



Introduction to Scala

An Overview of the Scala Language

Tim Dalton
St. Louis Java Users Group
Dec 13, 2007



What is Scala ?

- Scala is a programming language targeting the Java Virtual Machine that integrates features of both functional and object-oriented languages
- Compiles to Java byte code, but does have a scripting interpreter
- Statically typed
- Pure object orientation (no primitives)
- Has very flexible syntax (lots of sugar)



More About Scala

- Invented at the EPFL (Ecole Polytechnique Federale de Lausanne) in Switzerland primarily by Martin Odersky
- Odersky worked on Pizza language, Generic Java (GJ), and is the original author of “javac” compiler
 - GJ provided the basis for generics in Java 1.5
- First Released in 2003
- .NET version once orphaned, but now seems to be getting more attention



Hello World

- Hello World in Scala :

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

- Singleton **object** instead of **class**
 - No static methods or properties within **class**
 - All **object** methods equivalent of Java **static**
 - Singleton **object** and **class** of same name are called companions. A common idiom in Scala is **object** acting as a factory for its companion **class**.
- The **println** method is statically imported by default from **scala.Predef** object.
- Return type on main method is inferred (More about that later)
- **Array** is collection class with parameterized type (generic)



More Complex Example

```
package SwingDemo

import javax.swing.{JFrame, JLabel}

object Main extends Application {
  def getTitle() = "Scala can Swing"

  val frm = new JFrame(getTitle)
  frm.getContentPane.add(
    new JLabel("Hello World"))
  frm.setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE)
  frm.pack
  frm.setVisible true
}
```

```
package SwingDemo
```

```
import javax.swing.{JFrame, JLabel}
```

- Similar to Java
- The import is different
 - Multiple classes from package using curly braces, “{ }”
 - Uses “_” instead of “*” for wildcard
- Packages and imports can be scoped using curly braces
- Multiple objects and classes in single source file
 - No need to conform to naming or directory conventions



More Complex Example

```
object Main extends Application {
```

- `Application` object in standard library eliminates the need to explicitly implement the `main` method

```
def getTitle() = "Scala can Swing"
```

- Single expression code block does not need curly braces
- Trailing semi-colons are inferred, but sometimes are needed when there is ambiguity that the compiler can not resolve
- Return type for method is inferred to be `String`
 - To explicitly declare type:

```
def getTitle():String = "Scala can Swing"
```



More Complex Example

```
val frm = new JFrame(getTitle)
```

- Empty parenthesis on `getTitle` invocation optional
- The type for `frm` can be inferred
 - Explicitly declared:

```
val frm:JFrame = new JFrame(getTitle)
```

```
frm setVisible true // Look Mom no dots or parens!
```

- This is valid in Scala !
 - Parameter parenthesis and “dot” notation optional when expression in form:

```
<object> <method> <parameter(s)>
```

- Valuable for Domain Specific Language (DSL)



More About Type Inference

- Scala as a special type, `Unit`, that is the equivalent to Java's `void`
 - Technically, all Scala expressions return something even if it's `Unit`
- Type inference is being considered for future version of Java and is a feature of C# 3.0
- Compare Java to Scala declaration:

```
// Java
LinkedHashMap<String,ByteArrayInputStream>
streamMap =
new LinkedHashMap<String,ByteArrayInputStream> ();

// Scala
var streamMap =
new LinkedHashMap[String, ByteArrayInputStream]
```

Values and Variables

- Scala uses `var` to declare a mutable variable or property
 - Properties are declared in the context of object or class definition
 - Otherwise, it is the equivalent of local variable in Java
- Keyword, `val`, is used for immutable value or property
 - Equivalent to `final` in Java

- Scala is a pure object-oriented language and does not support primitives
- Package, `scala`, contains classes correlating to Java primitives:
 - Char
 - Byte
 - Short
 - Int
 - Long
 - Float
 - Double
 - Boolean
- When interacting the Java classes, conversions are implicitly performed



Notable Syntax Features

- Multiline strings using triple quotes:

```
val str = """This
is
a
multi-line string"""
```

