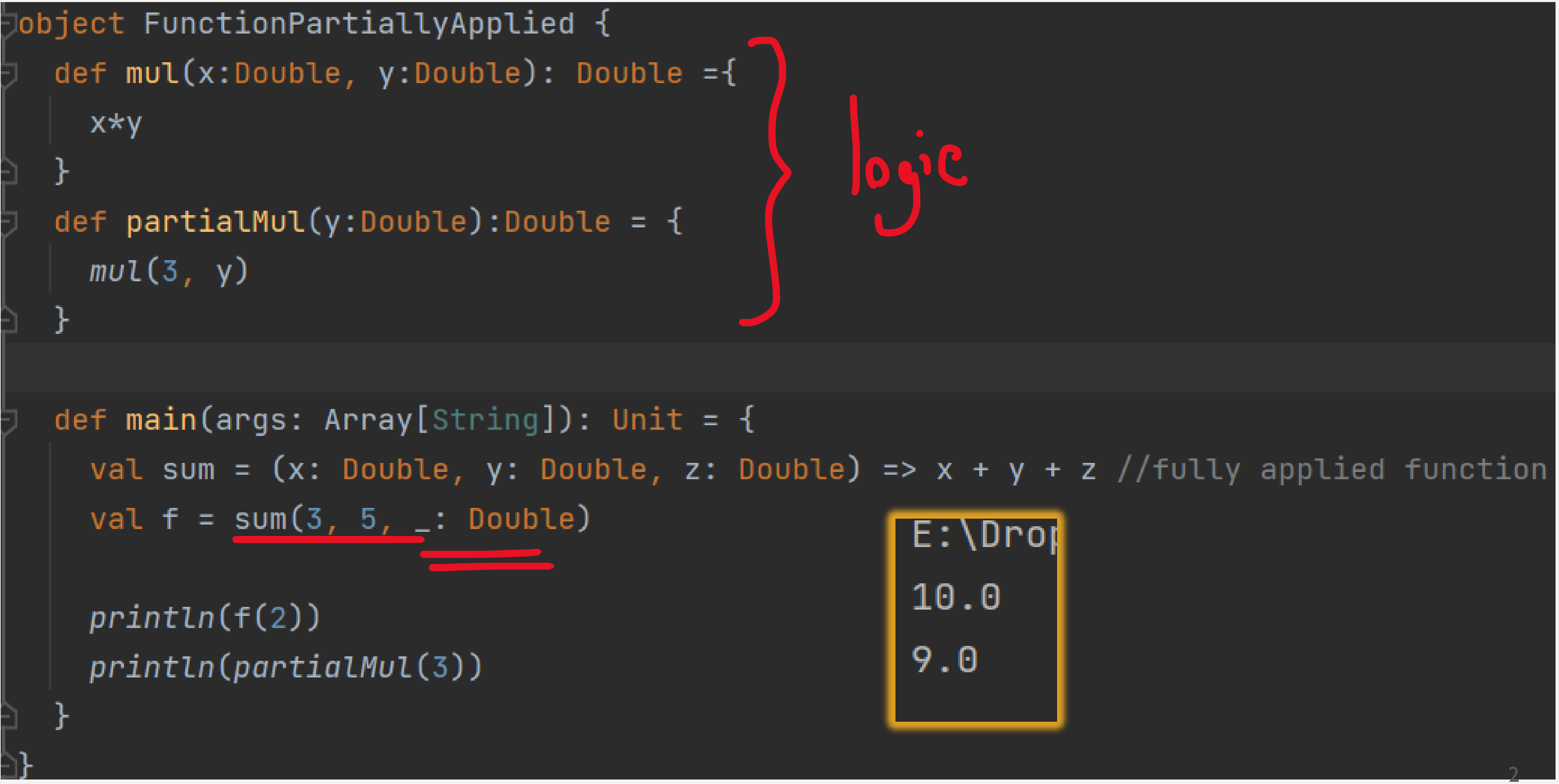
FEATURES: HIGHER ORDER&CURRYING

PARTIALLY APPLIED FUNCTION

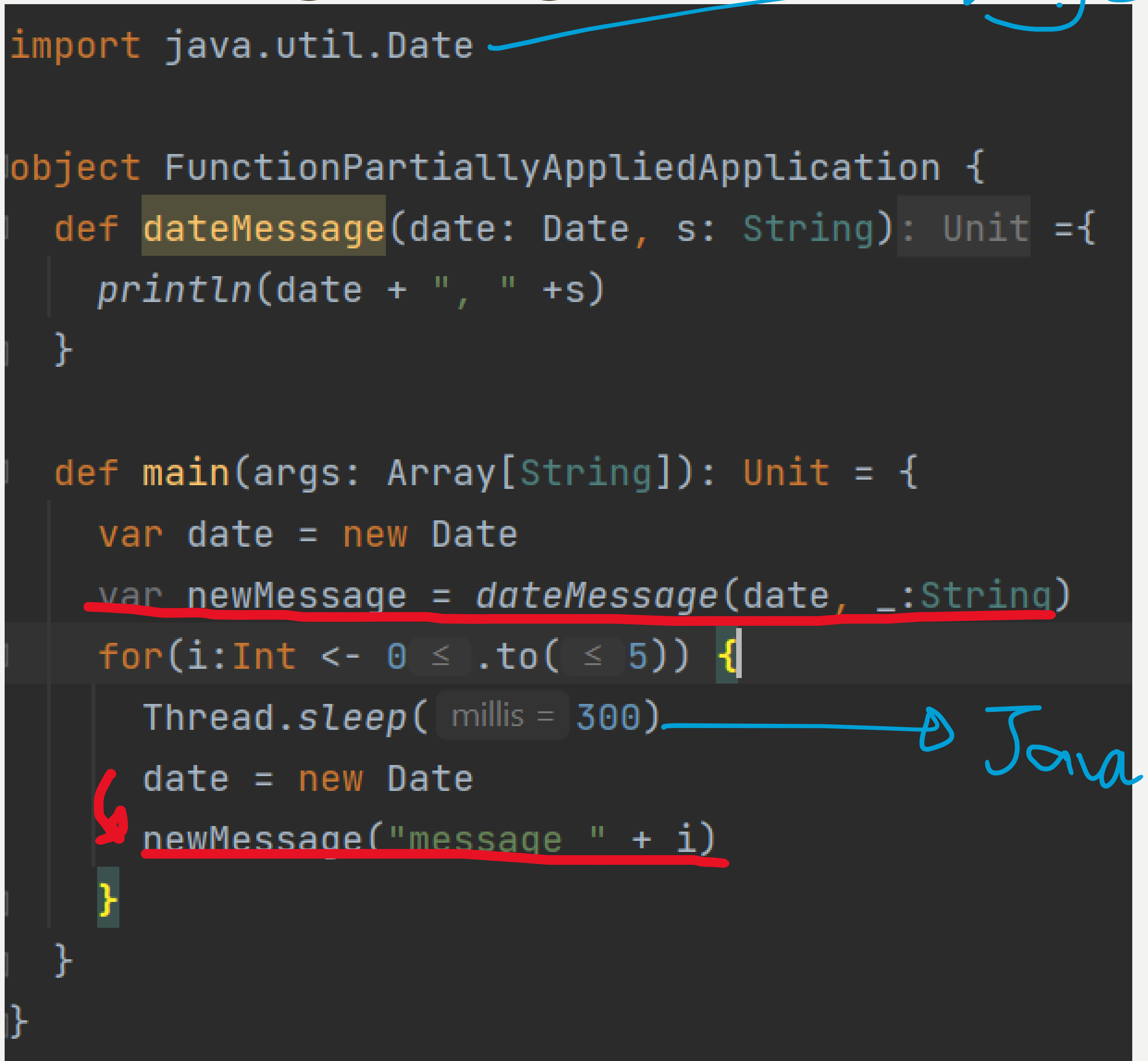


```
object FunctionPartiallyApplied {
  def mul(x:Double, y:Double): Double ={
    x*y
  }
  def partialMul(y:Double):Double = {
    mul(3, y)
  }


  def main(args: Array[String]): Unit = {
    val sum = (x: Double, y: Double, z: Double) => x + y + z //fully applied function
    val f = sum(3, 5, _: Double)

    println(f(2))
    println(partialMul(3))
  }
}
```

logic

```
E:\Drop
10.0
9.0
```

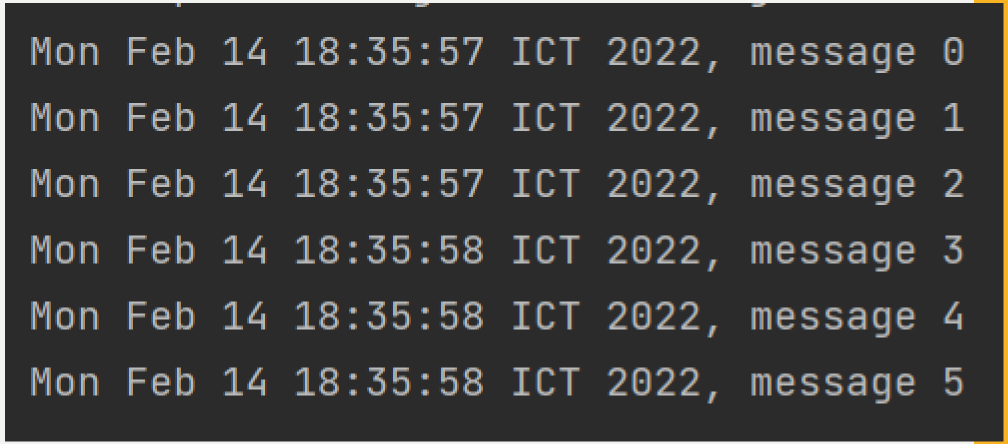
PARTIALLY APPLIED FUNCTION (APPLICATION)



```
import java.util.Date

object FunctionPartiallyAppliedApplication {
  def dateMessage(date: Date, s: String): Unit ={
    println(date + ", " +s)
  }

  def main(args: Array[String]): Unit = {
    var date = new Date
    var newMessage = dateMessage(date, _:String)
    for(i:Int <- 0 ≤ .to( ≤ 5)) {
      Thread.sleep( millis = 300)
      date = new Date
      newMessage("message " + i)
    }
  }
}
```

