Tail Recursion

In this lesson, you will be introduced to Scala's preferred form of recursion: tail recursion.

We'll cover the following

We already know that in recursion, a function recursively calls itself. A **recursive call** becomes a **recursive tail call** when the recursive function calls itself at the very end of the function body, i.e., the tail of the function.

Tail Recursive Functions

Tail recursive functions are iterative processes and are the functional form of a loop. They are an optimized version of recursive functions, as they allow the function's stack frame to be reused.

> **Stacks** are data structures which work like a stack of boxes. Each time a function calls itself, it adds another box or **stack frame** to the stack. When we add too many boxes to our stack, they fall down. In the same way, when a function makes a large number of recursive calls, the stack *overflows*.

If the last action of a function consists of calling a function (like a tail recursive function), one stack frame would be sufficient for both functions. Using a lesser number of stacks or a lesser number of stack frames will prevent a stack from overflowing.

A Comparison

Let's look at the evaluation of both a recursive function and a tail recursive function to better understand how they differ function to better understand how they differ.

A Simple Recursive Call

The factorial function we defined in a previous lesson uses a simple recursive call.

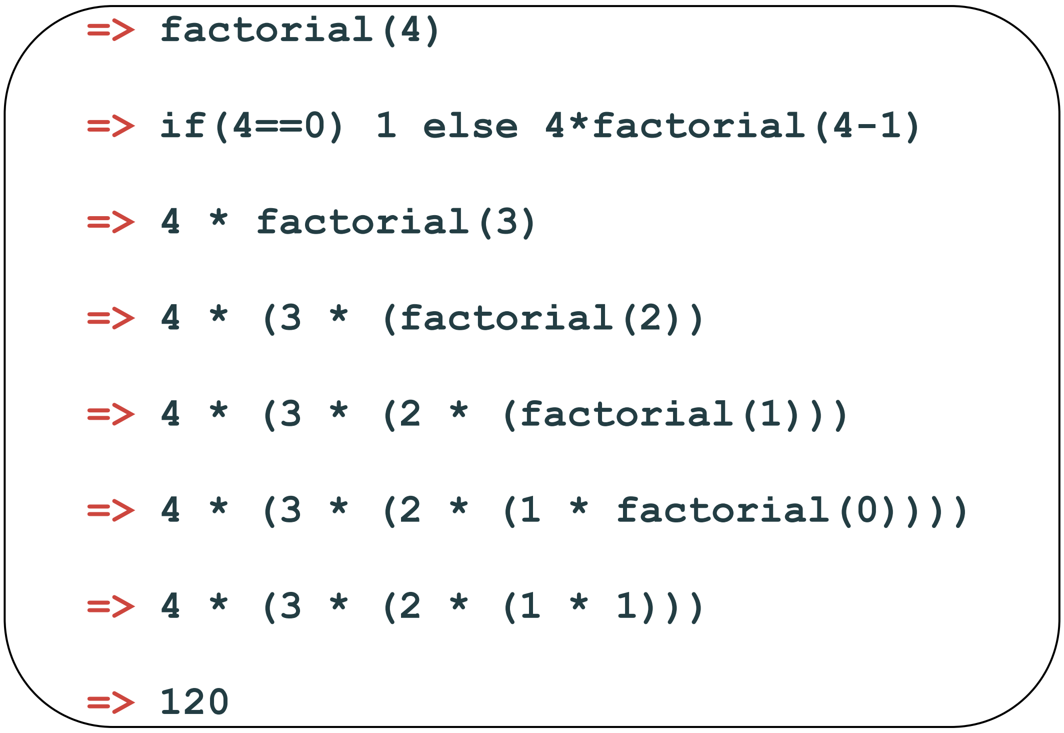
This code requires the following environment variables to execute:

LANG	C.UTF-8

```
def factorial(x: Int): Int = {
  if(x==0) 1
  else x * factorial(x-1)
}

println(factorial(4))
```

Let's see how `factorial(4)` is evaluated:



```
=> factorial(4)

=> if(4==0) 1 else 4*factorial(4-1)

=> 4 * factorial(3)

=> 4 * (3 * (factorial(2))

=> 4 * (3 * (2 * (factorial(1)))

=> 4 * (3 * (2 * (1 * factorial(0))))

=> 4 * (3 * (2 * (1 * 1)))

=> 120
```

We can see from the illustration above, that after the recursive call, the function must perform other operations of the result before termination.

A Recursive Tail Call

Let's define a function for computing the greatest common factor (gcd) of two numbers.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
def gcd(x: Int, y: Int): Int = {
  if(y==0) x
  else gcd(y, x%y)
}
 println(gcd(14,21))
```

Evaluation of `gcd(14,21)`:

```
=> gcd(14,21)

=> if(21==0) 14 else gcd(21,14%21)

=> if(false) 14 else gcd(21,14%21)

=> gcd(21,14%21)

=> gcd(21,14)

=> if(14==0) 21 else gcd(14,21%14)

=> if(false) 21 else gcd(14,21%14)

=> gcd(14,21%14)

=> gcd(14,7)

=> if(7==0) 14 else gcd(7,14%7)

=> if(false) 14 else gcd(7,14%7)

=> gcd(7,14%7)

=> gcd(7,0)
...
=> 0
```

In this illustration, we can see that the recursive call is the last thing the function needs to do, hence the function is tail recursive.

In the next lesson, you will solve an exercise on tail recursion.