Co-funded by the
Erasmus+ Programme
of the European Union

Previously on TORUS

Scala - episode 1

- Definition of variable
- Statically typed system
- How to print
- Expression
- Functions (or methods)
- Recusivity

Scala - episode 2

- A word about collections (underlying the vector collection)
- Functions on collections (sum, map, foreach, filter, find, count, foldLeft, foldRight, reduceLeft, reduceRight, union)

(Re)Introduction to type / Expression / Value

- Scala is a typed language (strong and static)
- Scala is object oriented and functional

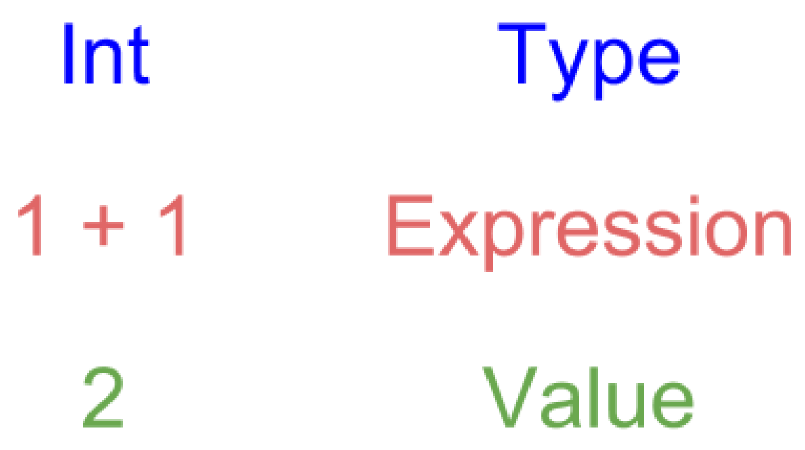
What is a type ?

A type defines a set of values a variable can possess.

It also defines a set of usable functions on these values

In Scala everything has a type (even function)

The “big” picture



Type coherence is on compile time (when writing code)
Values are at run-time (when a code is running)