Java

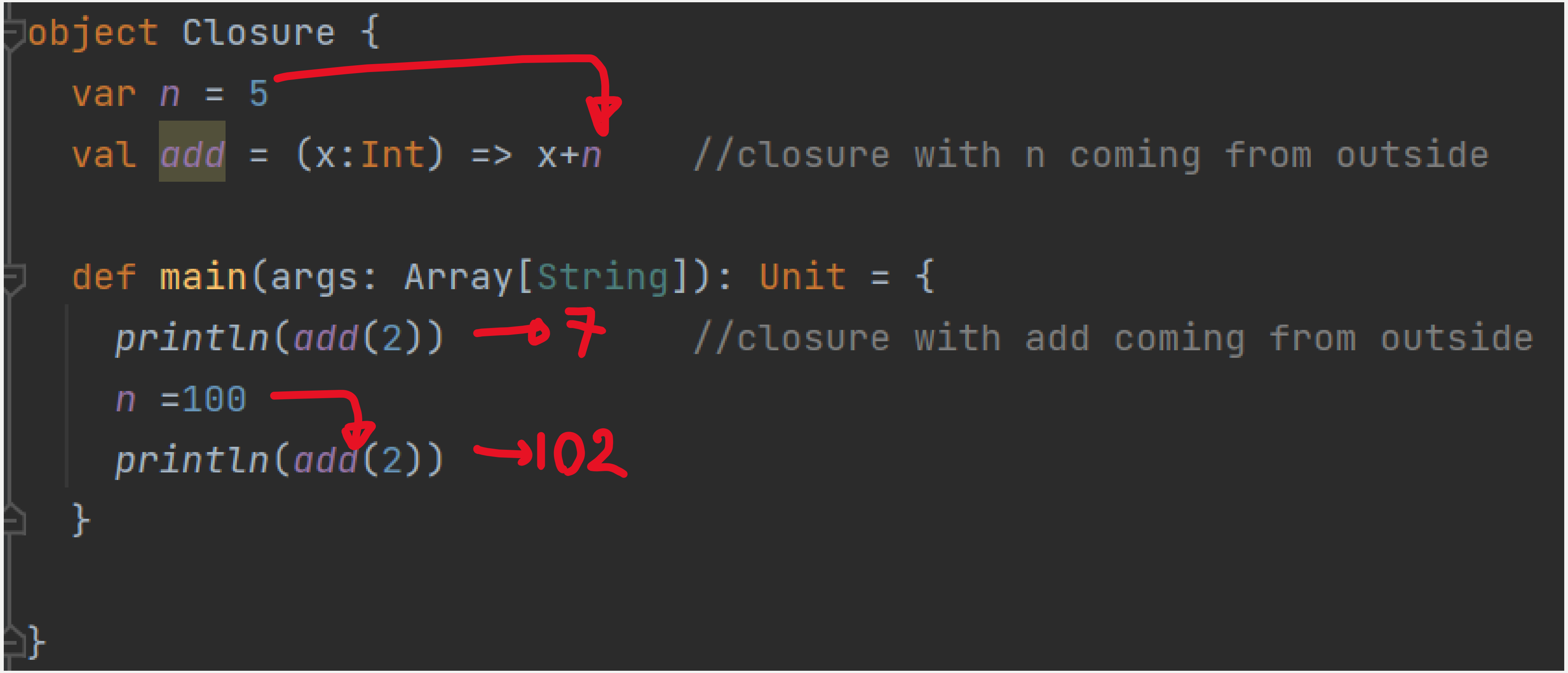
Java



```
Mon Feb 14 18:35:57 ICT 2022, message 0
Mon Feb 14 18:35:57 ICT 2022, message 1
Mon Feb 14 18:35:57 ICT 2022, message 2
Mon Feb 14 18:35:58 ICT 2022, message 3
Mon Feb 14 18:35:58 ICT 2022, message 4
Mon Feb 14 18:35:58 ICT 2022, message 5
```

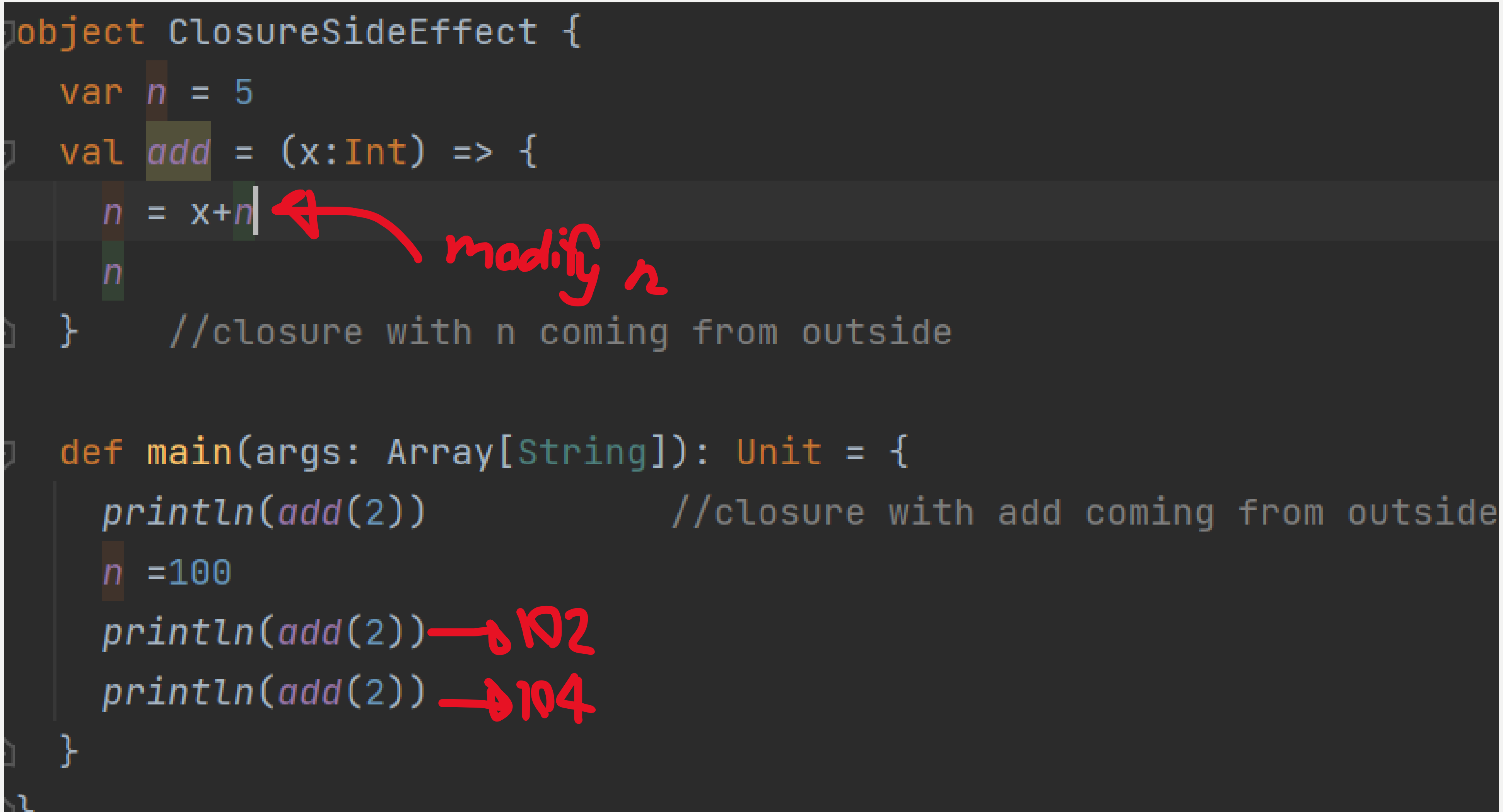
CLOSURE

- A function that uses variable(s) declared outside the function.



```scala
object Closure {
  var n = 5
  val add = (x:Int) => x+n    //closure with n coming from outside

  def main(args: Array[String]): Unit = {
    println(add(2))           //closure with add coming from outside
    n =100
    println(add(2))
  }

}
```

CLOSURE – WITH SIDE EFFECT ALLOWED ON VARIABLE (IMPURE CLOSURE))



```
object ClosureSideEffect {
  var n = 5
  val add = (x:Int) => {
    n = x+n
    n
  }    //closure with n coming from outside

  def main(args: Array[String]): Unit = {
    println(add(2))          //closure with add coming from outside
    n =100
    println(add(2))
    println(add(2))
  }
}
```

WHAT IS FUNCTIONAL PROGRAMMING?

