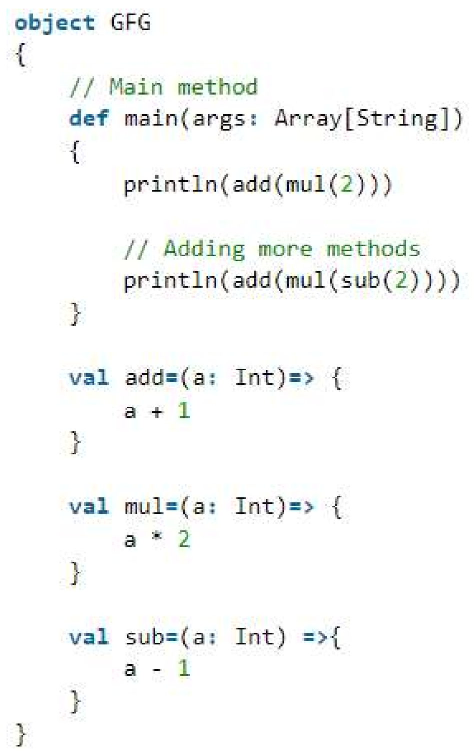
- Function composition is more like composing methods into another or passing to other methods.
- Passing methods to methods : Methods are passed to other methods.



```
function1(function2(parameter))
```

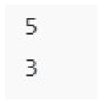
It works as same as compose function, but it works with def and val methods.

Example



```
object GFG
{
    // Main method
    def main(args: Array[String])
    {
        println(add(mul(2)))

        // Adding more methods
        println(add(mul(sub(2))))
    }

    val add=(a: Int)=> {
        a + 1
    }

    val mul=(a: Int)=> {
        a * 2
    }

    val sub=(a: Int) =>{
        a - 1
    }
}
```

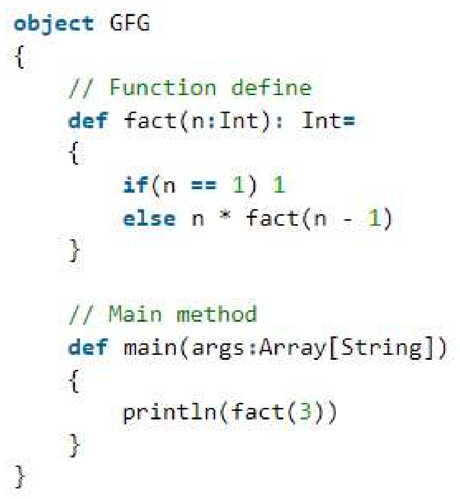
In this example, firstly mul function called we got 4(2 * 2) than add function called and we got 5(4 + 1). similarly add(mul(sub(2)) will print 3 (step1 : 2 – 1 = 1, step2 : 1 * 2 = 2, step3 : 2 + 1 = 3).



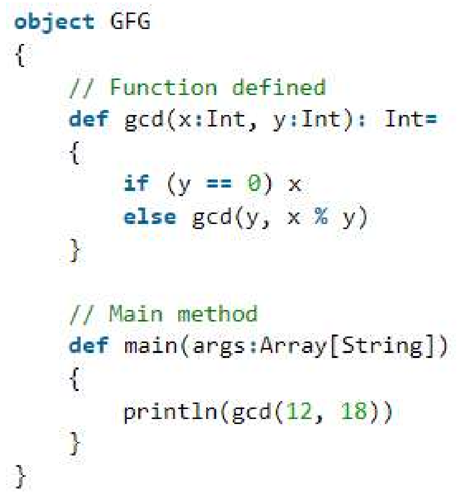
```
5
3
```

Recursion in Scala

- Recursion is a method which breaks the problem into smaller sub problems and calls itself for each of the problems. That is, it simply means function calling itself.



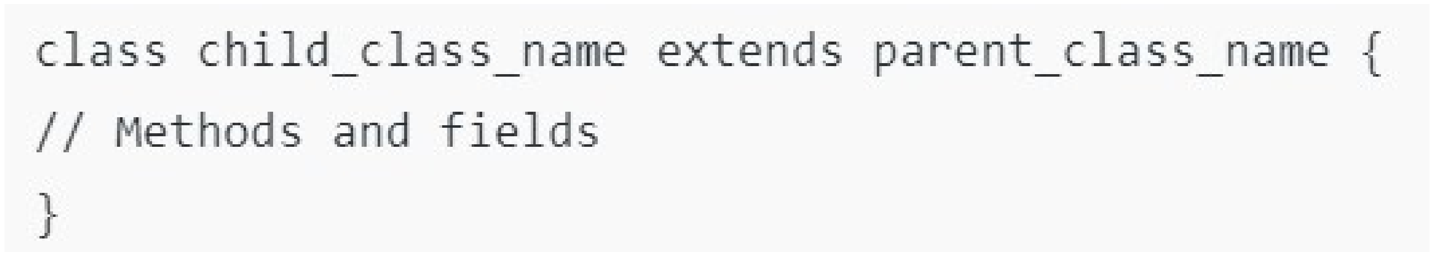
```scala
object GFG
{
    // Function define
    def fact(n:Int): Int=
    {
        if(n == 1) 1
        else n * fact(n - 1)
    }

    // Main method
    def main(args:Array[String])
    {
        println(fact(3))
    }
}
```



```scala
object GFG
{
    // Function defined
    def gcd(x:Int, y:Int): Int=
    {
        if (y == 0) x
        else gcd(y, x % y)
    }

    // Main method
    def main(args:Array[String])
    {
        println(gcd(12, 18))
    }
}
```

Inheritance

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Scala by which one class is allowed to inherit the features (fields and methods) of another class.



```
class child_class_name extends parent_class_name {
// Methods and fields
}
```

- Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).
- Sub Class: The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Scala Inheritance Types

- Single-Level
- Multi-level
- Hierarchical
- Multiple
- Hybrid