Functional Programming Learning

In Scala

ரஸ்மிவன் கன்னிப்பேச்சு

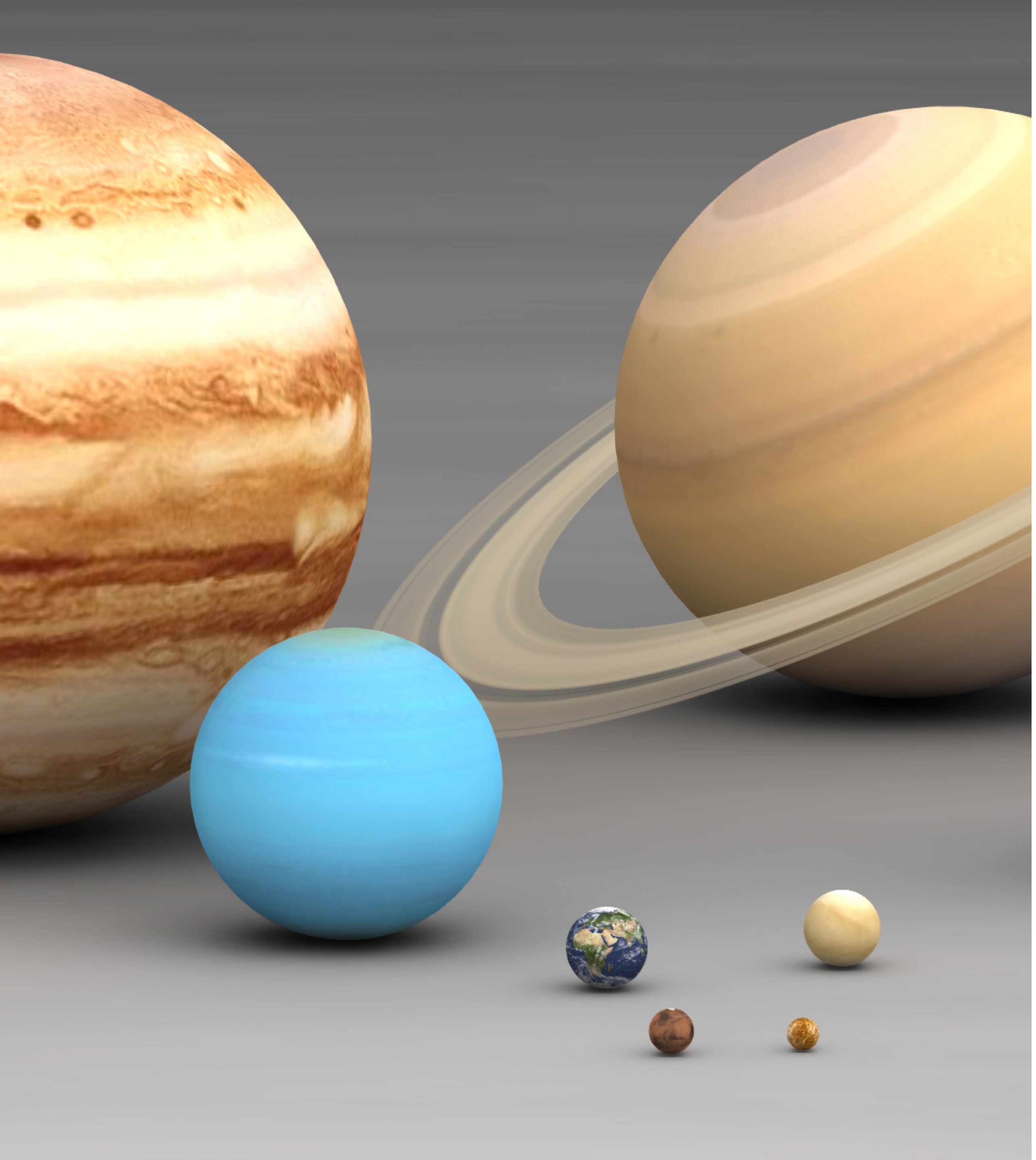
Whats is Functional Programming?

Its a Programming paradigm!

