UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

Introduction to Scala

Antonio Caliò[1]

[1]DIMES Dept., University of Calabria
Rende (CS), IT
a.calio@unical.it
Big Data Analytics - Computer Engineering for the IoT

Outline

Presentation agenda

What is Scala

- Scala stands for: *scalable language*.
- It combines object-oriented programming with functional programming
- A Scala program runs on a JVM.
- Scala can execute Java Code – thus, Scala understands Java code
- As an object-oriented language, we can define classes and class hierarchies via the inheritance
- As a functional language Scala provides lightweight syntax for defining *anonymous functions* – like Java lambdas – it also allows the definition of nested functions
- Scala is *statically* typed like many other programming languages (e.g. C, Pascal, Rust). Nonetheless, it does not need the programmer to specify the type of a variable the type information of a variable (most of the times)

Setting up your environment

1. You must have a working Java environment – I am sure you already have one on your machine!
2. Install Scala[1]. You can either:
   - Download the installer from Scala-Lang
   - If you are on Linux chances are that it is available within you repository
3. Download and install the Scala main build-system: SBT

[1] It is recommended to install the 2.12 version

You First Scala Program

It the installation is successful you should be able to run Scala in console mode.

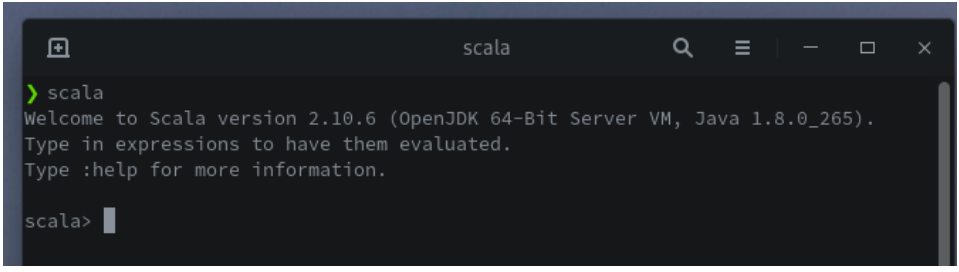

Figure: Scala Console

Then print the usual `Hello World` string.

```
println("Hello World")
```

Scala in Script Mode

- Create a file: `HelloWorld.scala`

```
object HelloWorld {
  def main(args: Array[String]) = {
     println("Hello World")
  }
}
```

- Compile with `scalac HelloWorld.scala`
- Run with `scala HelloWorld`

Presentation agenda

Basic Concepts

- Scala is Case Sensitive
- Class names should be in upper case
- Method names should start with lower letters
- Program file names should match the name of the object
- Every Scala program must have a main function

Scala Identifiers

Every Scala component requires a name.
Names are used for objects, classes, variables and methods. There are four types of identifier:

- Alphanumeric identifier, e.g., `age`, `salary`, `_age1`, `__1_value`
- Operator identifier, e.g., `+`, `++`, `:::`, `=>`. The scala compiler will mangle this operators by turning them into a correspondent Java identifier
- Mixed identifier. It is an alphanumeric identifier followed by an underscore and an operator identifier. For instance: `var_+`, `var_=`.
- Literal identifier. It is an arbitrary string enclosed within a pair of `.

Scala Packages

- In order to declare a package, in the first non-comment line in the file you should write:

  `package name.of.the.package`
- We can import the entire scope of a package as follows:

  `import scala.xml._`

  the underscore is equivalent to * in Java
- A single class is imported as follows:

  `import scala.collection.mutable.HashMap`
- You can also import multiple class as follow:

```
import scala.collection.immutable.{TreeMap, TreeSet}
```