- No changing variable.
- No assignment
- No loop
- Just focusing on functions.
- Functions can be defined anywhere, including in other functions.
- Functions can be passed as parameters and returned as results.
- There are operators that can compose functions.

WHAT ARE GOOD ABOUT FUNCTIONAL PROGRAMMING?

- Simpler reasoning.
- Good for multicore and cloud computing.
  - Avoid modifying variables by different parts of the program.
- Places to use (where we want scalable solutions)
  - Web
  - Trading platforms
  - Simulation

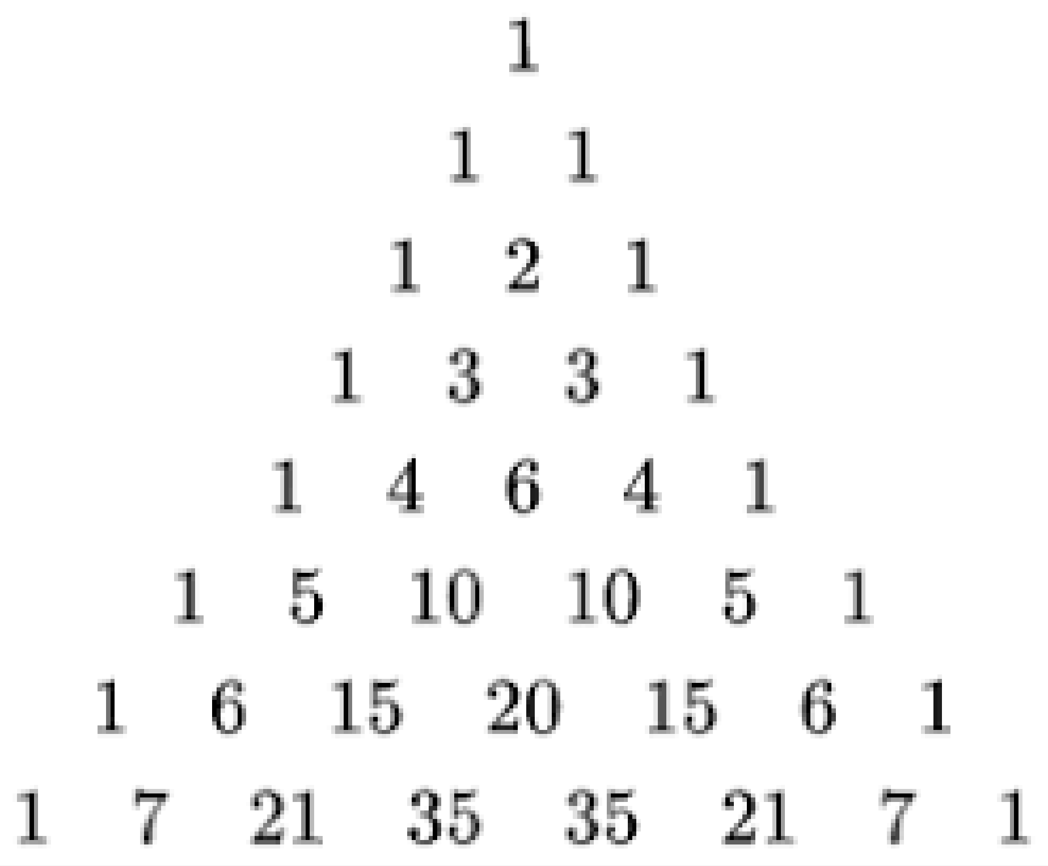
EVALUATING FUNCTION == EVALUATING EXPRESSION

- This substitution model (evaluating until getting a value) can be used as long as the function has no side effect.
  - square(square(2))
  - square(4)
  - 16
- Example of side effect (cannot be expressed in a substitution model)
  - x++

RECURSION IS IMPORTANT IN THIS PARADIGM.

- Need to be able to think of it instead of loop.
- Recursion can be optimized to use only 1 stack frame (if you convert it to tail-recursion)
- But first, you must be more familiar with recursion.

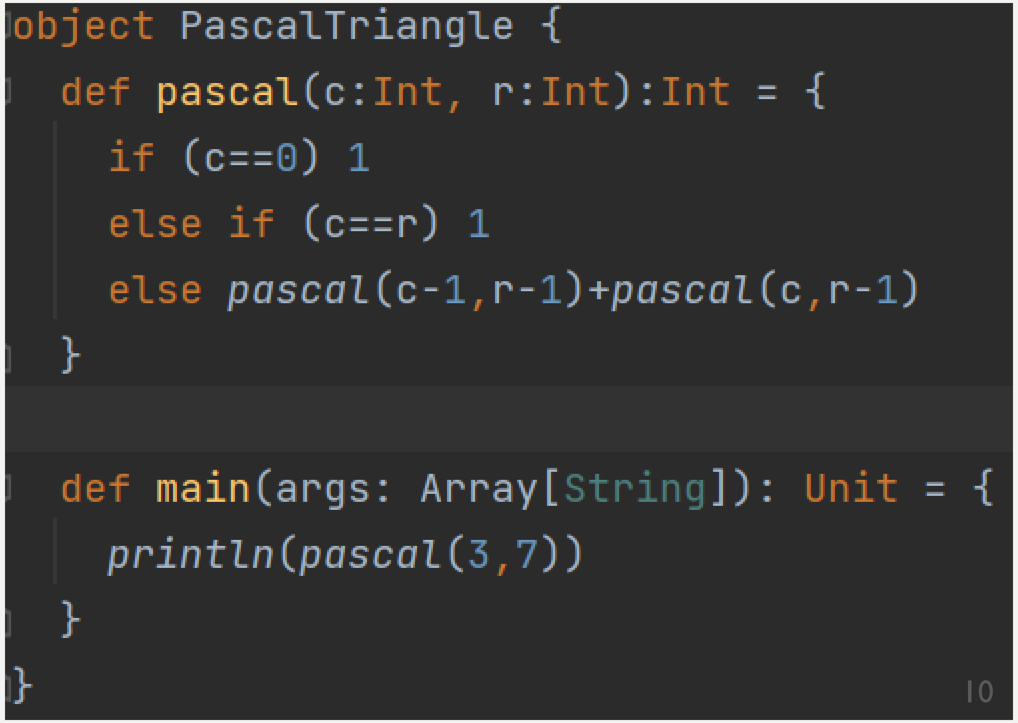
PASCAL'S TRIANGLE (RECURSION EXERCISE – 5 MINS)



```
              1
            1   1
          1   2   1
        1   3   3   1
      1   4   6   4   1
    1   5  10  10   5   1
  1   6  15  20  15   6   1
1   7  21  35  35  21   7   1
```

```
def pascal(c: Int, r: Int): Int
```

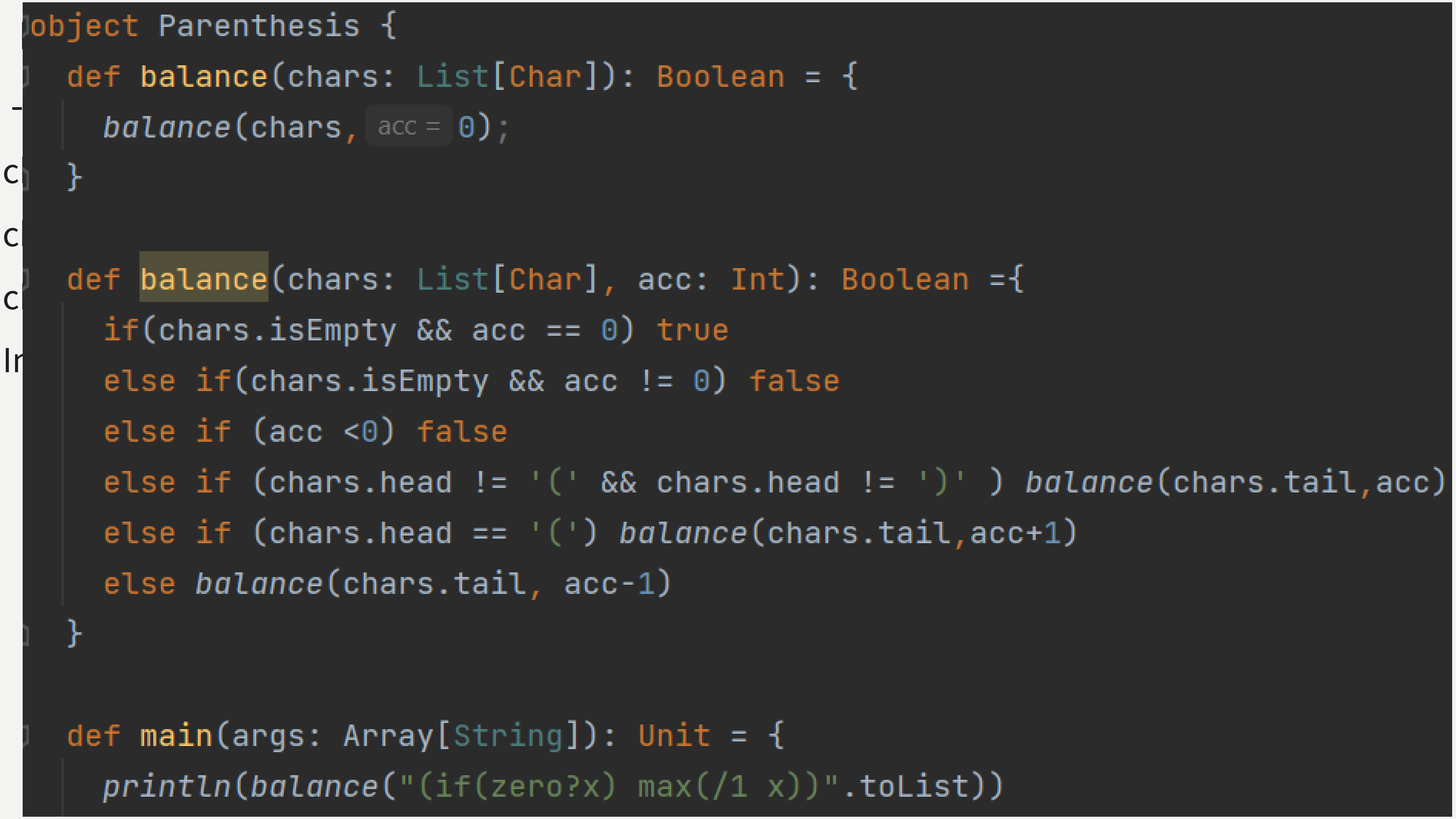
`Returns the number at column c in row r , where c and r start at 0, and value of c never exceeds value of r.`



```
object PascalTriangle {
  def pascal(c:Int, r:Int):Int = {
    if (c==0) 1
    else if (c==r) 1
    else pascal(c-1,r-1)+pascal(c,r-1)
  }

  def main(args: Array[String]): Unit = {
    println(pascal(3,7))
  }
}
```