- Inferred return value based on result of last expression in code block:

```
def NullSafeToUpper(s:String) = {
  println("in NullSafeToUpper")
  if (s == null) "NULL" else s.toUpperCase
}
```

- Return value is inferred to `String` since both results of `if` statement are strings
- Explicit `return` supported as well



Notable Syntax Features

- Non-alphanumeric characters in method names:

```
def #@%!& = println("Snoopy cursing")
```

- This can be used for a form of operator overloading:

```
class Foo(value:Int) {  
  def +(bar:Bar) = value + bar.value  
}
```

```
class Bar(val value:Int)
```

```
val foo = new Foo(13)
```

```
val bar = new Bar(23)
```

```
println(foo + bar) // outputs "36"
```

- Expression, `foo + bar`, same as `foo.+(bar)`

- Ability to define an method as a infix, prefix, or postfix operator
 - Method ending with colon (":") is right associative:

```
class Foo(val value:Int) {  
  def unary_! = "!!!" + value + "!!!"  
  def % = value + "%"   
  def *: (multiple:Int) = value * multiple  
}
```

...

```
var foo = new Foo(62)
```

```
!foo // result: String = "!!!62!!!"
```

- Prefix operator. Same as `foo.unary_!`

```
foo % // result: String = "62%"
```

- Postfix operator. Same as `foo.%`

```
2 *: foo // result: Int = 124
```

- Infix and right associative. Same as `foo.*(2)`

Constructors

- Primary constructor defined as part of the class definition:

```
class Dog(val name:String)
```

```
var rinTinTin = new Dog("Rin Tin Tin")
```

```
rinTinTin.name // result: String = "Rin Tin Tin"
```

- Primary constructor parameters are made into object properties with access controlled by use of **val** or **var**
 - Modifier, **val**, indicates property has accessor method, but no mutator
 - Modifier, **var**, indicates property has accessor and mutator methods
 - Accessors and mutators are invoked when property is accessed or assigned like a field in Java
 - Accessors and mutators are public by default

```
class ImmutableFoo(val value:Int)
class MutableFoo(var value:Int)
```

```
val immutable = new ImmutableFoo(54)
val mutable = new MutableFoo(32)
```

- Accessors and mutator used just like a field:

```
println("Mutable value = " + mutable.value)
mutable.value = 39
println("Immutable value = " + immutable.value)
```

- This mutator is not implemented and will result in error:

```
immutable.value = 65
```

Constructors

- Scala can subclass based on a super class constructor:

```
class Mammal(name:String) {  
  override def toString() =  
    "Mammal " + name  
}  
class Dog(name:String) extends Mammal(name) {  
  def speak() = this.toString + " says Woof"  
}  
...  
var benji = new Dog("Benji")  
benji.speak()  
  // result String: "Mammal Benji says Woof"
```

- The `Dog` class extends the `Mammal` class and uses the `Mammal(name:String)` constructor to pass the name to the super class

- Allowing constructors and fields to be specified as part of the class definition itself allows simple classes to be defined using single lines of code:

```
class Mammal (name:String)
class Feline (name:String) extends Mammal (name)
class Lion (name:String) extends Feline (name)
```

- Secondary constructors are declared in methods named **this**:

```
class AbsoluteNumber(num:Int) {  
  var value = Math.abs(num)  
  
  def this(dbl:Double) = this(dbl.toInt)  
}
```

- The **this(dbl:Double)** constructor converts its parameter to **Int** and delegates to the primary constructor
- Parameters of secondary constructors do not become object properties

- Accessors and mutators can be explicitly implemented to create “virtual” properties:

```
class AbsoluteNumber(num:Int) {  
  private var _value = Math.abs(num)  
  def value = _value // "getter" method  
  def value_=(num:Int) = _value = Math.abs(num)  
                        // "setter" method  
}  
...  
var absolute = new AbsoluteNumber(10)  
printf("Absolute = {0}\n", absolute.value)  
absolute.value = -5
```

- This allows Scala properties to more easily conform to the *Uniform Access Principle*