Scala Type Dynamic

- A `Dynamic` is a marker trait/interface that enables dynamic invocations. Therefore is a variable `x` is an instance of an object adhering to the `Dynamic` interface.
- There are four different types of dynamics:
  1. `selectDynamic` it allows to write field accessors: `x.foo`
  2. `updateDynamic` it allows to write field update: `x.foo = 5`
  3. `applyDynamic` it allows to call methods with arguments: `x.bar(0)`
  4. `applyDynamicNamed` it allows to call methods with named arguments: `x.bar(y=8)`
- In order to define a class adhering to this specifications you just need to extends the `Dynamic` interface:

```
import scala.language.dynamics
class MyClass extends Dynamic{
}
```

Select Dynamic

```
class DynImpl extends Dynamic {
  def selectDynamic(name: String) = name
}
```

Try:

```
val d = new DynImpl()
d.foo
d.selectDynamic("foo")
```

Update Dynamic

```
class DynImpl extends Dynamic {

  var map = Map.empty[String, Any]

  def selectDynamic(name: String) =
    map get name getOrElse sys.error("method not found")

  def updateDynamic(name: String)(value: Any) {
    map += name -> value
  }
}
```

Try:

```
val d = new DynImpl
d.foo
d.foo = 10
```

Apply Dynamic

```
class DynImpl extends Dynamic {
  def applyDynamic(name: String)(args: Any*) =
    s"method '$name' called with arguments ${args.mkString("'", "', '", "'")}"
}
```

Try:

```
val d = new DynIml
d.ints(1,2,3)
```

Apply Named Dynamic

```
class DynImpl extends Dynamic {
  def applyDynamicNamed(name: String)(args: (Strin,Any)*) =
    s"method '$name' called with arguments ${args.mkString("'", "', '", "'")}"
}
```

Try:

```
val d = new DynIml
d.ints(i1=1,i2=2,i3=3)
```

Presentation agenda

val vs var

A variable can be defined as:

- a value, with the `val` keyword. These are constants
- a variable, with the `var` keyword. These are mutable

Declaring a variable

- The syntax to declare a new variable is the following:

```
val myVar: String = "Foo"
```

  - It is the syntax rule:

```
[val|var] <variableName> {: <dataType>} = <Initial Value>
```

- Scala has also a mechanism for type inference, so you do not need to specify the type of the variable

```
val myVal = "Hello"
var myVar = 4
```

Example Program

```
object Demo {
   def main(args: Array[String]) {
      var myVar :Int = 10
      val myVal :String = "Hello Scala with datatype declaration."
      var myVar1 = 20
      val myVal1 = "Hello Scala new without datatype declaration."

      println(myVar)
      println(myVal)
      println(myVar1)
      println(myVal1)
   }
}
```

Variable Scope

Three possible scopes:

- Field - a field is a variable defined within the scope of an object. It is accessible both from inside the object and from inside the object.
- Method Parameter - it is always an immutable object, it is accessible only from inside the method it is passed to
- Local Variable - it accessible only from inside the method. Except for returned object

Presentation agenda

Basic Class

```
class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x);
      println ("Point y location : " + y);
   }
}

object Demo {
def main(args: Array[String]) = {
   val pt = new Point(10,20)
   pt.move(10, 10)
  }
}
```

Extending a class

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x);
      println ("Point y location : " + y);
   }
}

class Location(override val xc: Int, override val yc: Int,
   val zc :Int) extends Point(xc, yc){
   var z: Int = zc

   def move(dx: Int, dy: Int, dz: Int) {
      x = x + dx
      y = y + dy
      z = z + dz
      println ("Point x location : " + x);
      println ("Point y location : " + y);
      println ("Point z location : " + z);
   }
}
```

```
object Demo {
   def main(args: Array[String]) {
      val loc = new Location(10, 20, 15);

      // Move to a new location
      loc.move(10, 10, 5);
   }
}
```

Implicit Classes

Implicit classes are very useful as they allow implicit conversions with class's primary constructor when the class is in scope.
They can be declared as follows:

```
object <object name>{
  implicit class <class name> (<Variable>: Data type) {
    def <method>(): Unit = {}
  }
}
```

Implicit Class Example

Here is an example of an implicit class named `IntTimes`.
It has method that repeatedly print "Hello" to the screen

