Functional Programming: Classes, Objects and Traits

This document provides the basics of classes, objects and traits. Even though there are mutable constructs used in Scala, in this document only the immutable features of the language will be presented. This means that programming is a sequence of transformations rather than a changing state. The examples proposed here are simple and can be tested in a scala worksheet.

1. Classes

Scala class definition is much smaller than the definition in Java. Here an example of an animal class:

```
class Animal(val name: String, val height: Int) {
  def >(other: Animal) = this.height > other.height

  override def toString(): String = "name: " + name + " ; height:" + height;

}
```

Classes in Scala are parameterized with constructor arguments. The two constructor arguments, `name` and `height`, are both public, due to the val modifier. Without any modifier, the attributes would be private. Notice that immutable attributes being public is not a problem in object orientation because their values cannot be changed. This means there are NO getters and certainly NO setters.
Member functions are public by default. To create a private member function, a private keyword must be added to its definition.
The class has two public immutable attributes: `name` and `height`. It defines a method “>”, (this is a possible method name in Scala) which compares the animals’ heights. It also overrides the method `toString`, which returns a String value of the class. The name `this` denotes the instance of the class on which the current method is executed.
To test this class, we can use:

```
val dog = new Animal("dog", 50)
val bird = new Animal("bird", 30)
val dogName = dog.name
dog > bird
```

Notice that the attributes are immutable, so their values cannot be changed. The instruction:

```
dog.name = "newDog"
```

will yield a “Reassignment to val” error.

2. Abstract classes and subclasses

The animal class is an abstract concept, so we will refactor it into an abstract class.

```
abstract class Animal(val name: String) {
  val height: Int

  def >(other: Animal) = this.height > other.height

  override def toString(): String = "name: " + name + " ; height:" + height;

}
```

This class has an abstract value height, so this class is not instantiable. To instantiate, concrete subtypes must be created:

```
class Dog(val height: Int) extends Animal("dog")
class Bird(val height: Int) extends Animal("bird")
val bird = new Bird(15)
val dog = new Dog(50)
val animal : Animal = new Dog(20)
dog > bird
```

Like Java, there can be a reference of the Animal type, which points to a Dog object instantiation.

3. Objects

Singleton Objects

Methods and values that are not associated with individual instances of a class can belong in singleton objects. This characteristic is achieved by using the keyword `object` instead of `class`. A singleton object definition is similar to a class definition. Here is an example of xml handling in scala. The function `values` is supposed to return a list with all the texts contained in elements with a certain `label`.

```
// include xml in scala:
// https://github.com/scala/scala-xml/wiki/Getting-started
object utilsXML {
  def values(xml: scala.xml.Elem, label: String): List[String] = {
    val l = for (e <- (xml \\ label)) yield e.text
    l.toList
  }
}
```

To call the `values` function, we define an xml object and provide it to the function. The definition of an xml object in scala can be done using the xml syntax directly :

```
val xml = <div class="content"><p>Hello</p><p>world</p></div>

utilsXML.values(xml,"p")  // List(Hello, world)
```

Companion Objects

In Java a class with both instance methods and static methods are often needed. In Scala, you achieve this by having a class and a “companion” object. A singleton object with the same name as the class is called that class’s **companion object**. The class and its companion object have to be defined in the same source file. The class is called the **companion class** of the singleton object. A class and its companion object can access each other’s private members.

The apply method

It is common to have objects with an apply method that is called for expressions of the form `Object(arg1, ..., argN)`. Typically, these apply methods return an object of the companion class. There can be several apply methods. For instance, the following definition

```
class Point(val x: Int, val y: Int) {
  override def toString() = "( "+x+" ; "+y+" )"
}

object Point {

  def apply(x: Int, y: Int): Point = {
    new Point(x,y)
  }
}
```

allows the instances to be more easily created (`pt2`, as opposed to `pt1`) and to eventually validate values:

```
val pt1 = new Point(1,2)
val pt2 = Point(1,2)
```

To eventually validate values, a private constructor is needed to ensure objects are only created through the companion object. This is achieved using the private keyword between the class name and its parameters. In the example, only positive coordinates are allowed:

```
class PositivePoint private (val x: Int, val y: Int) {
  override def toString() = "( "+x+" ; "+y+" )"
}

object PositivePoint {

  def apply(x: Int, y: Int): PositivePoint = {
    new PositivePoint(if (x<0) 0 else x,if (y<0) 0 else x)
  }
}

val ppt1 = PositivePoint(10, -2) // PositivePoint = ( 10 ; 0 )
val ppt2 = PositivePoint(-10, -2) // PositivePoint = ( 0 ; 0 )
```

Exercises

1. Make a Student class, with attributes `firstName` and `lastName` of type String and `age` of type Int. The primary constructor is to consider all these variables as its parameters, but negative ages have to be turned to 0. Provide an auxiliary constructor that accepts a name such as “Ana Júlia Silva” and sets the members `firstName` and `lastName`.

