Exploring Scala

Another Programming Language
Or a Mood Altering Substance?

Overview. Some Topics:

- Functional Programming
- Scala as a general purpose scripting language.
- Scala as a concise form of Java
- Using the REPL as a learning tool.
- Similarities / Differences from Java
- IDE Support in IntelliJ Idea?
- Operator Overloading and Operator Over-Doing
- SBT, the "Simple Build Tool" for Scala that's impossible to use! (Or is it?)

Scala's Relation to Java

- Natural Order:
  - Similarities / Differences
  - What's Better / What's Worse
- Presentation Order:
  - Similarities
  - What's Worse
  - What's Better
  - Differences

Similarities:
How Are Java & Scala **The Same**?

- Both run on the JVM and share common JVM features internally.
- Because of this Scala can trivially use existing Java libraries / tools.
- Both are strongly typed.
- Great support for object oriented programming.
- Runtime performance is comparable for comparable language features.

What Scala Does **WORSE** than Java

- Compile time is worse (But... Play!).
- Learning curve / complexity for some features.
- Obscurity / pedantry:
  - operator overloading =~!*# operator overdoing
- Some tools particularly difficult / obtuse (Parts of SBT for example).
- Still a Little Bit of Newness Penalty:
  - Third party tooling catching up (less confident now).
  - Java has longer history of documentation & support.

What Scala Does **BETTER** than Java

- Natural support for functional programming
  - Compare my history with C → C++
- A different, and much simplified, way of doing concurrent / parallel programming.
- Better support for scripting.
- I can haz REPL!
  - Tee shirt idea: *“Programmers do it interactively.”*
- Generally much more concise.

How Java and Scala are **DIFFERENT**

- Lots of syntactical differences
  - Good ones
  - Surprising ones
- Supporting functional programming / stuff borrowed from other FP languages (Erlang , etc.)
  - Good stuff
  - Ugly stuff

How Java and Scala are **DIFFERENT**

- Objects Everywhere in Scala

  Scala will compile many expressions to primitive types for performance reasons, but in terms of the programming interface, everything is an object.

  Any ← AnyRef

  ← AnyVal

How Java and Scala are **DIFFERENT** (Continued)

Scala has type inference in many cases:

```
val greeting = “Hello World”
val answer = 42

var states = Array(“RI”, “IN”, “CA”)
var counter = 0


// But can also be explicit...
val theAnswer : Int = getAnswer()
```

Scala has no static methods. Instead uses “object” to define singletons.

```
// HelloWorld.scala

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

How Java and Scala are **DIFFERENT** (Continued)

- Many constructs return a value in Scala that don't in Java (i.e., more “Expression Based”):

```
// if_expression.scala

val answer = 42
val size = if (answer < 100) "big"
    else "little"
```

For Comprehensions with Yield Made Easy

- For also returns a value in an expression called a “for comprehension”. Python and Ruby also do these.

```
// for_comprehension.scala


val shouting = "HEY SHOUTING IS
  IMPOLITE".split(" ")
val whisper = for (s <- shouting)
  yield s.toLowerCase()
```

For Comprehsions with Yield Made Hard

```
// for_comprehension2.scala

val shouting2 = "HEY SHOUTING IS
  IMPOLITE".split(" ")
val whisper2 = for (s <- shouting2)
  yield {
    val t = s + "_NERD_SAYS_WHAT"
    t.toLowerCase()
  }
```

Match Expressions

- Scala has no switch statement, but match:
  - Can take any input
  - Can return a value or Unit (Unit is like void in Java, C, etc).

An untyped example matchUntyped.scala

```
// matchUntyped.scala

def roseColored(beloved: Any): Any = {
  beloved match {
    case "green eggs and ham" => -1
    case _ => beloved.toString() +
              " is beautiful!"
  }
}
```

“Typed match” example matchTyped.scala

```
// matchTyped.scala

def matchIntToString(someNumber: Int):
   String =    someNumber match {
     case 1 => "The loneliest number"
     case 2 => "It's company."
     case 3 => "It's a crowd."
     case _ => someNumber.toString()
   }
```

Functional Programming

- Is Scala a functional programming language?
- What Is It?
- What are the benefits?
- What are the difficulties?
- Scala as an impure functional language language

Functional Programming Defined

Functional programming is a style of programming that emphasizes programming using **pure functions** – that is, functions that do not have side effects and always return the same result given the same input.

“It treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.” – Wikipedia.

Scala – A Functional Language?

Unlike languages such as Haskell which are (or at least, which purport to be) purely functional, Scala is not a purely functional language BUT...

It has excellent support for functional and imperative programming.

Wikipedia: *"An interesting case is that of Scala – it is frequently written in a functional style, but the presence of side effects and mutable state place it in a grey area between imperative and functional languages."*

Functional Programming Defined (Continued)

Functional programming languages feature **first class functions**, that is, functions that can be assigned to variables, passed to other functions, and returned from functions.

This in turn supports the creation of **high order functions**, that is, functions that call other functions passed in as arguments.

Scala – A Functional Language?

Mixing Imperative and Functional Code?

- Should you?
- My answer is yes if:
  - You're in a hurry.
  - You're pragmatic (Just because you now own a hammer, the universe didn't suddenly become a collection of nails. What do you do with imperative APIs? What about rs.next())?

Functional Programming Defined (Continued)

Functional programming languages support a style of programming that is **declarative** (tell the machine what you want done) rather than **imperative** (tell the machine how to do every step, and as a result, tell it what you want done).

Imperative Programming
Copying an Array of Ints

```
// Imperative.scala

def copyIntArray(src: Array[Int]): Array[Int] = {

   var dest: Array[Int] = new Array[Int](src.length)
      var i = 0
      while (i < src.length) {
          dest(i) = src(i)
          i = i + 1
      }

   dest
}
```

Declarative Programming Function does the same thing

```
// Declarative.scala

def copyIntArray(src: Array[Int]): Array[Int] = {
   src.map(x => x)
}
```

[More on Functional Programming Here]

- Needed: High-order functions
- Needed: Partial functions

Concurrency With Scala Actors

- Currently implemented in “akka” library.
- “Simple high-level abstractions for concurrency and parallelism.”
- Event driven – relies on asynchronous message passing between threads in the same process.
- Can use the same mechanism across a cluster of machines – “location transparency”.
- Messages passed are generally immutable.

Actors “ask” and “tell”

- Tell is very common, recommended, high performance, fire and forget:

  myActorRef ! “Hello” // tell myActorRef “hello”

- Ask is low performance way of returning a future, should use only when you must:

  val future = myActorRef ? “Hello”

  // … Use future

“Typed match” example (Revisited)

```
// matchTyped.scala

def matchIntToString(someNumber: Int):
   String =    someNumber match {
     case 1 => "The loneliest number"
     case 2 => "It's company."
     case 3 => "It's a crowd."
     case _ => someNumber.toString()
   }
```

Match expressions are basis of Actor “receive” method

```
class SimpleActor extends Actor {
    def receive  = {
       case "Hey" => sender ! "Hay is for horses"
       case _ => sender ! "Unknown message type"
    }
```

Scala as a Scripting Language [Here?]