```
object Run {
   implicit class IntTimes(x: Int) {
      //f:=> means that f will be evaluated when it is accessed
      def times [A](f: =>A): Unit = {
 def loop(current: Int): Unit =
 if(current > 0){
    f
    loop(current - 1)
 }
 loop(x)
      }
   }

  def main(args: Array[String]) = {
    //4 is interpreted as a IntTimes object upon which we call the times methods passing the
    //function println
    4 times println("hello")
  }
}
```

Singleton Objects

- Scala is more object-oriented if compared to Java.
- In Scala a class cannot have static members
- We can define `object` as opposed to `class`. Objects works similarly to java static objects
- An `object` is a singleton
- An `object` has only the default constructor which is implicitly called when it gets created
- Usually objects are used to put the main method of the application

Singleton Object Example - Demo.scala

```
class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
   }
}

object Demo {
   def main(args: Array[String]) {
      val point = new Point(10, 20)
      printPoint

      def printPoint{
 println ("Point x location : " + point.x);
 println ("Point y location : " + point.y);
      }
   }
}
```

Presentation agenda

Private Members

It is visible only inside the class or object that contains the member definition.

```
class Outer {
  class Inner {
    private def f() { println("f") }
    class InnerMost {
      f() //ok
    }
  }
  (new Inner).f() // Error
}
```

- In the above example the first call to f() is legal because the class `InnerMost` is declared within the scope of `Inner`, thus it can see the method. Outside the `Inner` scope however, the function remains inaccessible.

Protected Members

A protected member is accessible from every subclass

```
package p {
   class Super {
      protected def f() { println("f") }
   }

   class Sub extends Super {
      f()
   }

   class Other {
      (new Super).f() // Error: f is not accessible
   }
}
```

Public Members

A member is public by default – unless we specify any of the previous keywords.
Public members can be accessed from anywhere in the code

```
class Outer {
   class Inner {
      def f() { println("f") }
       class InnerMost { f() } // OK
   }
   (new Inner).f() // OK because now f() is public
}
```

Scope of Protection

Access modifiers can be augmented. For instance we can declare a member with the syntax: `private[X]`. This means that the member is private "up to" X, where X can be a package, class or singleton object

```
package society {
   package professional {
      class Executive {
 private[professional] var workDetails = null
 private[society] var friends = null
 private[this] var secrets = null

 def help(another : Executive) {
    println(another.workDetails)
    println(another.secrets) //ERROR
 }
      }
   }
}
```

Presentation agenda

Operators

Nothing new here!

Presentation agenda

If-Else

Nothing new here!

Presentation agenda

Loops

There are three different kinds of loop statement:

1. `while loop`
2. `do-while loop`
3. `for-loop`

while loop - Nothing Weird

```
object Demo {
   def main(args: Array[String]) {
      // Local variable declaration:
      var a = 10;

      // while loop execution
      while( a < 20 ){
 println( "Value of a: " + a );
 a = a + 1;
      }
   }
}
```

do-while loop - Nothing Weird

```
object Demo {
   def main(args: Array[String]) {
      // Local variable declaration:
      var a = 10;

      // do loop execution
      do {
 println( "Value of a: " + a );
 a = a + 1;
      }
      while( a < 20 )
   }
}
```

for-loop

- For-loop leverages on the notion of range – very similar to Python.
- A range is ... a range! it has a start and an end.

