Scala Programming Introduction

June 07, 2016

Agenda

- Currying
  - Understanding Currying
  - Functions as return values
  - Automatic Resource Management - Practical example for implementing higher order functions
- Scala Closure
  - Understanding how scala implements closures
- Traits
  - Understanding Scala Traits
  - Traits as Decorators

Why Should I Learn Scala ?

"If I were to pick a language to use today other than Java, it would be Scala."

- James Gosling, creator of Java

"Scala, it must be stated, is the current heir apparent to the Java throne. No other language on the JVM seems as capable of being a "replacement for Java" as Scala, and the momentum behind Scala is now unquestionable. While Scala is not a dynamic language, it has many of the characteristics of popular dynamic languages, through its rich and flexible type system, its sparse and clean syntax, and its marriage of functional and object paradigms."

- Charles Nutter, creator of Jruby

"I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy."

- James Strachan, creator of Groovy.

Scala Pattern Matching / Case Classes

Coding Demos for Scala pattern matching

- Matching Literals
- Matching Constants
- Matching Lists
- Matching Types
- Matching Case Classes
- Pattern matching using custom extractors (left out as special topic)

Main take away:

Scala Pattern matching is extremely rich and out the box implementation of Visitor Design Pattern (GoF Design Patterns)

Scala: Functional Language Concepts

- Functions are first-class citizens
- Higher Order Functions
  - You can pass functions to functions as parameters / Anonymous Functions
  - Return functions from other functions / Currying
  - Nested functions within functions
- Closures are special forms of function values that close over or bound to variables defined in another scope or context.
- Currying
  - Currying in Scala transforms a function that takes more than one parameter into a function that takes multiple parameter lists.

Main take away:

All these above constructs allows for much more concise and reusable code. We can write higher level abstractions which are very either very verbose or not possible in pure object oriented languages like Java.

Scala Functions are Objects

Scala is a functional language, in the sense that every function is a value.

If functions are values, and values are objects, it follows that functions themselves are objects.

The function type S => T is equivalent to scala.Function1[S, T] where Function1 is defined as follows :

```
trait Function1[-S, +T] {
    def apply(x: S): T
}
```

So functions are interpreted as objects with apply methods.

For example, the *anonymous successor* function (x: Int) => x + 1 is expanded to

```
new Function1[Int, Int] {
    def apply(x: Int): Int =
        x + 1
}
```

Scala Type Hierarchy