SCALA Tutorial

Two faces of the same "language"

SCALA (**SCA**lable **LA**nguage) is a multi-paradigm programming language that mix together object-oriented and functional style.
Object-oriented side of SCALA is given by the entity representation: each of the entity is represented as a particular object model by classes (same as C# or Java) and traits (the equivalent of interface in C# and Java).
The ***functional side*** is visible looking the possible value that a variable can assume: functions are values, as *integers* or *strings*.
Program in scala is a mix of these two possible programming patterns: you can use typical features of functional paradigm such as pattern matching, currying, fold-r, fold-l, map etc. mixed by imperative and object oriented paradigm.
In effect, there are other languages that offers these functionality such as F# that inherit CAM-L functional notation mixed by .NET object oriented framework.
SCALA paradigms are mixed up together to extends Java code-design. Using SCALA is possible to overload operators, describe methods using optional parameters without overload them, extend exception management using non checked exceptions, call function using named parameters etc.
Another useful feature of SCALA is it's ***portability***. SCALA is full-integrated with Java, in fact, you can import SCALA libraries in your Java source-code and use them programming directly in Java and viceversa. This feature is guaranteed to the fact that SCALA compiler transforms a SCALA source code in Java byte-code that can be executed on among any Java Virtual Machines.
SCALA is ***statically typed***: compilers return you an error if there are some Types-mismatch.

Scoping rule

Names in Scala identify types, values, methods, and entities.
Names are introduced by local definitions and declarations, inheritance, import clauses or package clauses (also called bindings).
Bindings of different kinds have a precedence:

1) Definitions and declarations that are local, inherited, or made available by a package clause in the same compilation unit where the definition occurs have highest precedence.
2) Explicit imports have next highest precedence.
3) Wildcard imports have next highest precedence.
4) Definitions made available by a package clause not in the compilation unit where the definition occurs have lowest precedence.

Let's see, in this example, an application of the binding precedence:

```
package P {            // 'X' bound by package clause
 import Console._      // 'println' bound by wildcard import
 object A {
    println("L4: "+X)          // 'X' refers to 'P.X' here
  object B {
   import Q._         // 'X' bound by wildcard import
```

```
    println("L7: "+X)      // 'X' refers to 'Q.X' here
  import X._       // 'x' and 'y' bound by wildcard import
    println("L8: "+x)      // 'x' refers to 'Q.X.x' here
  object C {
   val x = 3       // 'x' bound by local definition
       println("L12: "+x)  // 'x' refers to constant '3' here
       { import Q.X._      // 'x' bound by wildcard import
...
```

A binding has a scope in which the entity defined by a single name can be accessed using a simple name. Scopes are nested. A binding in some inner scope shadows bindings of lower precedence in the same scope as well as bindings of the same or lower precedence in outer scopes.
Types and terms are bonded into two different namespace: for this reason you can assign the same name to a type and a term in the same environment (class, package etc..)