Some Programing paradigm are `Object-Oriented Programming`, `imperative programming`

Whats is Functional Programming?

```
A coding style
A mindset
```



In this Presentaion

Maybe an electron.

Share few programs!

Whats Qualifys as Functional Programming for Scala?

- Higher-order functions
- Immutability
- First class modules
- Currying
- Lambdas
- Closures
- Pattern matching
- Laziness

Whats Qualifys as Functional Programming for Scala?

- Monadic comprehensions
- Algebraic data types
- Type classes
- Higher-kinded types
- Phantom types
- Existential types
- Type-level programming

How to do it?

Do Everything as Function

```
Input => Output
```

Non Functional:

```
public class CubeSquare {
    public static void main(String args[]) {
        int number = 3;
        int square;
        int cube;

        System.out.println("\nNumber\tSquare\tCubes");
        square = number * number;
        cube = square * number;
        System.out.printf(" %d\t \t%d\t \t%d\n", number, square, cube);
    }//end main
}

Output:
    Number      Square      Cubes
    3           9               27
```

Functional Way:

```
def square(num: Int): Int = num * num
def cube(num: Int): Int = square(num) * num
def display(number: Int): Unit = {
    println(s"\nNumber\tSquare\tCubes")
    println(s"$number \t ${square(number)}\t ${cube(number)}")
}

display(3)

Output:
    Number      Square  Cubes
    3       9           27
```

Impure Function

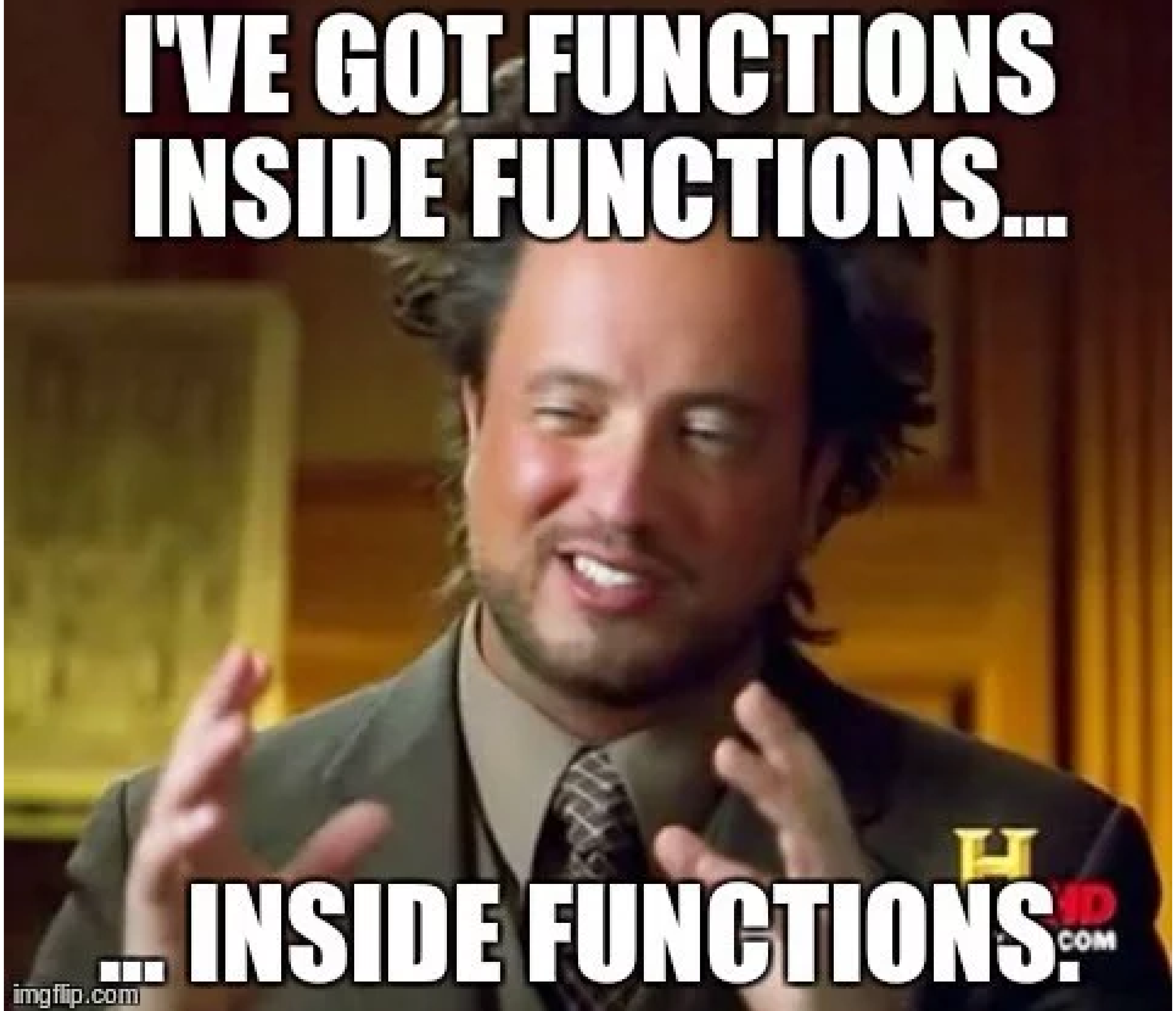
```
In general, you should watch out for functions with a return type of Unit.
Because those functions do not return anything, logically the only reason you ever call it is to achieve some side effect.
In consequence, often the usage of those functions is impure.

def display(number: Int): Unit // Is Impure
```