Example

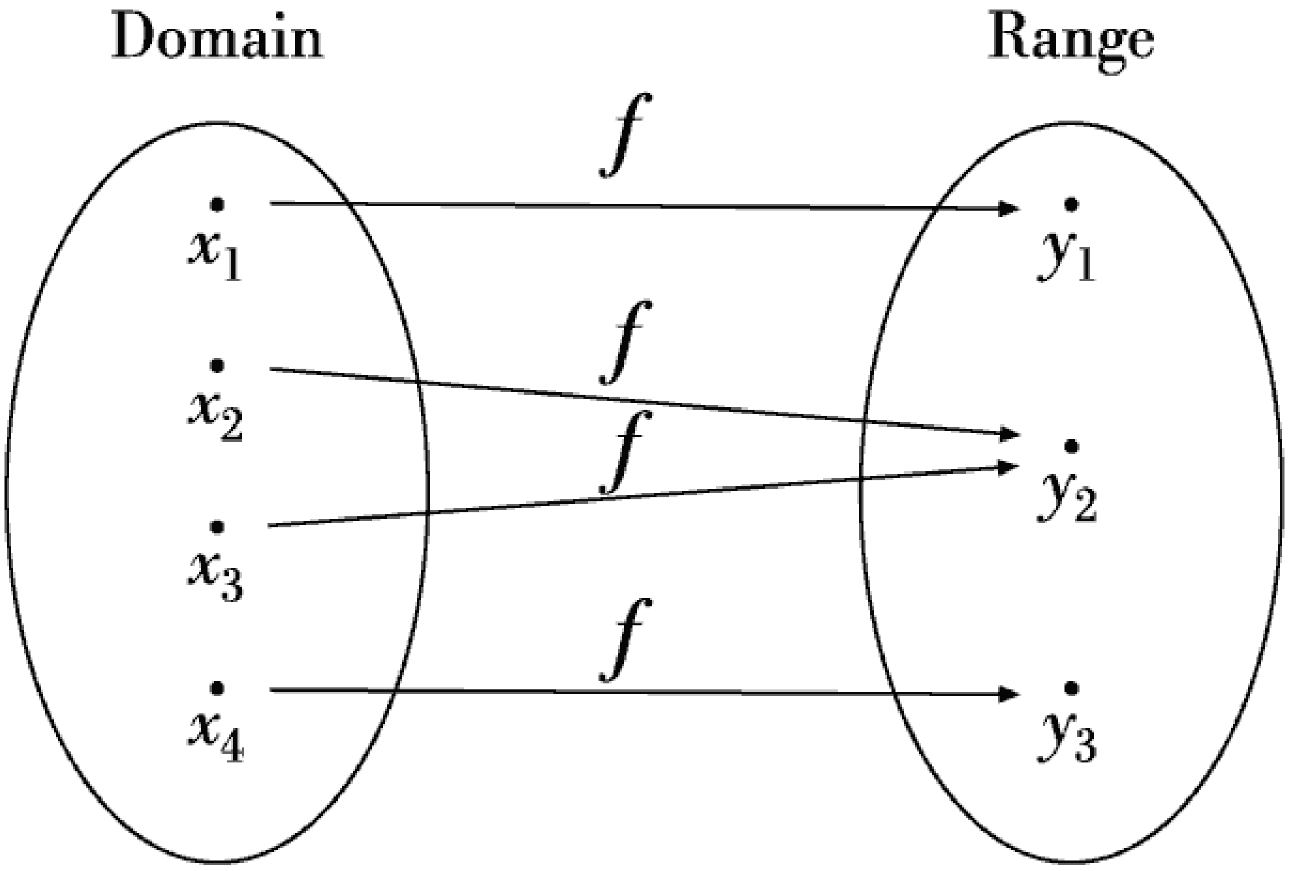
```
// Int is a type dedicated to integer
private val a : Int = 5
// With this type you can directly use
```

```
//%   +   >    >>>             isInstanceOf   toDouble   toLong     unary_+   |
//&   -   >=   ^               toByte         toFloat    toShort    unary_-
//*   /   >>   asInstanceOf    toChar         toInt      toString   unary_~
// All this function takes an Int and return something
a + 4
```

```
res3_1: Int = 9
```

But what is a function ?

A **function** is a processing that takes its *input* from a **domain** and produces a *result* in a **range** where *domain* and *range* are types



```
def f(a : Int) : Int = 2 * a
def g(a : Int) : Double = a.toDouble / 2
// These are two valid functions
```

Classes

A class is composed of **fields** and **methods**. Typically a class represents the formal description of a type. This means that a type describes the global shape (with fields) and also the behaviour (with methods).

For example, let’s create a new (traditional) type : the Rational type ($\mathbb{Q}$).

A rational is composed of two parts : numerator (p), denominator(q) $\Rightarrow \frac{p}{q}$
This can be define as :

```
class Rational (p : Int, q : Int)
```

But this definition does not provide, neither some behavior on rational, nor something useful, as the short example below shows. Furthermore, if you want to create an instance (this means a value) of this type, you have to use the keyword `new`

```
class Rational1 (p : Int, q : Int) //This is the definition of a new type (class)

// Here we will create new instances of Rational1 (this means new value)
private val r1 = new Rational1(1,3)
private val r2 = new Rational1(1,3)

// Lets use them
"Equality : " + (r1 == r2)
//r1 + r2 this can not compile !!!
//r1.p this does not compile, because you do not have direct access to the numerator, neither the denominator
```

```
defined class Rational1
res0_3: java.lang.String = "Equality : false"
```

The equality is not possible because most of oriented languages test the equality on reference in memory. The addition is not possible because we have not define the behavior of addition.

In scala, there is a keyword `case class` that transform the behavior of a class. So the definition

```
case class Rational2(p :Int, q : Int)
```

provides many things :

- equality on value rather than references
- a way to access to fields (numerator, denominator)
- a “pretty print” behavior
- no need to use the keyword new anymore.

