STEPPING INTO SCALA

Angelo Corsaro, PhD

CTO, ADLINK Tech. Inc.

Co-Chair, OMG DDS-SIG

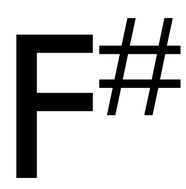
angelo.corsaro@adlinktech.com

Type Matters

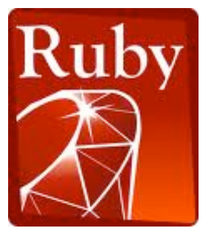
THROUGH DECADES OF TYPE SYSTEM'S DEBATE

The debate around type Programming Languages Type Systems has animated computer scientist over several decades and is far from being resolved

Statically Typed (ex.)



Dynamically Typed (ex.)



ARE YOU A TYPE-PHOBIC TYPE?

Proponents of Dynamically Typed Programming Languages advocate against strong typing in favour of reduced verbosity and added flexibility

Yet... Are they so sure that a type system always gets between you and what you are trying to achieve?

STRONGLY TYPED SYSTEMS

Well designed type-systems don't add **unnecessary verbosity** to your application since use sophisticated **inference** to derive types

Allow the **detection** of many **errors at compile time**, thus improving productivity, code quality and reducing the potential for run-time errors

Enable the generation of **more efficient code**

My Manifesto

Modern Statically typed programming languages are very valuable for building **scalable**, **safe**, and **efficient** applications

Stepping into Scala

WHAT IS SCALA

Scala (pronounced Skah-lah) stands for "**Sca**lable **la**nguage"

It is a language that carefully and creatively blends Object Oriented and Functional constructs into a statically typed language with sophisticated type inference

Scala targets the JVM and .NET runtime and is 100% compatible with Java

WHY SHOULD YOU CARE?

Scala is **simple to write**, **extremely compact** and easy to understand

Scala is **strongly typed** with a **structural type system**

Scala uses a sophisticated **type inference** to deduce types whenever possible

Scala is an **extensible** language (many construct are build in the standard library)

Scala makes it easy to design **Domain Specific Language (DSL)**

COMPLEX NUMBERS

To explore some of the nice features of Scala, let's see how we might design a Complex number class

What we expect to be able to do is all mathematical operations between complex numbers as well as scalar multiplications and division

```
[(1+i2)+2*(3-i5)*(i4)]/(1+i3)
~(1+i2) [Conjugate]
!(3+i4) [Modulo]
```

CONSTRUCTOR

Scala allows to implicitly create constructors and attributes starting from a simple argument list associated with the class declaration


```scala
class Complex(val re: Float, val im: Float)
```

...IN JAVA


```java
public class Complex {

   private final float re;
   private final float im;

   public Complex(float re, float im) {
      this.re = re;
      this.im = im;
   }

   public float re() { return re;}

   public float im() { return im;}
```

METHODS

Everything in Scala is a method even operators
Methods name can be symbols such as *, !, +,
etc.


```
def + (c: Complex) : Complex = Complex(re + c.re, im + c.im)
```

or, taking advantage of type inference....

```
def + (c: Complex) = Complex(re + c.re, im + c.im)
```

...IN JAVA


```
public Complex add(Complex that) {
     return new Complex(this.re() + that.re(),
                        this.im() + that.im());
   }
```

AS A RESULT...

Scala

```
val result = Complex(1,2) + Complex(2,3)
```

Java

```
Complex c1 = new Complex(1, 2);
Complex c2 = new Complex (3,4);
Complex c3 = c1.add(c2);
```

or...

```
Complex c3 = (new Complex(1, 2).add(new Complex (3,4));
```

NEGATION AND SCALAR MULTIPLICATION

In order to design a Complex class that is well integrated in our type system we should be able to support the following cases:

```
-(a+ib)
c*(a+ib)
(a+ib)*c
```

How can we supporting something like:

```
-(a+ib) and c*(a+ib)?
```

SCALA UNARY OPERATORS

Scala allows to define unary operators for the following method identifiers +, -, !, ~

```
def unary_-() = Complex(-re, -im)

def unary_!() = Math.sqrt(re*re + im*im)

def unary_~() = Complex(re, -im)
```

as a result we can write:

```
val result = -Complex(1,2) + ~Complex(2,3)
```

IMPLICIT CONVERSIONS

The expression:

```
val c3 = 3*Complex(5, 7)
```

Is equivalent to:

```
val c3 = 3.*(Complex(5, 7))
```

Yet, the method to multiply a Integer to a Complex is not present in the Scala Int class

What can we do to make the trick?

SCALA IMPLICIT CONVERSIONS

Scala does not support Open Classes, thus allowing to add new methods to existing classes

Yet Scala supports implicit conversions that can be used to achieve the same result

Lets see how...