PARENTHESIS BALANCING EXERCISE (RECURSIVE 15 MINS)

- def
- ())(-
  - c
  - c
  - c
  - Ir

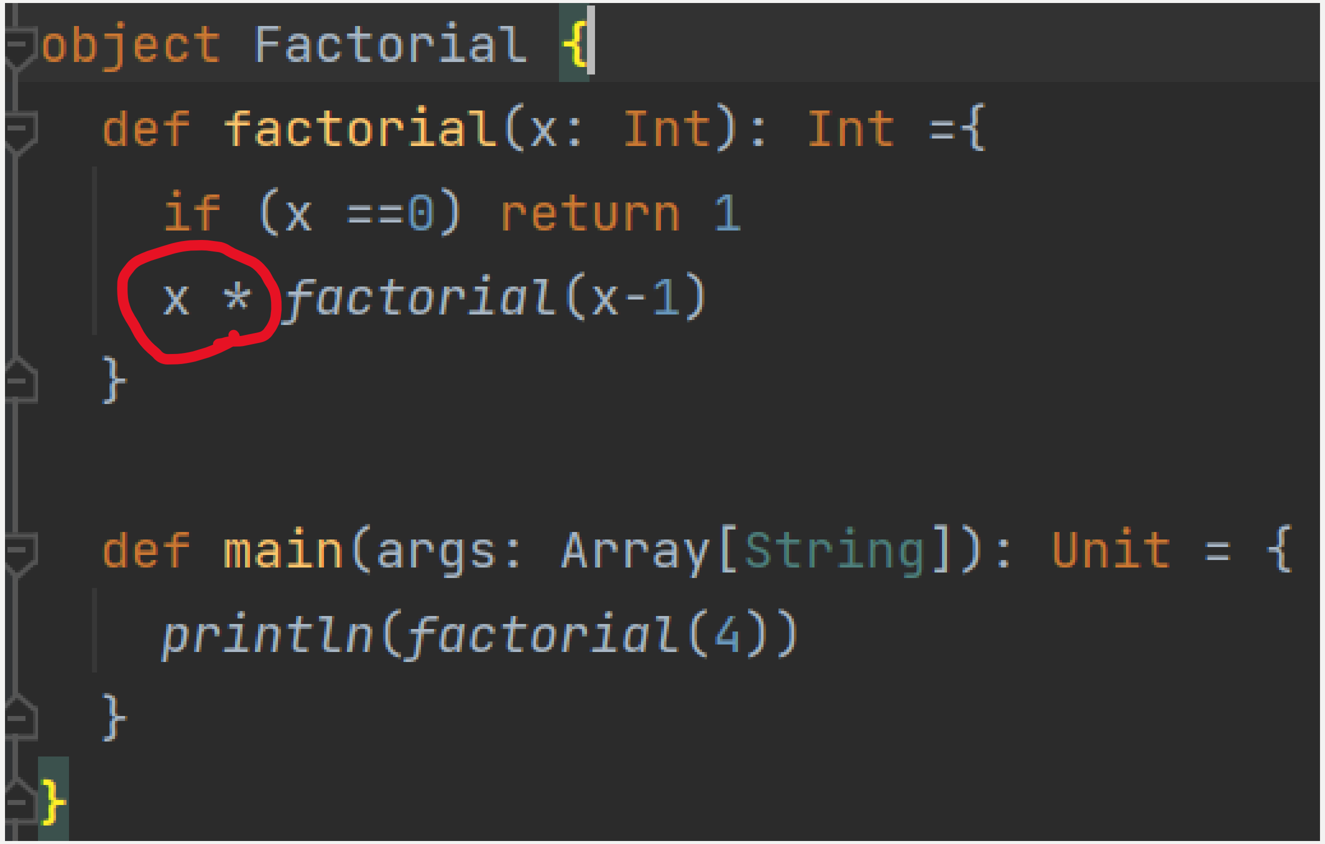


```
object Parenthesis {
  def balance(chars: List[Char]): Boolean = {
    balance(chars, acc = 0);
  }

  def balance(chars: List[Char], acc: Int): Boolean ={
    if(chars.isEmpty && acc == 0) true
    else if(chars.isEmpty && acc != 0) false
    else if (acc <0) false
    else if (chars.head != '(' && chars.head != ')' ) balance(chars.tail,acc)
    else if (chars.head == '(') balance(chars.tail,acc+1)
    else balance(chars.tail, acc-1)
  }

  def main(args: Array[String]): Unit = {
    println(balance("(if(zero?x) max(/1 x))".toList))
```

TAIL RECURSION

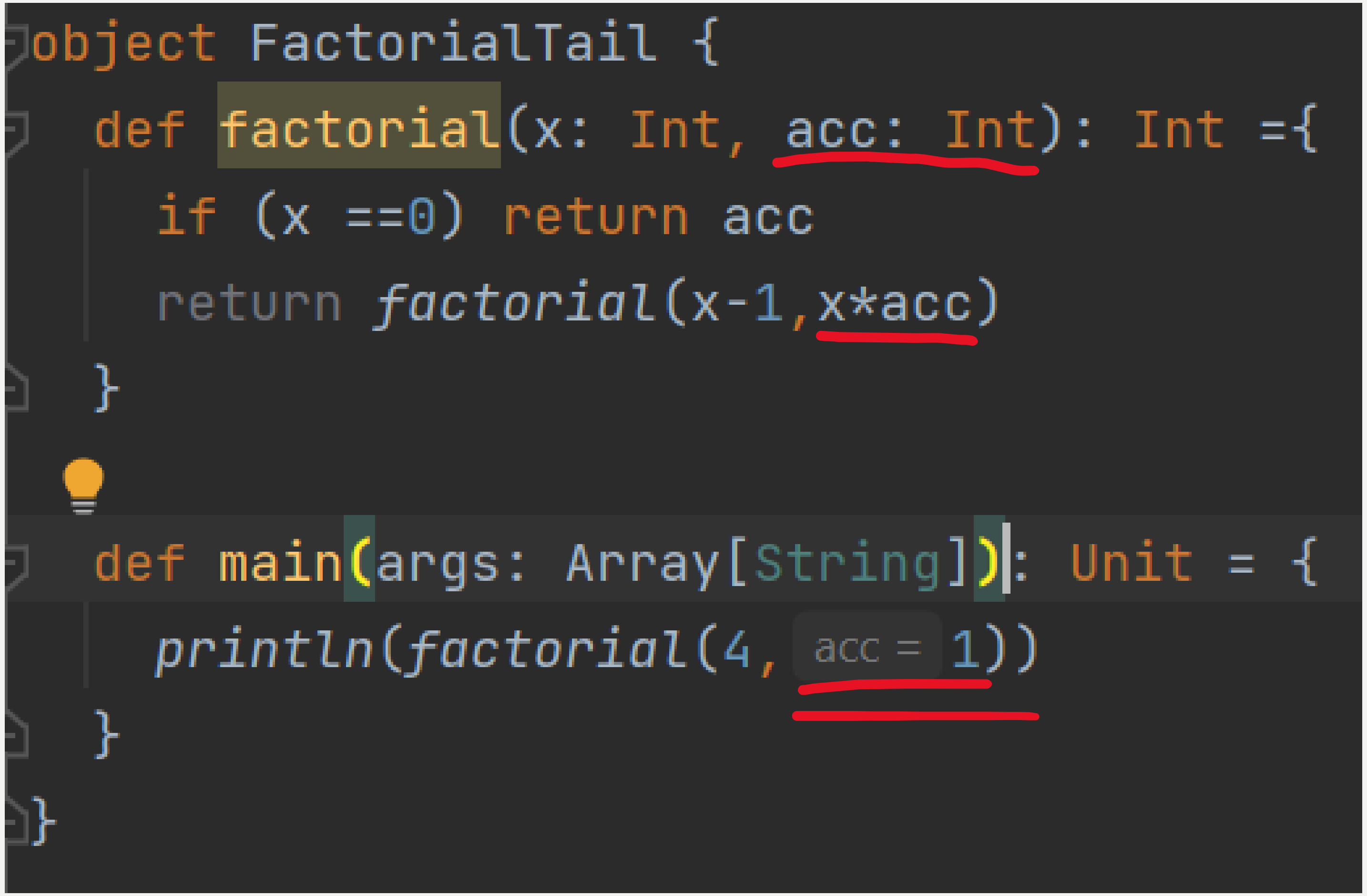
- If a function just calls another or call itself without any extra work, the language runtime system can optimize the function to use only one stack frame, just like using a loop.
- If you see a recursive function that is not tail-recursive, trying to make it tail-recursive will help optimize memory (stack frame) usage.

