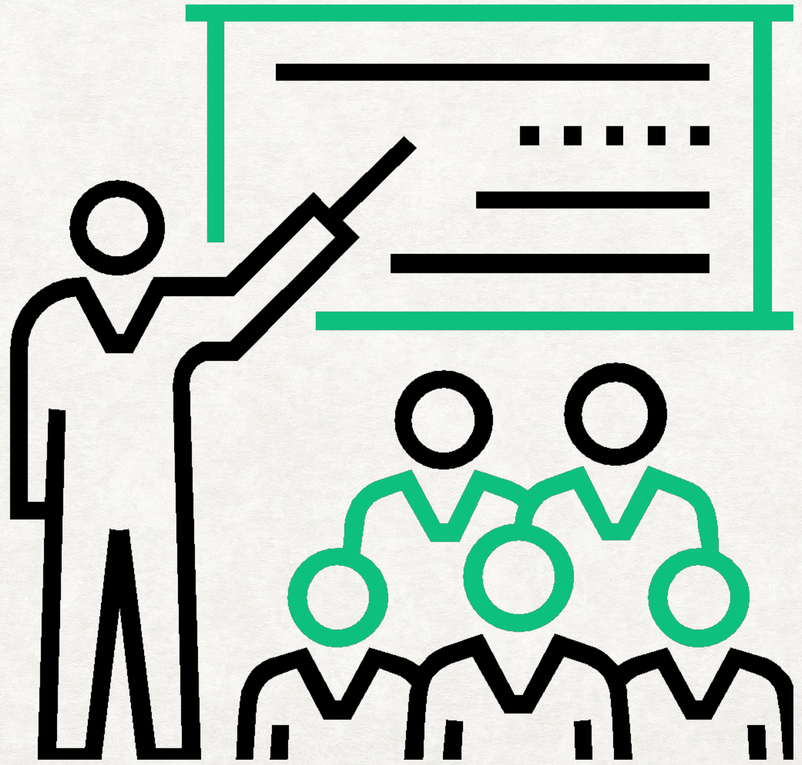
ADVANCED FUNCTIONAL THINKING

DR SURBHI SARASWAT
SCHOOL OF COMPUTER SCIENCE

TOPICS

- **Currying**
- **Use of _ in Scala**



CURRYING

- **Currying** in Scala is simply a technique or a process of transforming a function. This function takes *multiple arguments* into a function that takes a single argument.
- Converting a function like this `f(a,b,c,...)` into a function like this `f(a)(b)(c)...`.
- Reduces the number of arguments that you pass at one time.
- A curried function is a function that keeps returning functions until all its parameters are fulfilled.

```
object Curry
{
    // Define currying function
    def add(x: Int, y: Int) = x + y;

    def main(args: Array[String])
    {
        println(add(20, 19));
    }
}
```

```
object Curry
{
    // transforming the function that
    // takes two(multiple) arguments into
    // a function that takes one(single) argument.
    def add2(a: Int) = (b: Int) => a + b;

    // Main method
    def main(args: Array[String])
    {
        println(add2(20)(19));
    }
}
```

CURRYING

- **Using currying to get the Partially Applied function.**

```
object Curry
{
    def add2(a: Int) = (b: Int) => a + b;

    // Main function
    def main(args: Array[String])
    {
        // Partially Applied function.
        val sum = add2(29);
        println(sum(5));
    }
}
```

```
object Curry
{
    // Curring function declaration
    def add2(a: Int) (b: Int) = a + b;

    def main(args: Array[String])
    {
        // Partially Applied function.
        val sum=add2(29)_;
        println(sum(5));
    }
}
```

- The **Partially applied functions** are the functions which are not applied on all the arguments defined by the stated function i.e, while invoking a function, we can supply some of the arguments and the left arguments are supplied when required.
- These arguments which are not passed to function, we use underscore(_) as a placeholder.