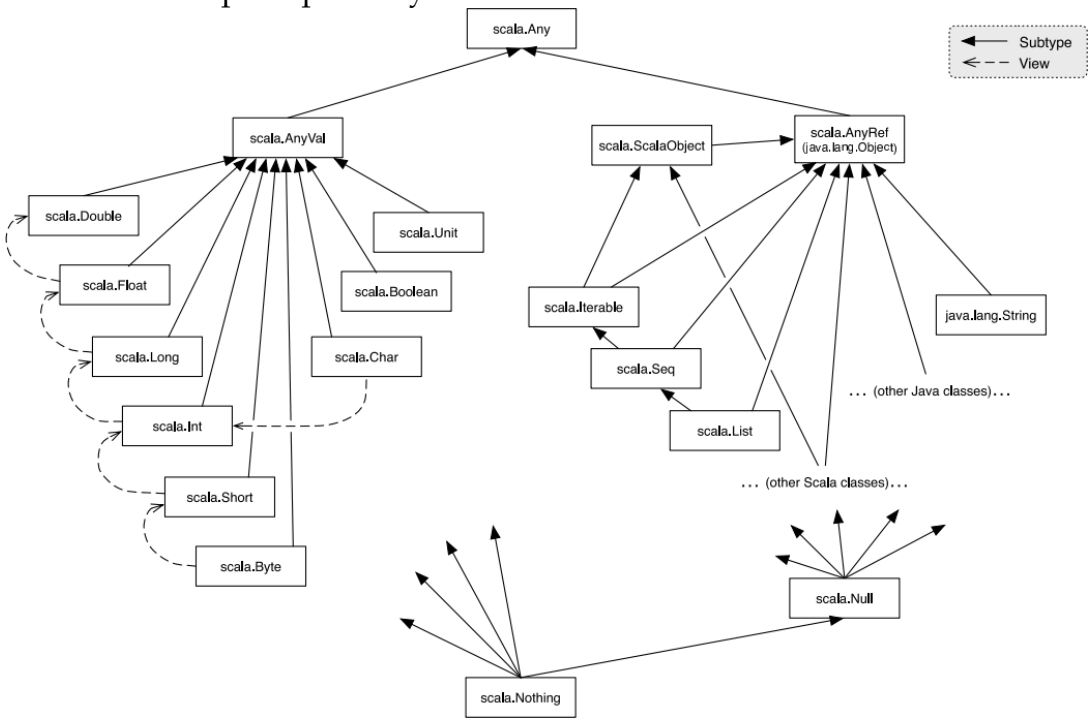
Parameters and Values passing methods

The standard SCALA passing method is ***by value,*** also for ***scala.AnyValue*** and ***scala.AnyRef***.
When a method or function is called it's possible to use the explicit parameter passing by name.
The objects "by reference" passing method is simulated (like pointers in C) using the object reference as value.
SCALA does not support the outer parameters (this is possible in C#) to pass a parameter that is forced to be setted in the method body.

Scala Classes

Hierarchies

Let's see the SCALA classes hierarchies: this scheme has been taken by SCALA official documentation and explain perfectly the inheritance of SCALA classes.



The principal superclass (the equivalent of object in Java) is ***scala.Any***, that have two subclasses ***scala.AnyVal*** and ***scala.AnyRef***.
The class ***scala.AnyVal*** has a fixed number of subclasses, which describe values are not implemented as objects. This is another difference from Java where "everything is an object".
Everything inherit from ***scala.AnyVal*** can be passed either by value or reference.
Different is for ***scala.AnyRef*** subclass that represent, instead, everything is represented ad object in the underlying system, whatever it is. For instance, if SCALA running on the Java Runtime Environment, this class corresponds to the java.lang.Object class.
User-defined types are reference types and are, directly or undirectly, subclasses of scala.AnyRef.
There are two differents type of "null-value" in SCALA: scala.Null which is used to assign null vale to a scala.AnyRef object; also values are nullable using scala.Nullable object.
Scala classes are a static item. Classes can be declared as in this example.

```
class MyClass
{
    // local variables
    var _myValue : Int

    // method declaration
    def setValue(v : Int)
    {
        _myValue = v
    }
}
```

It' possible to declare generic classes and polymorphic methods, even if SCALA language it is statically typed, like in this example.

```
class MyGenericClass[T]
{
    // local polymorphic variable
    var _myValue: T

    // polymorphic method
    def setValue(v: T)
    {
        _myValue = v
    }
}
```

SCALA Traits

Scala Traits are used to define what in Java is an interface types that have methods with a specific signature. Different from Java or C#, traits can be partially implemented using local definitions.
Let's consider the following example

```
trait MyInterface
{
    def Count(): Int
    def IsEmpty(): Boolean = (Count()!=0)
}
```

Note that a class that extend this trait is forced to implements the unimplemented methods, but can use or override already implemented methods.