FACTORIAL (NON TAIL-RECURSIVE)



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

  def main(args: Array[String]): Unit = {
    println(factorial(4))
  }
}
```

FACTORIAL (TAIL-RECURSIVE) -EXERCISE 5 MINS

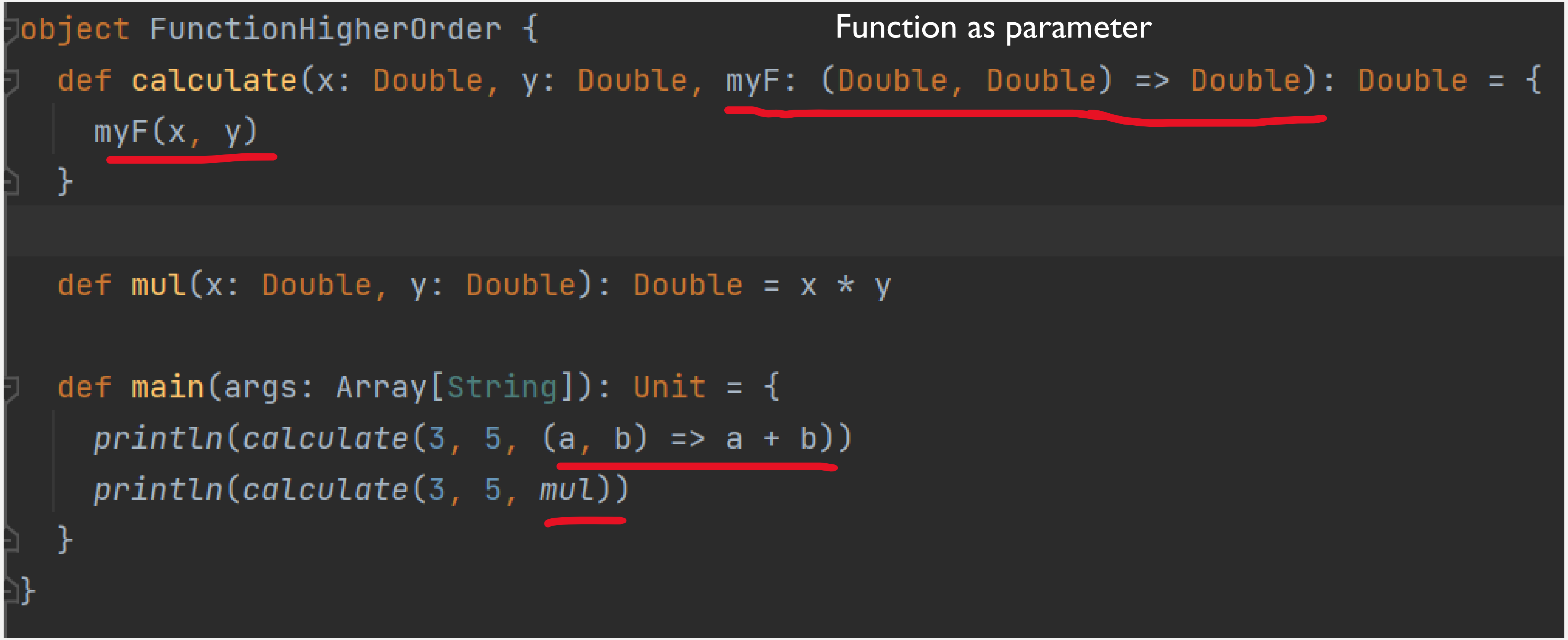


```
object FactorialTail {
  def factorial(x: Int, acc: Int): Int ={
    if (x ==0) return acc
    return factorial(x-1,x*acc)
  }

  def main(args: Array[String]): Unit = {
    println(factorial(4, acc = 1))
  }
}
```

HIGHER ORDER FUNCTION

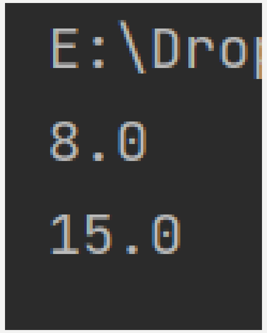
Take functions as arguments.

Can return function.



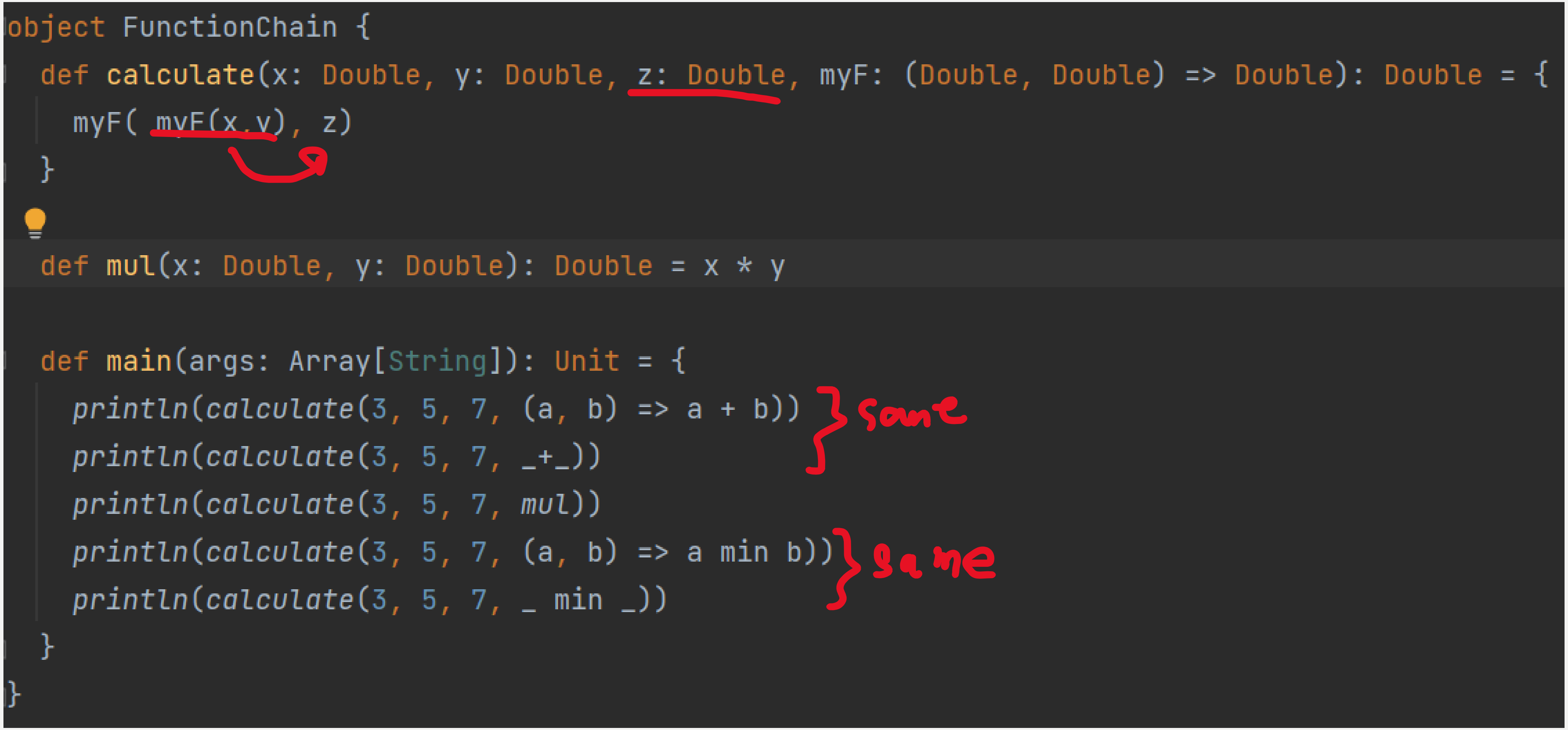
Function as parameter

