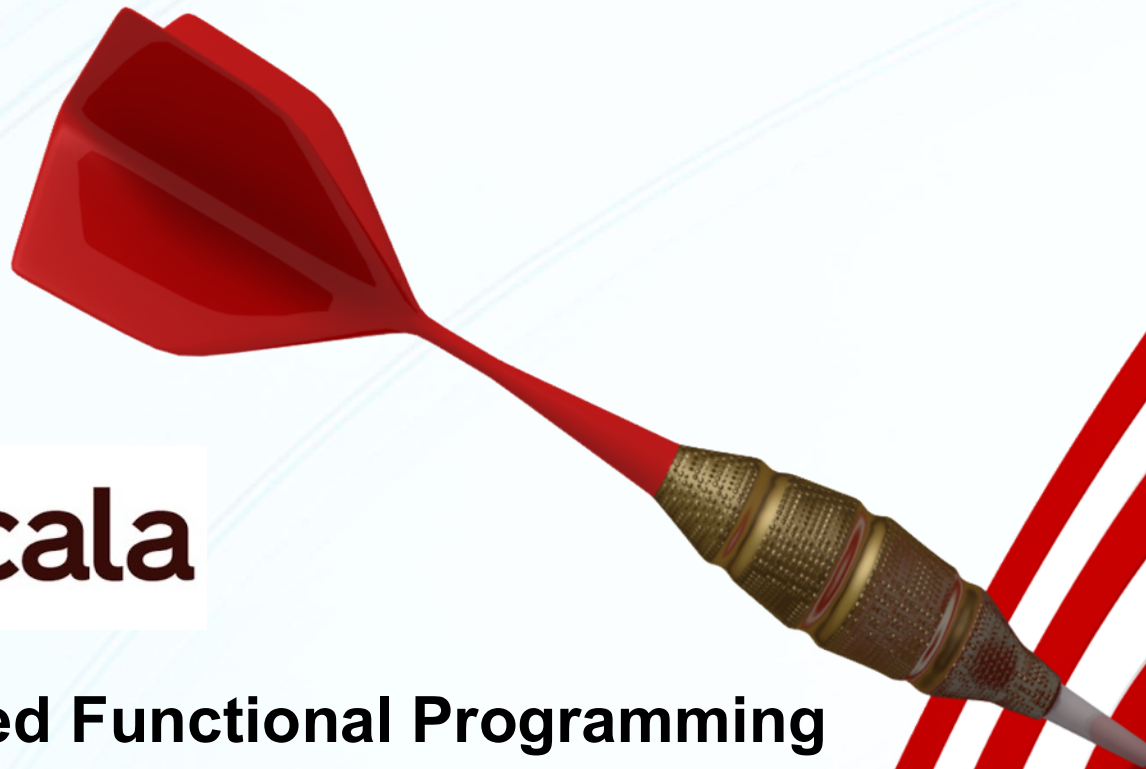




Advanced Functional Programming





ADVANCED FUNCTIONAL PROGRAMMING



Partial Functions



Partial Functions

A Partial function is a function is not able to produce a return for every single input data given to it.

- Partial function can determine an output for a subset of some practicable inputs only. Partial function can only be applied partially to the stated inputs.
- It is a Trait, which needs two methods namely **isDefinedAt** and **apply** to be implemented.



Partial Function Example 1

```
val r = new PartialFunction[Int, Int]
{
  def isDefinedAt(q: Int) = q > 0
  def apply(q: Int) = 12 * q
}
val input = -10
if (r.isDefinedAt(input)) println(r(input))
else println(r(0 - input))
```



Methods to define Partial functions

- There are some methods to define Partial function, which includes:
 - *case statements*
 - *collect method*
 - *andThen*
 - *orElse*



Partial Function using Case statements

```
val squareRoot: PartialFunction[Double, Double] = {  
  case x if x >= 0 => Math.sqrt(x)  
}  
  
val i = 36  
if (squareRoot.isDefinedAt(i)) println(squareRoot(i))  
else println(s"Value not defined at $i")
```

Here, Partial function is created using case statement. So, **apply** and **isDefinedAt** is not required here.



Partial function using orElse

```
val M: PartialFunction[Int, Int] =  
{  
  case x if (x % 5) == 0 => x * 5  
}  
  
val m: PartialFunction[Int, Int] =  
{  
  case y if (y % 2) == 0 => y * 2  
}  
  
val r = M orElse m  
  
println(r(5))  
println(r(4))
```

orElse method is helpful in chaining Partial functions together.



Partial function using collect

```
val M: PartialFunction[Int, Int] =  
{  
    case x if (x % 5) != 0 => x * 5  
}  
val y = List(7, 15, 9) collect { M }  
println(y)
```

Here, *Collect* will apply Partial function to all the elements of the List and will return a new List on the basis of the conditions stated.



Partial function using andThen

```
val M: PartialFunction[Int, Int] =  
{  
    case x if (x % 4) != 0 => x * 4  
}  
val append = (x: Int) => x * 10  
  
val y = M andThen append  
  
println(y(7))
```

Here, *andThen* will append the output of Partial function with the another function given and then will return that value.



Partially applied functions



Partially applied functions

- In functional programming languages, a call to a function that has parameters can also be stated as applying the function to the parameters. When a function is called with all the required parameters, it has fully applied the function to all of the parameters.
- **But when only a subset of the parameters to the function is passed, the result of the expression is a [Partially Applied Function](#).**
- Scala does not throw an exception when you provide fewer arguments to function, it simply applies them and returns a new function with rest of arguments which need to be passed.



Partially applied functions

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



Partially applied functions

- Partially applied functions are easily confused to be the same thing as Currying in Scala.
- Currying extends the concept of partially applying the function to the next level. Currying is the process of decomposing a function that takes multiple arguments into a sequence of functions, each with a single argument.



Partially applied functions

```
val divide = (num: Double, den: Double) => {  
    num / den  
}  
  
val curriedDivide: (Double) => (Double) => Double = divide.curried  
val curriedHalfOf: (Double) => Double = curriedDivide(_)(2)  
  
val x = curriedHalfOf(100)  
println(x)
```



Monads



Monods

In Scala, **Monad** is an object that wraps another object

Monads are structures that represent sequential computations.

A Monad is not a class or a trait; it is a concept.



Monods

```
val l1 = List(1,2)
val l2 = List(3,4)

// monad
val l3 = l1.flatMap{ x => l2.map{ y => x + y } }
```



THANK YOU