Default “`apply`” method

- Methods named `apply` have special meaning for Scala objects and classes.
- Invoked by using a "method-less" expression in the form, `<object>.[(<parameters>)]`:

```
object Foo {
  def apply(n:Int) = printf("FooObject({0})\n",n)
}
class Foo {
  def apply(n:Int) =
printf("FooClass({0})\n",n) }
...
Foo(1)           // prints "FooObject(1) "
var foo = new Foo
foo(2)           // prints "FooClass(2) "
```

- Scala’s `Array` class uses `apply` as an indexer, hence array elements are accessed using parenthesis, not square brackets



Functional Programming in Scala

- Functional programming is getting more serious consideration outside of academia
- Pure functional languages have no global state other than the stack. “Stack as the state” has these advantages:
 - Concurrency is much easier because this is no shared state for processes to contend over
 - Testing is easier because functions should always return the same result for the same parameters
 - Debugging is aided by having the state in one place
- Scala is not a pure functional language, but it does borrow many features from popular functional languages.
 - With a little discipline by developers, benefits can be reaped.

- Functions in Scala are objects that can be passed like any other object

- Expressed in form, [(<parameters>)] => <code>

```
val multiply = (x:Int, y:Int) => x * y
val product = multiply(2,3) // result: Int = 6
```

- Functions that accept other functions as parameters are called *Higher Order Functions*:

```
def doFunc(x:Int, y:Int,
           func:(Int, Int) => Int) = func(x,y)
...
doFunc(2,3,multiply) // result: Int = 6
```

- Function objects referencing methods can be generated using special syntax:

```
def doFunc(x:Int, y:Int,  
          func:(Int, Int) => Int) = func(x,y)
```

```
def add(x:Int, y:Int) = x + y
```

```
...
```

```
doFunc(3,4, add _) // result: Int = 7
```

- Trailing underscore generates what is called a partially applied function which in this case delegates to the `add` method

- Function objects can be declared inline or anonymously:

```
def doFunc(x:Int, y:Int,  
          func:(Int, Int) => Int) = func(x,y)
```

...

```
doFunc(3,4, (x:Int, y:Int) => x * 2 + y))  
          // result: Int = 10
```

```
((x:Int, y:Int) => x * 3 + y * 2) (3,5)  
          // result: Int = 19
```

- Inline function objects like these are often referred to as *Lambda Expressions*

- Function objects without parameters are code blocks:

```
def timeBlock(block: =>Unit) {  
  val start = System.currentTimeMillis()  
  block  
  printf("Block took {0} milliseconds\n",  
        System.currentTimeMillis -  
start)  
}  
...  
timeBlock {  
  (1 to 4).foreach { x => println("x = " + x) }  
}
```

- The code block is passed to the `timeBlock` method does not need to be in parenthesis

- Function objects can access names in the enclosing scope of where they are declared:

```
val start=2
def end=10
timeBlock {
  (start to end).foreach {
    x => println("x = " + x)
  }
}
```

- This is a form of a *Lexically Scoped Closure*
- Scala's influence on Java shows up in the BGGGA (Bracha, Gafter, Gosling, and von der Ahé) closure proposal. Martin Odersky is listed as a contributor.

- Functional languages use a technique called *currying* that converts function calls with multiple parameters to a series of function calls with usually one parameter
- One way that Scala support currying is to allow multiple parameter lists on method declarations:

```
def repeat(n: Int) (block: =>Unit) =  
    (1 to n).foreach { x => block }  
repeat(2) { println("yada") }  
val fiveTimes = repeat(5)_  
...  
fiveTimes { println("blah") }
```

- The “`repeat(5)_`” expression results in a partially applied function object that can be invoked with remaining parameters to complete the invocation of the function

- Currying can also be done in Scala using nested functions:

```
def repeat(n: Int) = {  
  def executeBlock(block: =>Unit)  
    = (1 to n).foreach { x => block }  
  executeBlock _  
}
```

```
val sevenTimes = repeat(7)  
sevenTimes { println("hello") }
```

- The `repeat` method returns partially applied function object as indicated by the “`executeBlock _`” expression

- List and tuple types are very common in functional programming languages and Scala implements them both

- Both types are immutable

- Lists are a grouping of objects of a common type:

```
var lst = List(1, 2, 3, 4) // List[Int]
```