Let’s look the changes :

```
case class Rational2(p: Int, q:Int)

private val r1 = Rational2(1,3)
private val r2 = Rational2(1,3)

"Equality : " + (r1 == r2)
"Fields access : " + r1.p
//r1 + r2 this can still not compile as we do not provide the behavior
```

```
defined class Rational2
res10_3: java.lang.String = "Equality : true"
res10_4: java.lang.String = "Fields access : 1"
```

Behavior

Now we have a type with a correct behavior, let's introduce behavior on our class.

A behavior on a class is defined using functions.

```
case class Rational3(p : Int, q : Int) {
  // We are defining a function reciprocal. The behaviour of this function is to create the reciprocal number
  // As reciprocal is define inside the class Rational3, this function is only avalaible for value of the type
  // Rational3. This means that you need a Rational3 value to call this function
  def reciprocal : Rational3 = Rational3(q,p)
}

private val r1 = Rational3(1,3)
r1.reciprocal // We are calling reciprocal on the value r1
```

```
defined class Rational3
res11_2: $user.Rational3 = [33mRational3(3, 1)
```

Let's go deeper in behavior

```
case class Rational4(numerator : Int, denominator : Int) {

  def add(that: Rational4) =
    new Rational4(that.numerator * denominator + numerator * that.denominator,
      denominator * that.denominator)
}

private val r1=Rational4(3,9)
r1
r1.numerator
r1.add(r1) // Now we can add two Rational
r1 add r1 // As add is a one parameter function we can use infix notation
```

```
defined class Rational4
res12_2: $user.Rational4 = [33mRational4(3, 9)
res12_3: Int = 3
res12_4: $user.Rational4 = [33mRational4(54, 81)
res12_5: $user.Rational4 = [33mRational4(54, 81)
```

Let's fix some problems :

- Function add is not an usual habit, usually we use +
- The rational has a bad display (54/81) could be simplified (or displayed) with (2/3)
- We miss all classical operations
- User can enter 0 as denominator (which should never happens)

```
import scala.annotation.tailrec

// Using denominator : Int = 1, is a way to define default value, if user does not provide some
case class Rational(numerator : Int, denominator : Int = 1) {
  require(denominator != 0, "denominator must be nonzero")

  // val or def
  private val gcd = {
    @tailrec
    def gcdRec(x: Int, y: Int): Int = {
      if(y == 0) x else gcdRec(y, x % y)
    }
    gcdRec(numerator, denominator)
  }

  // Here we use the simplify representation, but the number does not change
  val numer = numerator / gcd
  val denom = denominator  / gcd

  // The special keyword unary_ allow to use the - sign as a unary operator like -a
  def unary_- = new Rational(-numer, denom)

  def +(that: Rational) =
    Rational(that.numer * denom + numer * that.denom,
      denom * that.denom)
  def -(that: Rational) = this + -that
  def *(that: Rational) = Rational(numer * that.numer, denom * that.denom)
  def /(that: Rational) = Rational(numer * that.denom, denom * that.numer)
  def <(that: Rational) = numer * that.denom < that.numer * denom
  def >(that: Rational) = !(this < that)
  def max(that: Rational) = if (this < that) that else this
  // We have to define this equality because 1/3 == 2/6
  def ==(that: Rational) = that.numer == numer && that.denom == denom



    // If you want to make addition possible with Int, you should define :
    def + (a : Int): Rational = this + Rational(a)

    override def toString = numer + "/" + denom

}
```

```
import scala.annotation.tailrec
defined class Rational
```

Now we can play with Rational

```
// Now we have define our rational type we can play with it. It lacks of some features like converting to real,
// Enter your code here
private val integer = Rational(3)
private val half = Rational(1,2)
private val third = Rational(3,9)

private val result = integer + half
// This definition throws an exception (which will fail at runtime) ...
//val zero = Rational(5,0)
```

```
println("Let's play with rational")
println("Rational(3) : " + integer)
println("Rational(1,2) : " + half)
println("Rational(3,9) : " + third)
println("Rational(3) + Rational(1,2) : " + result)
private val toto = (half * (result + 3) ) / -third
println("(half * (result + 3) ) / -third : " + toto)
```

```
Let's play with rational
Rational(3) : 3/1
Rational(1,2) : 1/2
Rational(3,9) : 1/3
Rational(3) + Rational(1,2) : 7/2
(half * (result + 3) ) / -third : -39/4
```

Algebraic type : A way to describe your model into code.