2. Write metric unit conversion set of classes. The classes represent all the metric values from milimeters to kilometers. The sum of two metric values gives the correct result in meters. The use should be as follows.

```
val v1 = new MiliMeters(10)   // v1: MiliMeters = 10.0 mm
val v2 = new KiloMeters(2)    // v2: KiloMeters = 2.0 km
val r = v1+v2                 // r: Distance = 2000.01 m
```

Suggestion: start with a Distance abstract class which holds a value, a conversion factor and the unit name. Both the conversion factor and the name should be defined by each subclass. Use the following to get started:

```
abstract class Distance(val factor: Double, val unit: String) {
  val value : Double
  private def toMeters : Double = ???
  def +(other: Distance): Distance = ???
  override def toString() = ???
}
```

4. Case classes

A case class definition has the same syntax as any other class, but with the keyword `case` preceding it. Case classes are often used with pattern matching mechanisms. Example:

```
case class Person(name: String, age: Int)
```

A Person instance can be then created without the new keyword, just as if a Person companion object had been defined:

```
Person("Ana", 20)
```

Another characteristic of case classes is that equality structural comparisons are implicitly defined. Considered the definition of Person as a case class, observe the following comparisons:

```
Person("Ana", 20) == Person("Jose", 30)
//> res3: Boolean = false
Person("Ana", 20)== Person("Ana", 20)
//> res4: Boolean = true
Point(1,1)==Point(1,1)  //not a case class, but with apply method in company object
//> res5: Boolean = false
```

Case classes allow complex pattern matching on objects with simplicity. A single `case` keyword has to be added to each class that will be pattern matchable. When complex behavior is needed and only simple pattern matching mechanisms are to be applied, ordinary classes can be used instead.

A summary of case classes characteristics:

- All the parameters are implicitly val.
- `toString`, `equals`, `hashCode`, and `copy` are generated but can be supplied;
- `apply` and `unapply` are also generated. `apply` allows the omission of the word new when objects are created. `unapply` lets the use of objects in pattern matching.

Here an example from the book "Programming in Scala":

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

The domain of interest is arithmetic expressions consisting of variables, numbers, and unary and binary operations. This is expressed by the hierarchy of Scala classes. Five classes are used: the abstract base class `Expr` with four subclasses, one for each type of expression. Their bodies are empty.

A function `simplify` can be defined:

```
def simplify(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) => e
 case UnOp("-", Number(n)) if (n<0)  => Number(-n)
  case BinOp("*", e, Number(1)) => e
  case BinOp("+", e, Number(0)) => e
  case _ => expr
}
```

And then used:

```
simplify(UnOp("-", UnOp("-", Var("x")))) // Var(x)

simplify(UnOp("-", Number(-1))) // Number(1)

simplify(BinOp("+", Number(3.5), Number(0))) // Number(3.5)

simplify(BinOp("*", Number(3.5), Number(1))) // Number(3.5)
```

Patterns are tried in the order in which they were written, just like the switch case statement in Java. In the second example there is a pattern guard, which filters only negative numbers. Pattern guards can associate a condition to the pattern matching and validate only if both the pattern and the condition hold.

Exercises

1. Consider the following definitions:

```
abstract class IntTree
case object EmptyTree extends IntTree
case class Node(elem: Int, left: IntTree, right: IntTree) extends IntTree
```

Complete the implementation of the functions `contains` and `insert`:

```
def contains(t: IntTree, v: Int): Boolean = t match { ??? }

def insert(t: IntTree, v: Int): IntTree = t match { ??? }
```

to allow these results:

```
val tr1 = EmptyTree
val tr2 = insert(tr1, 5)
assert(tr2==Node(5,EmptyTree,EmptyTree))
val tr3 = insert(tr2, 3)
assert(tr3==Node(5,Node(3,EmptyTree,EmptyTree),EmptyTree))
val tr4 = insert(tr3, 7)
assert(tr4==Node(5,Node(3,EmptyTree,EmptyTree),Node(7,EmptyTree,EmptyTree)))
val tr5 = insert(tr4, 4)
assert(tr5==Node(5,Node(3,EmptyTree,Node(4,EmptyTree,EmptyTree)),
                  Node(7,EmptyTree,EmptyTree)))
assert(contains(tr4,4)==false)
assert(contains(tr5,4)==true)
assert(contains(tr5,7)==true)
```

