ADVANCED FUNCTIONAL THINKING

DR SURBHI SARASWAT
SCHOOL OF COMPUTER SCIENCE

TOPICS

- **Functors**
- **Monads**

OPTION

- The pure functions return container(context) values like Option, Future, Some, etc.
  - This provides elegant ways of handling exceptions in function without any side effects or special handling for different inputs.
  - For example, if we have a function where we return input 1 divided by input 2, special handling is needed when input 2 is zero. Instead, if the function return Option no special handling is required.
- The ***Option*** in Scala is referred to as a carrier of a single or no element for a stated type.
  - When a method returns a value which can even be null then Option is utilized i.e, the method defined returns an instance of an Option, in place of returning a single object or a null.

OPTION

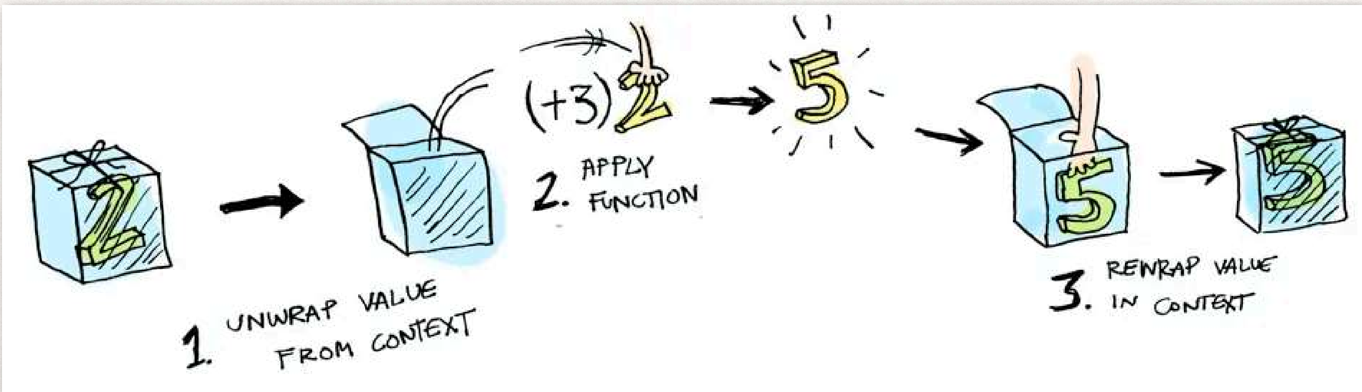
- Important points :
  - The instance of an Option that is returned here can be an instance of *Some* class or *None* class in Scala, where *Some* and *None* are the children of *Option* class.
  - When the value of a given key is obtained then *Some* class is generated.
  - When the value of a given key is not obtained then *None* class is generated.

```scala
// Creating object
object option
{
    // Main method
    def main(args: Array[String])
    {

        // Creating a Map
        val name = Map("Nidhi" -> "author",
                       "Geeta" -> "coder")

        // Accessing keys of the map
        val x = name.get("Nidhi")
        val y = name.get("Rahul")

        // Displays Some if the key is
        // found else None
        println(x)
        println(y)
    }
}
```

```
Some(author)
None
```

FUNCTORS

- Functors is any context that has a map function. It does the following
  - The actual value is wrapped in context(container) like Option , Future etc.
  - There is map method that unwraps the value from context.
  - It calls another method to transform the value. This another method is the method that map takes as input.
  - The transformed value is wrapped in same context.



FUNCTORS

- The Context that has map function that supports this transformation can be Functor. For context to be Functor, its map method needs to obey the laws listed below.
  - Identity Law: if map is called on a context with identity function, we get back the context.
  - Associative Law: f and g be functions we want to apply on the value in context. The following equality should hold
    - calling map with f and then map with g is same as calling map with g composed with f (i.e g(f(x)))
- Examples of Functors in Scala
  - List
  - Option
  - Some
  - Seq

FUNCTORS

- It defines how a map will be applied to data.
- Syntax: map :(A=>B) => F[A]=>F[B]

```
List(1, 2, 3).map(_ + 1)
// res0: List[Int] = List(2, 3, 4)
```

We specify the function to apply, and the map ensures it is applied to every item.

The values change but the structure of the list remains the same.

```
Vector(1, 2, 3).map(_.toString)
```

```
val x: Option[Int] = Some(1)
val y: Int = 2
val m: Int = 2

val z = x.map(a => (a+y) * m)
//or with the help of associative law
val z = x
  .map(_ + y)
  .map(_ * m)
```

MONADS

- In Scala the data types that implements *map* as well as ***flatMap()*** like Options, Lists, etc. are called as *Monads*.
- Monads are functors that also support unit() and flatMap() methods.
    - **unit() :** It is like void in Java, it does not returns any data types.
    - **flatMap() :** It is similar to the map() in Scala but it returns a series in place of returning a single component.
- Informally, a monad is a container of elements, notated as `F[_]`, packed with 2 functions: `flatMap` (to transform this container) and `unit` (to create this container).

MONADS

```
// Creating list of numbers
        val list1 = List(1, 2, 3, 4)
        val list2 = List(5, 6, 7, 8)

        // Applying 'flatMap' and 'map'
        val z = list1 flatMap { q => list2 map {
                r => q + r
        }
        }

        // Displays output
        println(z)
```

```
List(6, 7, 8, 9, 7, 8, 9, 10, 8, 9, 10, 11, 9, 10, 11, 12)
```

MONAD LAWS

- These functions must satisfy three laws:

1. *Associativity*: (m flatMap f) flatMap g = m flatMap (x => f(x) flatMap g)
   That is, if the sequence is unchanged you may apply the terms in any order. Thus, applying m to f, and then applying the result to g will yield the same result as applying f to g, and then applying m to that result.

2. *Left unit*: unit(x) flatMap f == f(x)
   That is, the unit monad of x flat-mapped across f is equivalent to applying f to x.

3. *Right unit*: m flatMap unit == m
   This is an 'identity': any monad flat-mapped against unit will return a monad equivalent to itself.

MONAD LAWS

Example:

```
val m = List(1, 2, 3)
def unit(x: Int): List[Int] = List(x)
def f(x: Int): List[Int] = List(x * x)
def g(x: Int): List[Int] = List(x * x * x)
val x = 1
```

1. *Associativity*:

```
(m flatMap f).flatMap(g) == m.flatMap(x => f(x) flatMap g) //Boolean = true
//Left side:
List(1, 4, 9).flatMap(g) // List(1, 64, 729)
//Right side:
 m.flatMap(x => (x * x) * (x * x) * (x * x)) //List(1, 64, 729)
```

2. *Left unit*

```
unit(x).flatMap(x => f(x)) == f(x)
List(1).flatMap(x => x * x) == 1 * 1
```

3. *Right unit*

```
//m flatMap unit == m
m.flatMap(unit) == m
List(1, 2, 3).flatMap(x => List(x)) == List(1,2,3) //Boolean = true
```