```scala
object FunctionHigherOrder {
  def calculate(x: Double, y: Double, myF: (Double, Double) => Double): Double = {
    myF(x, y)
  }

  def mul(x: Double, y: Double): Double = x * y

  def main(args: Array[String]): Unit = {
    println(calculate(3, 5, (a, b) => a + b))
    println(calculate(3, 5, mul))
  }
}
```



```
E:\Drop
8.0
15.0
```

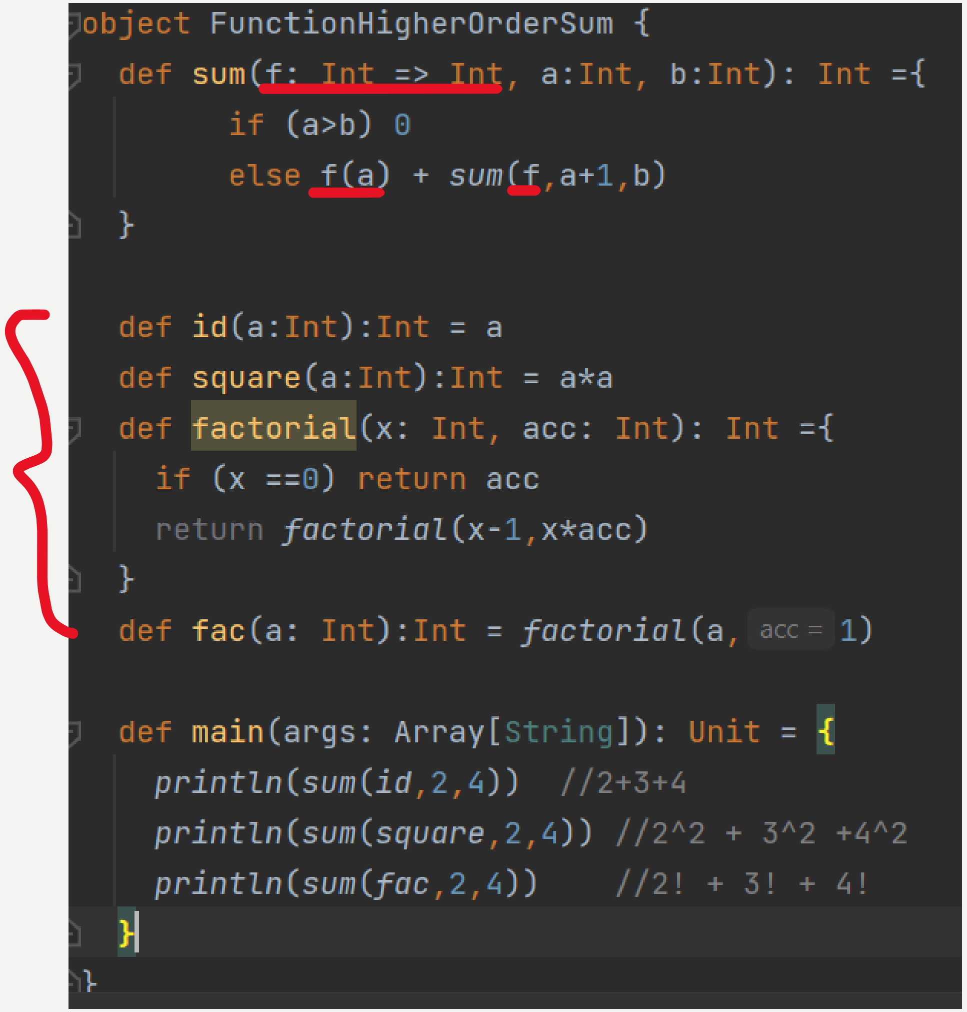
CHAINING FUNCTIONS



```
object FunctionChain {
  def calculate(x: Double, y: Double, z: Double, myF: (Double, Double) => Double): Double = {
    myF( myF(x,y), z)
  }

  def mul(x: Double, y: Double): Double = x * y

  def main(args: Array[String]): Unit = {
    println(calculate(3, 5, 7, (a, b) => a + b))
    println(calculate(3, 5, 7, _+_))
    println(calculate(3, 5, 7, mul))
    println(calculate(3, 5, 7, (a, b) => a min b))
    println(calculate(3, 5, 7, _ min _))
  }
}
```

LET'S DEFINE $\sum_{n=a}^{b} f\ (n)$ WHERE F CAN BE ANY FUNCTION

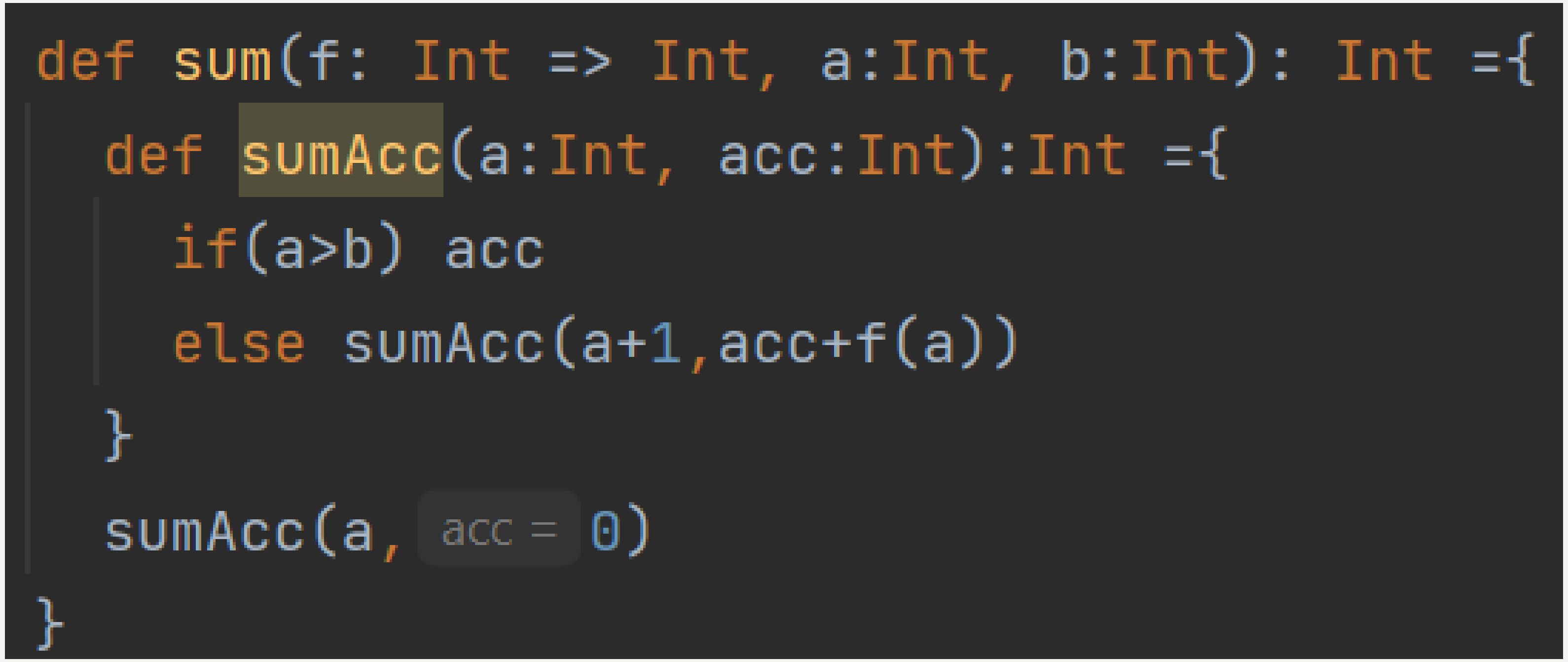
f



```
object FunctionHigherOrderSum {
  def sum(f: Int => Int, a:Int, b:Int): Int ={
        if (a>b) 0
        else f(a) + sum(f,a+1,b)
  }

  def id(a:Int):Int = a
  def square(a:Int):Int = a*a
  def factorial(x: Int, acc: Int): Int ={
    if (x ==0) return acc
    return factorial(x-1,x*acc)
  }
  def fac(a: Int):Int = factorial(a, acc = 1)