The description can be modeled with **logical and** and **logical or.** The code should follows the structure of your model.

> *Linus Torvalds says :*
> *"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."*

Two patterns :

- product types (**and**)
- sum types (**or**)

Product type : `A` *has* a `B` *and* a `C`

To write a **product type** we use:

```
    case class A(b: B, c : C)
```

For example :

A coordinate has a latitude, a longitude and a projection

A projection is a value in (for example) : WGS84, Mercator, UTM
So let's define a type for the projection, here we will use an enumeration

```
object Projection extends Enumeration {
  type Projection = Value
    val WGS84, Mercator, UTM = Value
}
```

```
defined object Projection
```

Then we will define the coordinate.*

As a reminder a coordinate is a product type composed of three fields (latitude, longitude, projection).

So :

```
import Projection._



case class Coordinate(latitude : Double, longitude : Double, projection : Projection) {
    // Maybe here we can define the transformation to change from one projection to another one
    def toMercator = {
        projection match {
            case Projection.WGS84 => this.WGS84ToMercator
            case Projection.Mercator => this
            case Projection.UTM => this.UTMToMercator
        }
    }
    // Here are the helper to convert from one projection to another one
    // Dummy implementation, just for test.
    private def WGS84ToMercator = {
        Coordinate(latitude, longitude, Projection.Mercator)
    }
    private def UTMToMercator = {
        Coordinate(latitude, longitude, Projection.UTM)
    }
}

val hanoi = Coordinate(21.033, 105.85, Projection.WGS84)
val ferrare = Coordinate(44.83333, 11.61667, Projection.WGS84)
val nextWorkshop = Coordinate(14.01972, 100.535, Projection.WGS84)
val finalWorkshop = Coordinate(43.3017, -0.3686, Projection.WGS84)
val pau = Coordinate(43.3017, -0.3686, Projection.WGS84)
```

```
import Projection._
defined class Coordinate
hanoi: $user.Coordinate = [33mCoordinate(21.033, 105.85, WGS84)
ferrare: $user.Coordinate = [33mCoordinate(44.83333, 11.61667, WGS84)
nextWorkshop: $user.Coordinate = [33mCoordinate(14.01972, 100.535, WGS84)
finalWorkshop: $user.Coordinate = [33mCoordinate(43.3017, -0.3686, WGS84)
pau: $user.Coordinate = [33mCoordinate(43.3017, -0.3686, WGS84)
```

```
pau.toMercator
```

Sum type : `A` *is* a `B` *or* a `C`

To write a **sum type** we use:

```
    sealed trait A
    case class B() extends A
    case class C() extends A
```

A plot has a coordinate **and** may have a Value **or** not

So :

```
sealed trait Data[+A] {
    def hasValue : Boolean = this match {
        case Value(x : A) => false
        case NoValue => true
    }
    def get : A
    def getOrElse[A](default : A) : A = this match {
        case Value(x : A) => x
        case NoValue => default
    }
}

case class Value[A](value : A) extends Data[A] {
    override def get : A = value
}

case object NoValue extends Data[Nothing] {
    def get = throw new NoSuchElementException("MyNone.get")
}



case class Plot(coord : Coordinate, value : Data[Double])
```

```
defined trait Data
defined class Value
defined object NoValue
defined class Plot
```

```
private val l1 = List(Plot(pau, Value(3.14)), Plot(pau, Value(42)),Plot(pau, NoValue))
```

Option : you should never use `null` value.

Charles Antony Richard Hoare :

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*
>
> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*

In scala, there is a type that consider having or not a Value (like we did previously) name `Option` composed of two types :

- `Some` : means there is a value
- `None` : means there is no value

Of course `Option` type is more complete than the one we defined before !

Now your turn !

You have to create a data type for representing a person. A person has a firstName, a lastName and probably an age. A person can be a normal human or super hero. And a super hero have a nickName, eventually a “human” identity. A human is just a normal person ! A super hero has super power. Super power can be what you want !