- Singleton object, `List`, using an `apply` method with variable arguments act as a factory:

- Tuples are a grouping of objects of differing types:

```
var tup = (1, 1.0, "one")  
          // Tuple[Int, Double, String]
```

- Tuples are simply grouped in parenthesis

- Lists can also be declared using different syntax:

```
val lst = 1 :: 2 :: 3 :: 5 :: 8 :: Nil
```

- `Nil` is a special instance of `List` that represents an empty list

- This is an example of a right associative operator. Equivalent expression:

```
val list1 = Nil :: (8) :: (5) :: (3) :: (2) :: (1)
```

- Four common operations on lists are:
 - Return first item of list:
`List(1,3,4).head // result: Int = 1`
 - Return remainder of list without first item:
`List(1,3,4).tail // result: List[Int]=(3,4)`
 - Apply function to all items and return result `List`:
`List(1,3,4).map(x => x * 2)`
`// result: List[Int] = (2,6,8)`
 - Apply function object that returns a `Boolean` to all items and return `List` of items where result is true:
`List(1,3,4,5).filter(x => x % 2 == 1)`
`// result: List[Int] = (1,3,5)`

- Scala Tuple objects have methods in the form, `_<n>`, where `n` is number between one and number of objects in the tuple:

```
val tuple1 = (1, "one", 1.0)
```

```
val tuple2 = (1, "two", 2.0)
```

```
tuple1._1 // result: Int = 1
```

```
tuple1._2 // result: String = "one"
```

```
tuple2._3 // result: Double = 2.0
```

- Tuples are immutable, so there are only accessors for tuple fields and no mutators

- Pattern matching is another common feature of functional languages
- Allows extraction of values from matched pattern
- Scala uses a **match/case** construct to implement
- Example with basic types:

```
def doMatch(any:Any) = any match {  
  case x:Int => "Integer = " + x  
  case x:Float => "Float = " + x  
  ...  
doMatch(1)      // result: String = "Integer = 1"  
doMatch(1.0F)  // result: String = "Float = 1.0"
```

- Pattern matching with lists:

```
def doMatch(any:Any) = any match {  
  ...  
  case x::10::rest => "List head = " + x + "  
  second = ten rest = " + rest  
  case x::rest => "List head = " + x + " rest = "  
  + rest  
  ...  
  doMatch(7.5::2.5::Nil)  
  // "List head = 7.5 rest = List(2.5)"  
  
  doMatch(7::10::19::Nil)  
  // "List head = 7 second = ten rest = List(19)"
```

- Pattern matching with tuples:

```
def doMatch(any:Any) = any match {  
  ...  
  case (x,10) => "Tuple(" + x + ", ten)"  
  case (x:Int,y:Int) if (x + y) == 10 =>  
    "Tuple(" + x + ", " + y + ") sum = 10"  
  case (x,_) => "Tuple(" + x + ", other)"  
  ...  
  doMatch((3,10)) // "Tuple(3,ten)"  
  doMatch((4,6))  // "Tuple(4,6) sum = 10"  
  doMatch((1,2))  // "Tuple(1,other)"
```

- Case clauses can include `if` expressions called *guards* to further to specify the match
- Underscore, "`_`", is as wild card that can be within a pattern. It does not extract any value

- To match patterns to user defined classes and singleton objects, Scala provides “**case**” classes and objects:

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

- The **case** keyword adds functionality to facilitate pattern matching
 - For classes, generates a companion singleton **object** to act as a factory
 - Case classes can be instantiated without **new** because the factory object has **apply** method with same signature as primary constructor
 - Implements code to extract properties

- Pattern matching with `case class`:

```
case class Person(name:String, age:Int)
...
def doMatch(any:Any) = any match {
...
  case Person(name, age) if (age < 18) =>
    "Young Person named " + name
  case Person(name, age) =>
    "Adult Person named " + name
...
doMatch(Person("John", 6))
// "Young Person named John"
doMatch(Person("Tim", 40))
// "Adult Person named Tim"
```

- Underscore is used as a catch-all clause much like the `default` clause in a Java `switch` statement:

```
def doMatch(any:Any) = any match {  
  ...  
  case _ => "Something else"  
  ...  
doMatch(new Date()) // "Something else"
```