  def main(args: Array[String]): Unit = {
    println(sum(id,2,4))  //2+3+4
    println(sum(square,2,4)) //2^2 + 3^2 +4^2
    println(sum(fac,2,4))    //2! + 3! + 4!
  }
}
```

$\sum_{n=a}^{b} f\ (n)$ CAN BE WRITTEN USING TAIL RECURSION TOO (EXERCISE – 5 MINS)

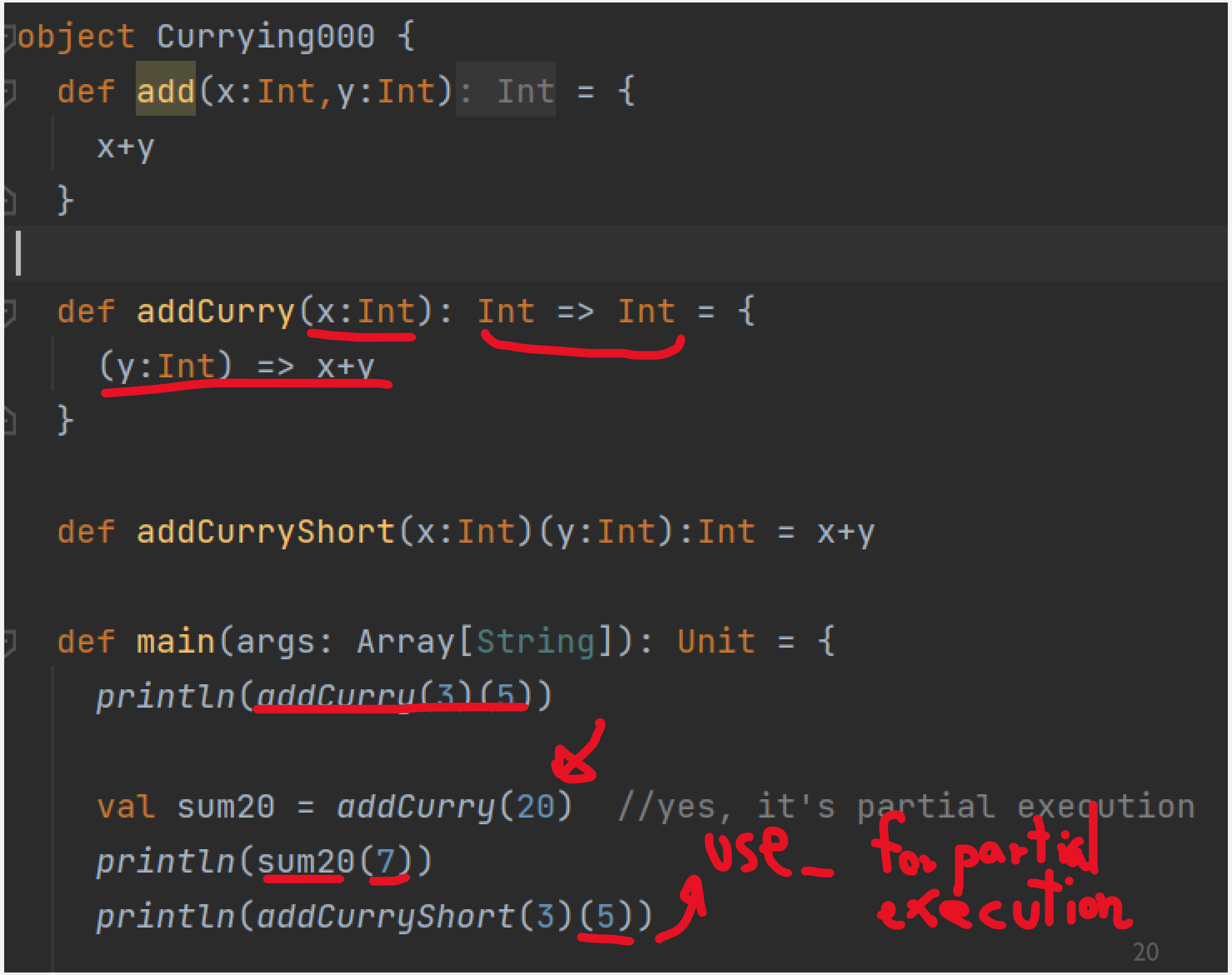
- Write only the definition of function sum



```scala
def sum(f: Int => Int, a:Int, b:Int): Int ={
  def sumAcc(a:Int, acc:Int):Int ={
    if(a>b) acc
    else sumAcc(a+1,acc+f(a))
  }
  sumAcc(a, acc = 0)
}
```

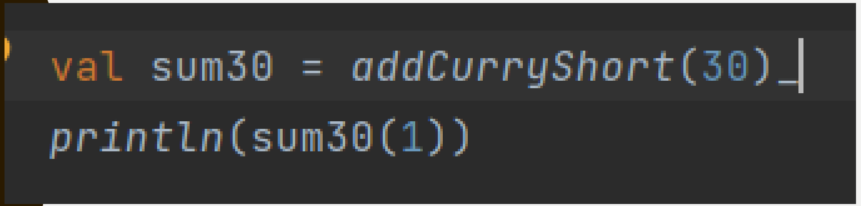
CURRYING - FUNCTION AS RETURN VALUE

- Function with multiple arguments ->
  - Function with one argument, returning another function.



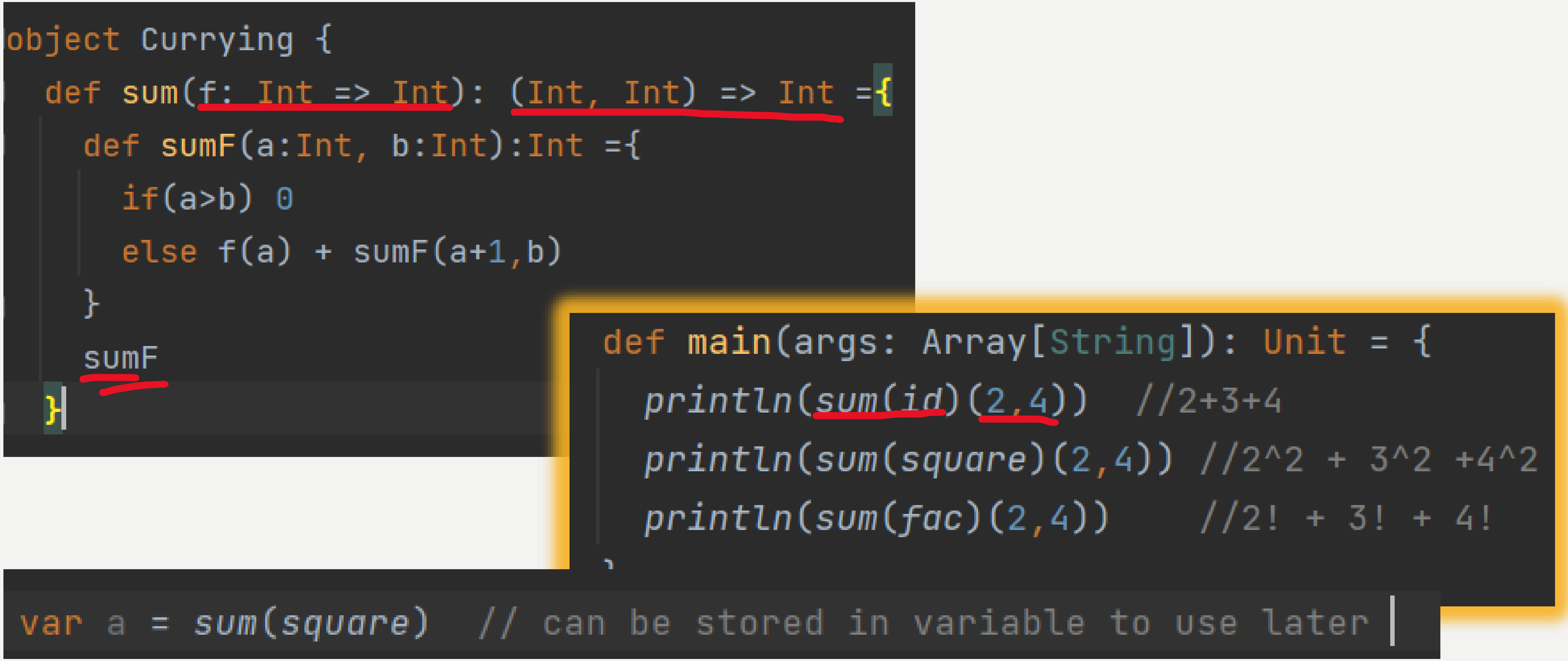
```scala
object Currying000 {
  def add(x:Int,y:Int): Int = {
    x+y
  }

  def addCurry(x:Int): Int => Int = {
    (y:Int) => x+y
  }

  def addCurryShort(x:Int)(y:Int):Int = x+y

  def main(args: Array[String]): Unit = {
    println(addCurry(3)(5))

    val sum20 = addCurry(20)  //yes, it's partial execution
    println(sum20(7))
    println(addCurryShort(3)(5))
```

use _ for partial execution



```scala
val sum30 = addCurryShort(30)_
println(sum30(1))
```

CURRYING – Example on $\sum_{n=a}^{b} f(n)$

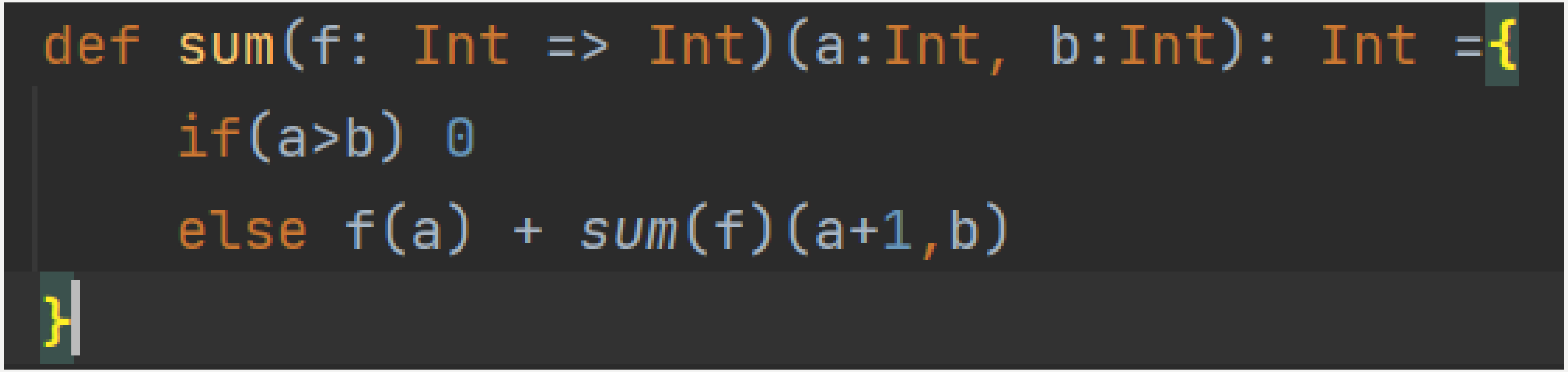