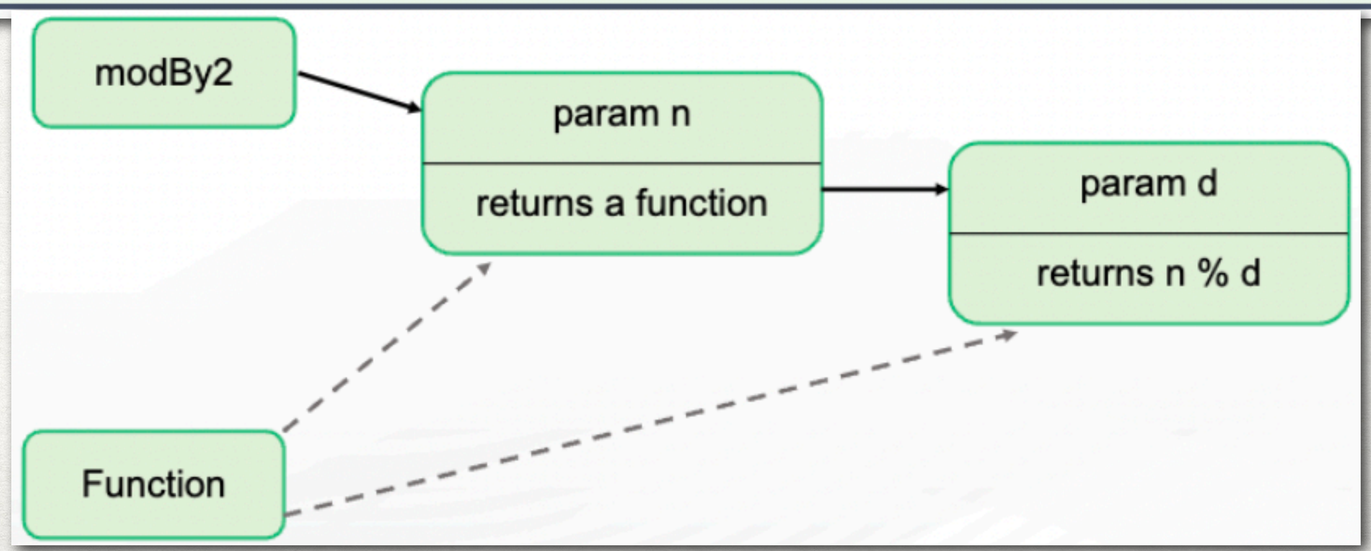
CURRYING

```
//Here, modBy2 method has 2 arguments passed like functions.

scala> def modBy2(n: Int)(d: Int) = n % d
modBy2: (n: Int)(d: Int)Int

scala> modBy2(10)(3)
res0: Int = 1

scala> modBy2 _
res3: Int => (Int => Int) = <function1>
```



```
//Here, modBy2 method has 2 arguments passed like functions.

scala> def modBy2(n: Int)(d: Int) = n % d
modBy2: (n: Int)(d: Int)Int

scala> modBy2(10)(3)
res0: Int = 1

scala> modBy2 _
res3: Int => (Int => Int) = <function1>
```

```
scala> modBy2(10) _
res5: Int => Int = <function1>

scala> val p = modBy2(10) _ // n is fixed to 10
p: Int => Int = <function1>

scala> p(2)
res6: Int = 0  // 10 % 2
scala> p(3) // 10 % 3
res7: Int = 1
```

CURRYING

- **Frequent Function Calls:** Currying is helpful when you have to frequently call a function with a fixed argument.

```
def log(type, msg) {
    if (type == "error")
      console.error(msg);
    if (type == "warn")
      console.warn(msg);
    if (type == "info")
      console.info(msg);
}
```

```
// Without currying
const error = msg => log("error", msg);
const warn = msg => log("warn", msg);
const info = msg => log("info", msg);

// With currying
log = curry(log)
const error = log("error")
const warn = log("warn")
const info = log("info")
```

CURRYING

- Helps in creating higher-order functions
- Cumulative Sum: Let’s assume we want to get a cumulative sum of an array.

```
val cumSum = sum => value => sum += value;

Val sumArray = array.map(cumSum(0))

// -> array

// <- [5, 23, 6, 8, 34]

// -> sumArray

// <- [5, 28, 34, 42, 76]
```

USE OF _ IN PATTERN MATCHING

```
scala> def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "anything other than one and two"
}
matchTest: (x: Int)String

scala> matchTest(7)
res0: String = anything other than one and two


scala> matchTest(1)
res1: String = one
```

USE OF _ IN ANONYMOUS FUNCTIONS

```
Scala> List(1,2,3,4,5).foreach(print(_))
res0: 12345

Above _ can be represented as,
Scala> List(1,2,3,4,5).foreach( a => print(a))
res1: 12345

scala> val sum = List(1,2,3,4,5).reduceLeft(_+_)
res0: sum: Int = 15

Above _+_ can be represented as,
scala> val sum = List(1,2,3,4,5).reduceLeft((a, b) => a + b)
res1: sum: Int = 15
```

1. What does below scala program results?

```
object gfg
{
  def main(args: Array[String])
    {
        var name = (15, "chandan", true)
        println(name._1)
    }
}
```

A) chandan
B) 15
C) _15
D) _chandan

1. What does the below given partially applied function results?

```
val divide = (num: Double, den: Double) => {
  num / den
}
val halfOf: (Double) => Double = divide(_, 2)
halfOf(20)
```

A) 20/2
B) 20.0
C) 0.5
D) 10.0