- Java's `try/catch` is a basic form of pattern matching. Scala uses a `case` clauses within `catch` blocks:

```
try {  
    // Exception prone code  
    ...  
} catch {  
    case ex:FileNotFoundException =>  
        System.err.println("File Not Found")  
    case ex:EOFException =>  
        System.err.println("Unexpected EOF")  
    case ex:Exception =>  
        System.err.println("Other Exception "  
            + ex.getMessage)  
}
```

- The full potential of pattern matching in Scala is sometimes limited by the “Wall of Erasure”
 - The following will generate a warning because they are in effect the same pattern after erasure:

```
def doMatch(any:Any) = any match {  
  case x:List[String] => "List[String]"  
  case x:List[Int]   => "List[Int]"  
}  
...  
doMatch(List(1,2,3)) // "List[String]" !!!
```

- Scala has traits that are like Ruby mixins that enable a form of multiple inheritance
 - Like Java interfaces, but can include implementation:

```
trait Foos {  
  def doFoo(text:String) =  
    printf("{0}: Foo({1})\n",this.toString, text)  
}  
class FooClass extends Foos {  
  override def toString = "FooClass"  
}  
...  
val foo = new FooClass  
foo.doFoo("one") // "FooClass: Foo(one)"
```

- Scala classes can use multiple traits:

```
trait Bars {
  def doBar(text:String) =
    printf("{0}: Bar({1})\n",this.toString, text)
}
class FooBarClass extends Foos with Bars {
  override def toString = "FooBarClass"
}
...
val fooBar = new FooBarClass
fooBar.doFoo("one") // "FooBarClass: Foo(one)"

fooBar.doBar("two") // "FooBarClass: Bar(two)"
```

- Objects with traits can be declared inline:

```
class Now {  
  override def toString =  
    new java.util.Date().toString()  
}  
  
...  
val fooBarNow = new Now with Foos with Bars  
fooBarNow.doFoo("three")  
// "<current time>: Foo(three)"  
fooBarNow.doBar("four")  
// "<current time>: Bar(four)"
```

- Scala supports Sequence Comprehensions that act a sort of query language
 - Can iterate over one or more sequences
 - Apply conditionals for filtering
 - Return a new sequence or apply functions to each item
- Simple Sequence Comprehension returning a new sequence:

```
for (i<- 1 to 10 if i % 2 == 0) yield i
// result:Seq[Int] object with contents:
// (2, 4, 6, 8, 10)
```

- More complex comprehension:

```
class Team(val name:String, val score:Int)

var teams = List(new Team("Daleks", 93)
                  ,new Team("Vogons", 55)
                  ,new Team("Ewoks", 33))

for (t1<-teams;
     t2<-teams if t1 != t2
     && t1.score > t2.score) {
println(t1.name + " defeated " + t2.name) }
```

- Output:

```
Daleks defeated Vogons
Daleks defeated Ewoks
Vogons defeated Ewoks
```

- Scala supports XML as a built-in data type:

```
val xmlDoc = <phonebook>
  <entry>
    <name>Joseph Blow</name>
    <number type="home">312-542-2311</number>
    <number type="mobile">526-321-8729</number>
  </entry>
  <entry>
    <name>John Q. Public</name>
    <number type="home">526-442-9332</number>
    <number type="mobile">312-333-1228</number>
  </entry>
</phonebook>
```

- XML document can be queried using operators resembling XPath expressions and processed using sequence comprehensions:

```
for (name <- xmlDoc\\"name") println(name.text)
```

Output:

```
Joseph Blow  
John Q. Public
```

```
for (number <- xmlDoc\\"number"  
      if number.text.startsWith("526") {  
  println(number.text);  
}
```

Output:

```
526-321-8729  
526-442-9332
```

- Scala expressions can be embedded in the XML within curly braces, “{ }” :

```
class HelloServlet extends HttpServlet {
  override def doGet(
    request:HttpServletRequest,
    response:HttpServletResponse) = {
    response.getWriter.println(
      <html>
        <body>
          <h1>Hello
            { request.getParameter("user") }
          </h1>
        </body>
      </html>
    )
  }
}
```