https://docs.scala-lang.org/overviews/scala-book/pure-functions.html

Pure Function

```scala
def square(num: Int): Int = num * num
def cube(num: Int): Int = square(num) * num

def getSqrCub(number: Int): String = {
    s"$number \t ${square(number)}\t ${cube(number)}"
}

getSqrCub(3) //3            9       27
```

Higher-order functions


I'VE GOT FUNCTIONS INSIDE FUNCTIONS...
... INSIDE FUNCTIONS.
imgflip.com

```scala
def HOF_sqr_cub(num: Int, fn:Int => Int): Int = fn(num)

def square(num: Int): Int = num * num
def cube(num: Int): Int = square(num) * num

def getSqrCub(number: Int): String = {
    s"$number \t ${HOF_sqr_cub(number, square)}\t ${HOF_sqr_cub(number, cube)}"
}

getSqrCub(3) //3            9       27
```

```scala
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {
val schema = if (ssl) "https://" else "http://"
(endpoint: String, query: String) => s"$schema$domainName/$endpoint?$query"
}

val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName)
val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query) // "https://www.example.com/users?id=1": String
```

Don't Iterate using `LOOP`, Insted Use `HOF`:

Map

```
val salaries = Seq(20000, 70000, 40000)
val doubleSalary = (x: Int) => x * 2
val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)
```

FoldLeft

```
val l = List(1, 3, 5, 11, -1, -3, -5)
l.foldLeft(0)(_ + _) // 11: Int
```

HOF in Options:

Default Value:

```
option match {
    case Some(i) => i
    case None => default
}

option.getOrElse(default) // This is HOF
```

Checking If Empty:

```
option match {
    case Some(a) => false
    case None => true
}

option.isEmpty // This is HOF
```

Ref: https://alvinalexander.com/scala/how-use-higher-order-functions-option-some-none-match-expressions/

Avoid mutability

```
use immutable data
```

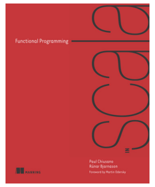
Mutation (bad!):

```
var numbers = Array(2000, 7000, 4000)

println("Number before Update")
for ( x <- numbers ) {
    println( x )
}
numbers(1) = 3000; // This is BAD
println("Number After Update")
for ( x <- numbers ) {
    println( x )
}

Number before Update
2000 7000 4000
Number After Update
2000 3000 4000
```

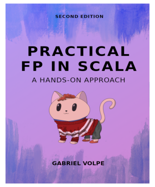
No mutation (good!):

```
val numbers = Seq(20000, 70000, 40000)
val new_numbers = numbers.updated(1,30000)
new_numbers //List(20000, 30000, 40000)
numbers //List(20000, 70000, 40000)
```

Resources



Functional Programming in Scala



Practical FP in Scala A hands-on approach

Zainab Sessions

QUESTIONS