```
for (i <-1 to 10){
   doSomething()
}
```

- the "<-" operator is called *generator*. It generates all the value between both ends in the range and it assigns them to the variable on the left side.

multiple ranges

You can also use multiple range within the same for-loop as follows:

```
for( a <- 1 to 3; b <- 1 to 3)
   println( "a,b: " + a + " " + b  );
```

It is the same as having two nested loop. Thus it is equivalent to having the following loop:

```
for( a <- 1 to 3)
  for(b <- 1 to 3)
   println( "a,b: " + a + " " + b  );
```

loop over collections

- You can iterate over a collection of object as follows:

```
for(var x <- someList){
  doSomething(x)
}
```

- You can also filter the objects within a collection while you are iterating them

```
for(var x <-someList if condition1; if condition2...){
  doSomething(x)
}
```

for-loop with yield

You can store values from a "for" loop in a variable or can return through a function. To do so, you prefix the body of the for expression with the keyword `yield`

```
object Demo {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      // for loop execution with a yield
      var retVal = for{ a <- numList if a != 3; if a < 8 }  yield a

      // Now print returned values using another loop.
      for( a <- retVal){
 println( "Value of a: " + a );
      }
   }
}
```

Presentation agenda

Function Declarations

- A function declaration can appear anywhere in the code
- Scala permits the definition of nested function, namely function inside other functions
- There is no particular difference between methods and function, these two words are often used interchangeably
- A function declaration has the following syntax:

```
def functionName([list of parameters]) : [return type]
```

Function definition

Let's imagine we want to define a function that sums two integers. This would be its definition:

```
object add {
  def addInt(a: Int, b: Int) : Int = {
     var sum:Int = a + b
     return sum
  }
}
```

In Scala a void returning function is declared as a `Unit` return function.

```
   def fName(a:Int, b:Int) : Unit = {...}
//or
   def fName(a:Int, b:Int) {...}
```

Call-by-Name

- A call-by-name mechanism passes a code block to the call and each time the call accesses the parameter, the code block is executed and the value is calculated.

```
object Demo {
   def main(args: Array[String]) {
delayed {
 //code block
  time()
};
   }

   def time() = {
      println("Getting time in nano seconds")
      System.nanoTime
   }
   // This function cantake
   def delayed( t: => Long ) = {
      println("In delayed method")
```

Default parameter values

As in many other languages you can define default values for function parameters

```
object Demo {
   def main(args: Array[String]) {
      println( "Returned Value : " + addInt() );
   }

   def addInt( a:Int = 5, b:Int = 7 ) : Int = {
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

Partially Applied Functions

- When you specify only a fraction of the parameters required by a function you have a partially applied function.
- The mechanism is simple, you bind some the parameters to some value, while you are only required to specify all the remaining parameters required by the function

```
import java.util.Date

object Demo {
   def main(args: Array[String]) {
      val date = new Date
      //the first argument is fixed. For the second we leave a placeholder _
      val logWithDateBound = log(date, _ : String)

      logWithDateBound("message1" )
      Thread.sleep(1000)

      logWithDateBound("message3" )
   }
```

Scala Functions with Named Arguments

As it happens for other languages like Python, you can call a function by naming each parameter.

```
object Demo {
   def main(args: Array[String]) {
      printInt(b = 5, a = 7);
   }

   def printInt( a:Int, b:Int ) = {
      println("Value of a : " + a );
      println("Value of b : " + b );
   }
}
```

Higher-order functions

- These are functions that takes other functions as parameter
- In the following example the `apply` function requires as input a function `f` that requires an Integer as input and it returns a String.
- The `apply` function returns whatever `f` is returning.

```
object Demo {
   def main(args: Array[String]) {
      println( apply( layout, 10) )
   }

   def apply(f: Int => String, v: Int) = f(v)

   // A is a sort of template parameter
   def layout[A](x: A) = "[" + x.toString() + "]"
}
```

Anonymous Functions

- Anonymous functions are also called *function literals*
- They are created and evaluated at runtime. The resulting objects are called *function values*
- The syntax to create a function literals is the following:

```
var inc = (x: Int) => x+1
```

- `inc` can be used as a regular function
- it is also possible to have function literals without input parameters

```
var userDir = () => {Syste.getProperty("user.dir")}
```

Currying Functions

- A currying function transforms a function that takes multiple parameter into a chain of functions having a single parameter.

```
def strcat(s1: String)(s1: String) = s1 + s2
//alternatively
def strcat(s1: String) = (s2:String) => s1 + s2
//call
strcat("foo")("bar")
```

