

# 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 E:\Drop  
    println(f(2)) 10.0  
    println(partialMul(3)) 9.0  
  }  
}
```

some parameter is pre-defined

logic

specify type of input

parameters that need to be fill later input !

# PARTIALLY APPLIED FUNCTION (APPLICATION)

Java

```
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)
    }
  }
}
```

predefined

Receive i as input

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.

```
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)) → 7      //closure with add coming from outside  
    n = 100 → Change value of n !  
    println(add(2)) → 102  
  }  
}
```


# CLOSURE – WITH SIDE EFFECT ALLOWED ON VARIABLE (IMPURE

```
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))  
    n = 100  
    println(add(2))  
    println(add(2))  
  }  
}
```

*Handwritten annotations:*

- A red arrow points from the text modify n to the line `n = x+n` inside the closure.
- Red arrows point from the two `println(add(2))` calls in the `main` function to the values **102** and **104** respectively.
- A bracket on the right side of these values is accompanied by the text *n was modified in closure*.

# 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: *Logical*
- 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. ! no change to others

- square(square(2))
- square(4)
- 16

$\text{add}(x, y) \Rightarrow x + y$


call by value :  $\text{add}(5+4, 5+1) \longrightarrow \text{add}(9, 6) \xrightarrow{\text{access function}}$

call by name :  $(5+4) + (5+1)$

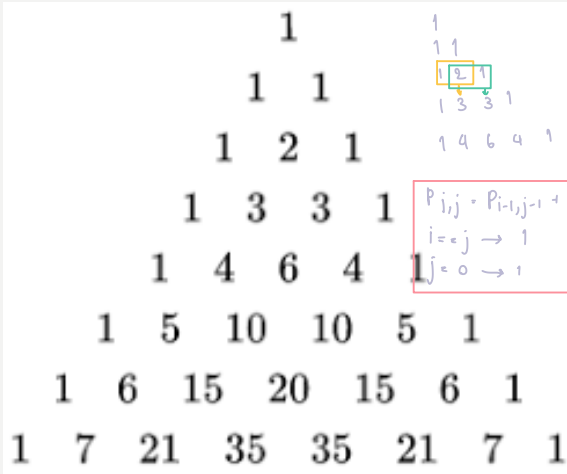
- Example of side effect (cannot be expressed in a substitution model)
  - $x++$   $x$  is changed !



# 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)



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

# PARENTHESIS BALANCING EXERCISE (RECURSIVE 15 MINS)

- `def balance`
- `()()`
- `ch`
- `ch`
- `ch`
- `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"))  
  }  
}
```


default is "0"

state of parenthesis: "(" → +1, ")" → -1

not parenthesis case

List without head

# 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)  
  }  
  // ↳ extra operation with function call  
  
  def main(args: Array[String]): Unit = {  
    println(factorial(4))  
  }  
}
```

1 2 3 4

1+2+3

# 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) * no operation need  
                                     after calling function  
  }  
  
  def main(args: Array[String]): Unit = {  
    println(factorial(4, acc = 1))  
  }  
}
```

Variables:  
store value each recursion

# HIGHER ORDER FUNCTION

---

Pass / Return Function

①

Take functions as arguments.

②

Can return function.

```

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))
}

```

Function as parameter

Return Function

Pass Function

anonymous function

defined function

```

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)  
  }
```

*Disrupt Orderly*



```
  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, (a, b) => a + b))  
    println(calculate(3, 5, 7, mul))  
    println(calculate(3, 5, 7, (a, b) => a min b))  
    println(calculate(3, 5, 7, _ min _))  
  }
```

*same*

*function*

*same*

# LET'S DEFINE $\sum_{n=a}^b$

$$(\sum_{n=a}^b f(n))$$

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

$f(a) + \sum_{n=a+1}^b f(n) *$

f {

```
  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!  
  }  
}
```

# $\sum_{a=b}^b f(n)$ CAN BE

- Write only the definition of function sum

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

Recursive Method

# CURRYING - FUNCTION AS RETURN VALUE

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

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

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

  def addCurry(x:Int): Int => Int = {
    (y:Int) => x+y
  }
  // add within and return

  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))
  }
}
```

Handwritten notes on the code:

- Red underline under `(y:Int) => x+y` in `addCurry`.
- Handwritten text "add within and return" below the `addCurry` function.
- Red underline under `addCurryShort`.
- Red underline under `addCurry(3)(5)` in `main`.
- Red arrow pointing from `addCurry(20)` to `sum20` with text "need both params".
- Red arrow pointing from `addCurryShort(3)(5)` to `sum20` with text "use for partial execution".
- Red arrow pointing from `3` in `addCurryShort(3)(5)` to `3` in `sum20(7)` with text "Or (3)".
- Red arrow pointing from `5` in `addCurryShort(3)(5)` to `7` in `sum20(7)` with text "specify later".

# CURRYING -

$\sum_{n=a}^b f(n)$

$\sum h$

```
object Currying {  
  // input function that  $\rightarrow$  Function Condition  
  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)

The diagram shows a Scala function definition for `sum` with handwritten annotations explaining currying. The code is: `def sum(f: Int => Int)(a: Int, b: Int): Int = { if(a > b) 0 else f(a) + sum(f)(a+1, b) }`. Annotations include: a red arrow from `f` to `f(a)` labeled "pass f"; a red arrow from `Int` in the parameter list to `Int` in the return type labeled "Return Int"; a red arrow from `f` to `sum(f)` labeled "which f is"; a red arrow from the `Int` in the return type to the closing brace labeled "this function return"; a yellow arrow from `sum(f)` to `sum(f)(a+1, b)` labeled "pass this into function"; and a yellow squiggly line under `sum(f)(a+1, b)` indicating a recursive call.

```
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

$(\text{Int} \Rightarrow \text{Int}) \Rightarrow ((\text{Int}, \text{Int}) \Rightarrow \text{Int})$  or  $(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int}, \text{Int}) \Rightarrow \text{Int}$

Since function types are right associative, so  $\text{Int} \Rightarrow \text{Int} \Rightarrow \text{Int}$  is equivalent to  $\text{Int} \Rightarrow (\text{Int} \Rightarrow \text{Int})$

# 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  
}
```