- Actors are basically concurrent processes that communicate via message passing.
 - This language feature was inspired by a similar concept in Erlang
 - Messages can be passed synchronously or asynchronously
 - Messages are stored in queues until they are processed
 - Messages are simply objects that are handled using pattern matching
 - Actors can be thread based or event based

- Thread based Example:

```
case class Stop
class Receiver extends Actor {
  def act() {
    while (true) {
      receive {
        case msg:String => {
          println("Receiver received:\n" + msg)
          sender ! "Yes I can."
        }
        case _:Stop => exit()
      }
    }
  }
}
```

- The **act** method is analogous to the **run** method in threads
- The **receive** method blocks until a message is received
- The “!” method sends a response to the **sender**

- Example Continued:

```
class Initiator(receiver:Actor) extends Actor {
def act() {
  receiver ! "Can you here me now?"
  receive {
    case response:String => {
      println("Initiator received response:\n"
        + response)
    }
  }
  receiver ! Stop
}
}
```

- A message is sent to the **Receiver**
- The actor waits for and processes the response
- A **Stop** object is sent to tell the **Receiver** to exit.

- Example Continued:

```
val receiver = new Receiver
val initiator = new Initiator(receiver)
receiver.start
initiator.start
```

Output:

```
Receiver received:
Can you here me now?
Initiator received response:
Yes I can.
```

- Scala actors use a thread pool that initially contains four threads
- When an actor blocks in a **receive** a thread is blocked
- The actor library will grow the pool if a new thread is needed and all threads are blocked
- Scala actors have a **loop/react** construct that does not block a thread
 - Due to some complexity of how **react** is implemented, nesting it in a **while** loop does not work.

- `Receiver` class re-implemented using `loop/react`:

```
class Receiver extends Actor {  
  def act() {  
    loop {  
      react {  
        case msg:String => {  
          println("Receiver received:\n" + msg)  
          sender ! "Yes I can."  
        }  
        case _:Stop => exit()  
      }  
    }  
  }  
}
```

- This is an example of an event based actor
- Event based actors scale better than thread based one.

- Scala supports Anonymous (or Structural) typing
 - Types can be declared based on methods implemented
 - This is a form of “duck” typing:

```
type Duck = {
  def quack:String
  def waddle(d:Int, u:String):String
}

class Foo {
  def quack = "Foo - quacks"
  def waddle(distance:Int, units:String) =
    "Foo waddles " + distance + " " + units
}

class Bar {
  def quack = "Bar - quacks"
  def waddle(distance:Int, units:String) =
    "Bar waddles " + distance + " " + units
}
```

- Example Continued:

```
val duckList = List[Duck](new Foo, new Bar)
duckList.map(_.quack).foreach(println)
duckList.map(_.waddle(10, "feet")).foreach(println)
```

Output:

```
Foo - quacks
Bar - quacks
Foo waddles 10 feet
Bar waddles 10 feet
```

- The “`_.quack`” expression is shorthand for `(x => x.quack)` where type of `x` can be inferred
- The “`println`” expression can be expressed as “`println _`” which in turn is shorthand for `(x => println x)`

- Summary
 - Scala's features make it as close to a dynamic language as a statically type language can get
 - Maintains the performance characteristics of compiled Java
 - Scala approximates features that make dynamic languages like Ruby and Groovy attractive
 - Functional language features allow Scala programs to follow a more functional style when it is better suited for the task at hand or take an non-functional imperative approach
 - Features of Scala will certainly get consideration for inclusion in future versions of Java
 - The unique characteristics of Scala as compared to other languages for the JVM platform make it compelling language for Java developers to learn

- Any Questions ???

- Scala language home:
<http://www.scala-lang.org>
- Scala distribution download page
<http://www.scala-lang.org/downloads/index.html>
- Artima's Scalazine online magazine
<http://www.artima.com/scalazine>
- Whitepaper, *Actors That Unify Threads and Events*
by Philipp Haller and Martin Odersky:
<http://lamp.epfl.ch/~phaller/doc/haller07coord.pdf>
- Lift – Web framework in Scala:
<http://www.liftweb.net>