Closure function

- A *closure* is a function whose return value depends on the value of one or more variables declared outside the function

```
object Demo {
   def main(args: Array[String]) {
      println( "multiplier(1) value = " +  multiplier(1) )
      println( "multiplier(2) value = " +  multiplier(2) )
   }
   var factor = 3
   val multiplier = (i:Int) => i * factor
}
```

Presentation agenda

Creating Format Strings

In Scala there is a `printf` and `format` method that print output with formatted numbers. These methods return a `PrintStream` oject

```
object Demo {
def main(args: Array[String]) {
   var floatVar = 12.456
   var intVar = 2000
   var stringVar = "Hello, Scala!"

   var fs = printf("The value of the float variable is " + "%f, while the

   println(fs)
}
}
```

String Interpolation

- With this feature we can embed variable references directly inside a string literal
- In order to use string interpolation you need to prefix the string with 's' as follows:

```
object Demo {
   def main(args: Array[String]) {
      val name = "James"

      println(s"Hello, $name")
      // the block within ${ } is interpreted
      println(s"1 + 1 = ${1 + 1}")
   }
}
```

Presentation agenda

Declaring Array Variables

- As it happens in Java, when you need to define an array you must specify the type along with the size of the array

```
var z: Array[String] = new Array[String](3)
//alternatively
var z = new Array[String](3)
```

- To access an array you use the following syntax:

```
z(0) =" Ciao"; z(1) = "Ciao"; z(3) = "Ciao"
```

Multi-Dimensional Arrays

A multi-dimensional array can be defined as follos:

```
var  myMatrix = ofDim[Int](3,3)
```

In order to access the multi-dimensional array you must provide an index for each different dimension. For instance:

```
myMatrix(i)(j)
```

Presentation agenda

Available Collections

- Scala has a very rich sets of collection library.
- A collection may be strict or lazy. A lazy collection have elements that may not consume memory until they are accesses.
- Collections can either be *mutable* or *immutable*
- The most commonly used type of collections are:
  1. Lists
  2. Sets
  3. Maps
  4. Tuples
  5. Options
  6. Iterators

Lists I

- Creating a list

```
val fruit: List[String] = List("apple", "orange", "caciocavallo")
//alternatively - Nil marks the end of the list
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil ))
//two dimensional list
val dim = (1 :: (0 :: (0 :: Nil))) ::
  (0 :: (1 :: (0 :: Nil))) ::
  (0 :: (0 :: (1 :: Nil))) :: Nil
```

- Concatenating Lists. You can do it with the List.concat() method or with List.:::() operator

```
object Demo {
  def main(args: Array[String]) {
     val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))
     val fruit2 = "mangoes" :: ("banana" :: Nil)

     // use two or more lists with ::: operator
     var fruit = fruit1 ::: fruit2
     println( "fruit1 ::: fruit2 : " + fruit )

     // use two lists with Set.:::() method
     fruit = fruit1.:::(fruit2)
     println( "fruit1.:::(fruit2) : " + fruit )
```

Lists II

```
      // pass two or more lists as arguments
      fruit = List.concat(fruit1, fruit2)
      println( "List.concat(fruit1, fruit2) : " + fruit  )
   }
}
```

- Uniform lists. An uniform lists is a list that contain the same element repeated multiple times

```
object Demo {
   def main(args: Array[String]) {
      val fruit = List.fill(3)("apples") // Repeats apples three times.
      println( "fruit : " + fruit  )

      val num = List.fill(10)(2)         // Repeats 2, 10 times.
      println( "num : " + num  )
   }
}
```

Sets I

- Creating a set

```
var s : Set[Int] = Set()
var s : Set[Int] = Set(1,2,3,4)
var s = Set(1,2,3,4)
```