SCALA IMPLICIT CONVERSION

```
object Complex {
   implicit def floatToReComplex (f: Float) = new ReComplex(f)
   implicit def intToReComplex(i : Int) = new ReComplex(i)
}


class ReComplex(re: Float) {

   def * (that: Complex) = Complex(re*that.re,re*that.im)

}
```

THE RESULT IS...

```
val c3 = 3*Complex(5, 7)
```

is converted automatically into:

```
val c3 = ReComplex(3).*(Complex(5, 7))
```

PUTTING IT ALL TOGETHER

```
class Complex(val re: Float, val im: Float) {

    // Binary Operators
    def + (c: Complex) = Complex(re + c.re, im + c.im)
    def - (c: Complex) = Complex(re - c.re, im - c.im)

    def * (f: Float) = Complex(f*re, f*im)
    def * (c: Complex) = Complex((re*c.re) - (im*c.im),
                                 ((re*c.im) + (im*c.re)))

    def / (c: Complex) = {
        val d = c.re*c.re + c.im*c.im
        Complex(((re*c.re) + (im + c.im))/d,
                ((im*c.re) - (re*c.im))/d )
    }

    // Unary Operators
    def unary_-() = Complex(-re, -im)
    def unary_!() = Math.sqrt(re*re + im*im)
    def unary_~() = Complex(re, -im)


    // Formatting
    override def toString() : String  = {
        if (im > 0) re + "+i" + im
        else if (im < 0) re + "-i" + (-im)
        else re.toString

    }
}
```

Pattern Matching

PATTERN MATCHING

Scala supports a **match** statement that allows to perform pattern matching — in other terms decompose object in their composing parts

PATTERN MATCHING & COMPLEX NUMBERS

```
c match {
    case Complex(0,0)  => println(“Zero”)
    case Complex(0, _) => println(“Pure Imaginary Number”)
    case Complex(_, 0) => println(“Pure Real Number”)
}
```

Functions.Closures.Currying

FUNCTIONS

Scala has first-class functions
Functions can be defined and called, but equally functions can be defined as unnamed literals and passed as values

```
def inc(x: Int) = x + 1
inc(5)
```

```
val vinc = (x: Int) => x+1
vinc(5)
```

Notice once again the uniform access principle

PLAYING WITH FUNCTIONS

```
val list = List(1,2,3,4,5,6,7,8,9)
val g5 = list.filter((x: Int) => x > 5)
g5: List[Int] = List(6, 7, 8, 9)
```

Or with placeholder syntax

```
val list = List(1,2,3,4,5,6,7,8,9)
val g5 = list.filter(_ > 5)
g5: List[Int] = List(6, 7, 8, 9)
```

CLOSURES

Scala allows you to define functions which include **free variables** meaning variables whose value is not bound to the parameter list

Free variable are resolved at runtime considering the closure of visible variable.

Example:

```
def mean(e : Array[Float]) : Float = {
      var sum = 0.0F
      e.foreach((x: Int) => sum += x)
      sum/e.length
   }
```

```
def mean(e : Array[Float]) : Float = {
      var sum = 0.0F
      e.foreach(sum += _)
      sum/e.length
   }
```

CURRYING

Scala provides support for curried functions which are applied to multiple argument lists, instead of just one

Currying is the mechanism Scala provides for partial application and introducing new control abstraction

```
def curriedSum(x: Int)(y: Int) = x + y

curriedSum(1) {
 3 +5
}
```

Traits

TRAITS

Scala supports single inheritance from classes but can mix-in multiple traits

A trait is the unit of code reuse for Scala. It encapsulate methods and field definitions

Traits usually expect a class to implement an abstract method, which constitutes the “narrow” interface that allows to implement a rich behaviour

ORDERED COMPLEX NUMBERS

Our complex numbers are not comparable

Let’s assume that we wanted to make them comparable, and lets supposed that we define the total order as based on the module of the complex number

How can we implement this behaviour?

ORDERED TRAIT

The **`Ordered[T]`** traits encapsulates the set of methods that allow to define a total ordering over a type

All the behaviour is defined in terms of an abstract method, namely “compare”

Classes that mix-in this trait have to implement the “compare” method

```
class Complex(val re: Float, val im: Float) extends Ordering[Complex] {
def compare(x: Complex, y: Complex) = {
      if (x == y) 0
      else if (!x > !y) 1
      else -1
   }
}
```

Type Parametrisation

TYPE PARAMETRISATION

Scala provides support for type parametrization and makes it available for both classes as well as traits

Scala allows to annotate the parametrized type to control the resulting type variance

```
trait Queue[T] {
  def head: T
  def tail: Queue[T]
  def append(x: T) : Queue[T]
}
```

TYPE VARIANCE

If S <: T is Queue[S] <: Queue[T]?

By **default** Scala makes generic types **nonvariant**.

This behaviour can be changed using the following annotations:

Queue[+T] indicates that the the sub-typing is **covariant** in the parameter T

Queue[-T] indicates that the the sub-typing is **contravariant** in the parameter T

“Option” and More..

OPTION[T]

The Scala library provides an implementation of the Option Monad (like Maybe in Haskell) that allows to model **optional values** safely

Notice that in Java optional values are often represented by using the **null** reference

EXAMPLE

```
let sf: Option[Foo] = map.get(key)
// return Some(x) or None depending
// on the result of the lookup

let ob: Option[Bar] = sf.map(someFunc(_))
// invoking map on None simply returns None,
otherwise assuming that someOperation returns a
Bar than Some(bar) will be returned
```

EXAMPLE

```
// For comprehension (like haskell do):
for {
  f <- map.get(k);
  b <- someFunc(f)
} yield b
```

WHO IS USING SCALA?

Scala has been growing fast and accelerating in the past few years.