```
object Currying {
  def sum(f: Int => Int): (Int, Int) => Int ={
    def sumF(a:Int, b:Int):Int ={
      if(a>b) 0
      else f(a) + sumF(a+1,b)
    }
    sumF
  }
```

```
def main(args: Array[String]): Unit = {
  println(sum(id)(2,4))  //2+3+4
  println(sum(square)(2,4)) //2^2 + 3^2 +4^2
  println(sum(fac)(2,4))    //2! + 3! + 4!
}
```

```
var a = sum(square)  // can be stored in variable to use later
```

CURRYING – SPECIAL SYNTAX (MULTIPLE PARAMETER LIST)

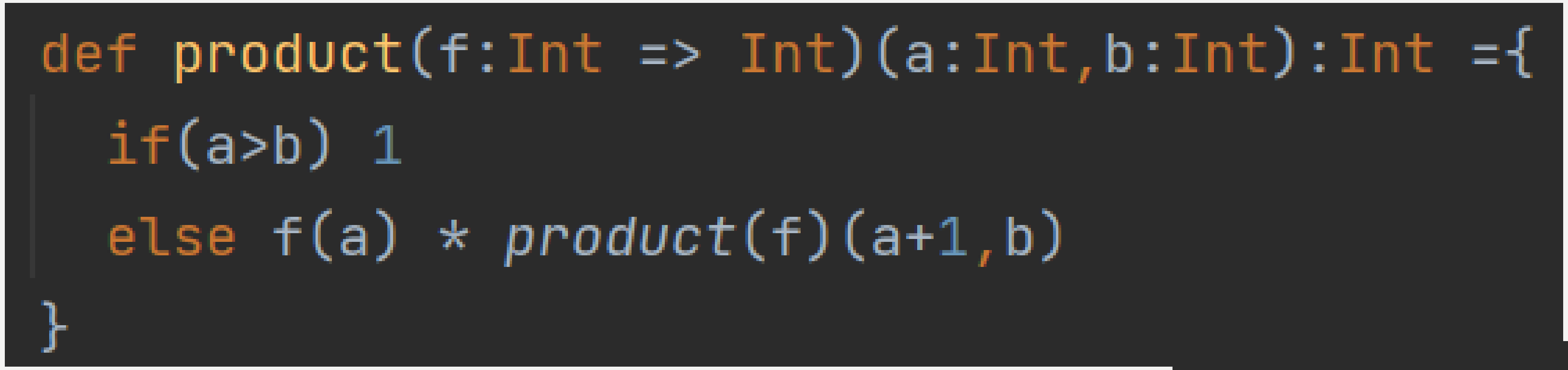


```
def sum(f: Int => Int)(a:Int, b:Int): Int ={
      if(a>b) 0
      else f(a) + sum(f)(a+1,b)
}
```

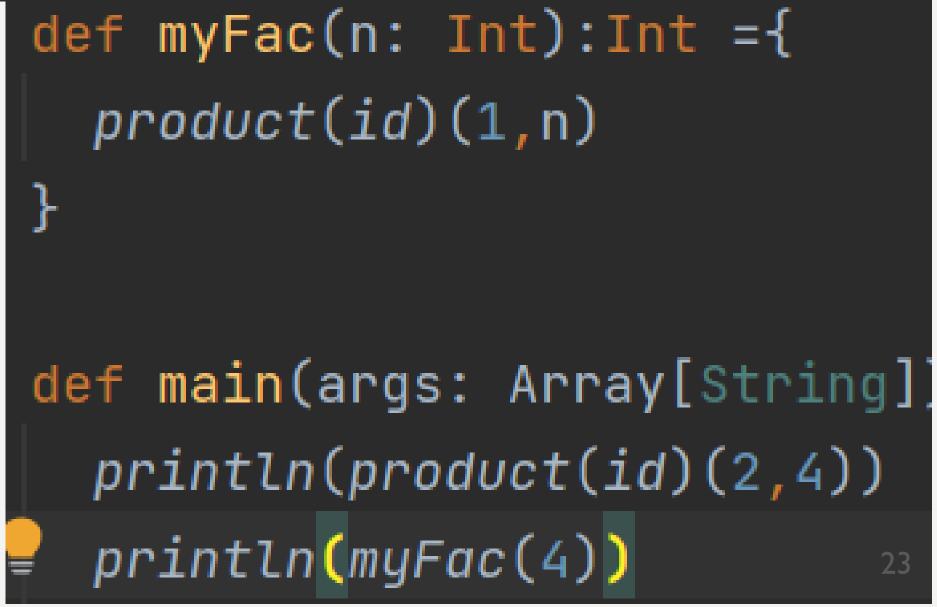
The type of this function is
(Int => Int) => ((Int,Int) => Int)   or (Int => Int) => (Int,Int) => Int

Since function types are right associative, so Int => Int => Int is equivalent to Int => (Int => Int)

EXERCISE: FACTORIAL IN TERMS OF PRODUCT? – 2 MINS



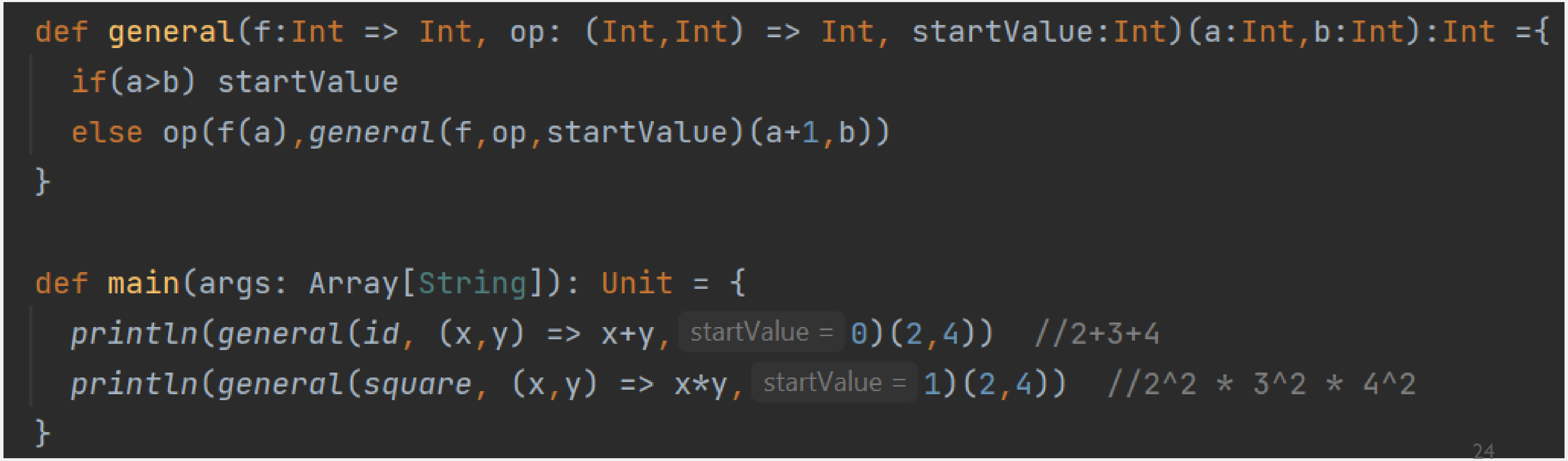
```
def product(f:Int => Int)(a:Int,b:Int):Int ={
  if(a>b) 1
  else f(a) * product(f)(a+1,b)
}
```



```
def myFac(n: Int):Int ={
  product(id)(1,n)
}

def main(args: Array[String]
  println(product(id)(2,4))
  println(myFac(4))
```

EXERCISE: WRITE A FUNCTION THAT CAN BE CHANGED TO USE EITHER SUM OR PRODUCT (EACH WITH 2 PARAMETER LIST) – 5 MINS

- Using the new function, in main, calculate 2+3+4 and 2^2 * 3^2 * 4^2



```
def general(f:Int => Int, op: (Int,Int) => Int, startValue:Int)(a:Int,b:Int):Int ={
  if(a>b) startValue
  else op(f(a),general(f,op,startValue)(a+1,b))
}

def main(args: Array[String]): Unit = {
  println(general(id, (x,y) => x+y, startValue = 0)(2,4))  //2+3+4
  println(general(square, (x,y) => x*y, startValue = 1)(2,4))  //2^2 * 3^2 * 4^2
}
```