- concatenating two sets

```
object Demo {
   def main(args: Array[String]) {
      val fruit1 = Set("apples", "oranges", "pears")
      val fruit2 = Set("mangoes", "banana")
      // use two or more sets with ++ as operator
      var fruit = fruit1 ++ fruit2
      println( "fruit1 ++ fruit2 : " + fruit )

      // use two sets with ++ as method
      fruit = fruit1.++(fruit2)
      println( "fruit1.++(fruit2) : " + fruit )
   }
}
```

- Finding max and min

Sets II

```
object Demo {
   def main(args: Array[String]) {
      val num = Set(5,6,9,20,30,45)

      // find min and max of the elements
      println( "Min element in Set(5,6,9,20,30,45) : " + num.min )
      println( "Max element in Set(5,6,9,20,30,45) : " + num.max )
   }
}
```

- Finding the intersection

```
object Demo {
   def main(args: Array[String]) {
      val num1 = Set(5,6,9,20,30,45)
      val num2 = Set(50,60,9,20,35,55)

      // find common elements between two sets
      println( "num1.&(num2) : " + num1.&(num2) )
      println( "num1.intersect(num2) : " + num1.intersect(num2) )
   }
}
```

Maps I

- Creating a map

```
// Empty hash table whose keys are strings and values are integers:
var A:Map[Char,Int] = Map()

// A map with keys and values.
val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")
```

- Adding new key/value element to a map

```
A + = ('I' -> 1) // I is the key, 1 is the value
```

- Basic operations on maps
  1. keys - returns and iterable containing the keys stored in the map
  2. values - returns an iterable with the values stores in the map
  3. isEmpty - returns true if the map is empty
- Concatenating maps

Maps II

```
object Demo {
   def main(args: Array[String]) {
      val colors1 = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")
      val colors2 = Map("blue" -> "#0033FF", "yellow" -> "#FFFF00", "red" -> "#FF0000")

      // use two or more Maps with ++ as operator
      var colors = colors1 ++ colors2
      println( "colors1 ++ colors2 : " + colors )

      // use two maps with ++ as method
      colors = colors1.++(colors2)
      println( "colors1.++(colors2)) : " + colors )
   }
}
```

- Iterating over a map

```
object Demo {
   def main(args: Array[String]) {
      val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF","peru" -> "#CD853F")

      colors.keys.foreach{ i =>
 print( "Key = " + i )
 println(" Value = " + colors(i) )}
   }
}
```

Tuples I

- A tuple combines a heterogeneous fixed number of items inside a variable. It can be created as follows:

```
val t =(1, "hello", 3)
```

- Accessing the elements of a tuple (by position):

```
t._1 // returns 1
t._2 // returns "hello"
t._2 // returns 3_
```

- iterate over a tuple

```
t.productIterator.foreach{ i => println("Value"+i)}
```

Options I

- A Scala Option is container that can contain zero or one element of a specific type. An option can be either be a `Some[T]` or `None` object.
- Options are very useful, and are frequently used by the Scala API. For instance, if you miss a key in a Map, Scala will return an empty Option

```
object Demo {
   def main(args: Array[String]) {
      val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

      println("show(capitals.get( \"Japan\")) : " + show(capitals.get( "Japan")) )
      println("show(capitals.get( \"India\")) : " + show(capitals.get( "India")) )
   }

   def show(x: Option[String]) = x match {
      case Some(s) => s
      case None => "?"
   }
}
```

- The getOrElse() method is a convenient way to express default value when using `Option`. Under the hood this method the method `isEmpty` upon the `Option` instance.

Options II

```
object Demo {
   def main(args: Array[String]) {
      val a:Option[Int] = Some(5)
      val b:Option[Int] = None

      println("a.getOrElse(0): " + a.getOrElse(0) )
      println("b.getOrElse(10): " + b.getOrElse(10) )
   }
}
```

Iterators I

- It is not a collection, but rather a way of to access a collection. It can be created as follows:

```
 object Demo {
  def main(args: Array[String]) {
     val it = Iterator("a", "number", "of", "words")

     while (it.hasNext){
println(it.next())
     }
  }
}
```

Presentation agenda

References I