2. Consider the following definitions:

```
abstract class Item

case class Article(description: String, price: Double) extends Item

//Bundle: a number of things that are sold together with a discount
case class Bundle(description: String, discount: Double, items: List[Item]) extends Item

//Multiple: a number of things that are sold together without a discount
case class Multiple(count: Int, item: Item) extends Item
```

Complete the implementation of the function `price`:

```
def price(it: Item): Double = it match { ??? }
```

to ensure these results:

```
val a1 = Article("a1",2.5)
val a2 = Article("a2",10.0)
val b = Bundle("b1",0.2,List(a1,a2))
val m = Multiple(3, b)
assert(price(a1)==2.5)
assert(price(a2)==10.0)
assert(price(b)==10.0)
assert(price(m)==30.0)
```

5. Traits

Traits can be seen as combination of Java's interfaces and Ruby's mixins. Full method definitions are allowed. They are similar to Java 8's interfaces. Classes and objects can extend traits but traits cannot be instantiated and therefore have no parameters.

Traits are a fundamental for code reuse in Scala. They encapsulate method and field definitions, which can then be reused by mixing them into classes. A class must inherit from just one superclass, but a class can mix in any number of traits.

Here is an example:

```
trait Similarity [A] {
  def isSimilar(e: A): Boolean
  def isNotSimilar(e: A): Boolean = !isSimilar(e)
}
```

There two methods `isSimilar` (abstract method) and `isNotSimilar` with a concrete implementation. Consequently, classes that integrate this trait only need a concrete implementation for `isSimilar`. Notice that this is a generic trait with type A defined as generic. Thus, the `Point` class, if using the Similarity trait, needs to provide an implementation of the method `isSimilar`:

```
class Point(val x: Int, val y: Int) extends Similarity[Point] {
  override def toString() = "( "+x+" ; "+y+" )"
  override def isSimilar(e: Point) = x == e.x && y == e.y
}
```

Here is some tests:

```
val p1 = Point(2, 3)
val p2 = Point(2, 4)
val p3 = Point(2, 3)
assert(p1.isSimilar(p2)==false)
assert(p1.isSimilar(p3)==true)
assert(p3.isSimilar(p2)==false)
```

Here is another trait:

```
trait Equality [A] {
  def isEqual(e: A): Boolean
  def isNotEqual(e: A): Boolean = !isEqual(e)
}
```

Class Point can consider both traits:

```
class Point(val x: Int, val y: Int)
                    extends Similarity[Point] with Equality[Point] {

  override def toString() = "( "+x+" ; "+y+" )"
  override def isSimilar(e: Point) = x == e.x || y == e.y
  override def isEqual(e: Point)   = x == e.x && y == e.y
}
```

Notice that the "diamond problem" of multiple inheritance is resolved with the following rule: if two traits provides the implementation of the same method, the one on the right in the class definition is considered.

Consider the following guidelines for using traits:

- If the behavior will not be reused, use a concrete class;
- If it might be reused in multiple, unrelated classes, make it a trait as only traits can be mixed into different parts of the class hierarchy;
- If you want to inherit from it in Java code, use an abstract class.

The Ordered trait

The Ordered trait is used by providing a single `compare` method. The Ordered trait then defines <, >, <=, and >= in terms of this method. == is not covered by this trait.

The Ordered trait requires the specification of a generic type parameter when used. When you mix in Ordered, you must actually mix in `Ordered[C]`, where `C` is the generic class whose elements are to be compared. The method `compare` should compare the receiver, `this`, with the object passed as an argument to the method. It should return an integer that is zero if the objects are the same, negative if receiver is less than the argument, and positive if the receiver is greater than the argument.

Exercises

1. Define a class `OrderdedPoint`, subclass of the `Point` class with the `Ordered` trait. If a point `a` has a coordinate x inferior than the same coordinate of another point `b`, `a<b`. If these coordinates have the same value, the coordinate y is to be considered.

```
class OrderedPoint(x: Int,y: Int)
                    extends Point(x, y) with Ordered[OrderedPoint] {
  def compare(that: OrderedPoint): Int = { ??? }
}
```

Create the `compare` function to obtain these results:

```
val op1 = new OrderedPoint(1, 1)
val op2 = new OrderedPoint(1, -1)
val op3 = new OrderedPoint(2, 1)
val op4 = new OrderedPoint(1, 1)
assert( (op1 < op2) == false)
assert(op1 > op2)
assert((op1 >= op3) == false)
assert(op1 >= op4)
assert(op4 >= op1)
```