Using Scala for building DSL's

Abhijit Sharma

Innovation Lab,
BMC Software

6TH IndicThreads.com Conference On JAVA

2,3 DEC 2011

PUNE, INDIA

What is a DSL?

- Domain Specific Language
  - Appropriate abstraction level for domain - uses precise concepts and semantics of domain
  - Concise and expressive for a specific domain – not general purpose
  - Domain experts can communicate, critique better with programmers and amongst themselves
  - Math – Mathematica, UI – HTML, CSS, Database - SQL

DSL's at large

- Build tool Ant is an XML based DSL
  - Task to build a jar with a dependency on task compile

```
<target name="jar" depends="compile">
  <mkdir dir="${build.dist}"/>
  <jar jarfile="${build.dist}/${name}-${version}.jar">
    <fileset dir="${build.classes}" includes="**"/>
  </jar>
</target>
```

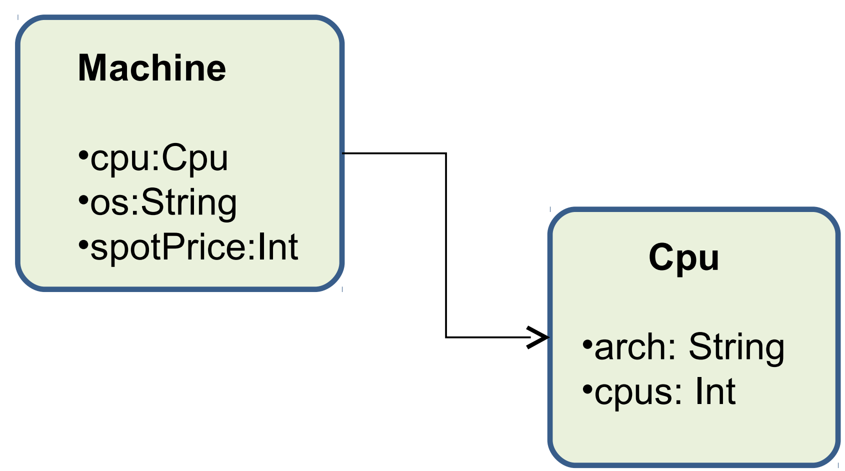
- Web App framework Ruby on Rails
  - ActiveRecord to model domain objects and persist them –
  - Domain constraint implementations do not clutter API – uniqueness, cardinality, null check

```
class Machine < ActiveRecord::Base
  has_one     :macAddress
  has_one     :hostName
  has_one     :os
  has_many    :softwares

  validates_uniqueness_of     :macAddress
  validates_presence_of       :os
end
```

Cloud Computing DSL

- Audience – Non tech savvy cloud end users
- DSL - English language sentence for requesting a machine with technical and price specifications
- Domain Concept – Machine
  - Technical specifications - cpu, os
  - Pricing specifications - spot price, pricing strategy – default or define inline

```
new Machine having (8 cpus "64bit") with_os "Linux"
  at_spot_price 30 with_price_strategy defaultPriceStrategy

new Machine having (8 cpus "64bit") with_os "Linux"
  at_spot_price 30 with_price_strategy {(x) => 1.15 * x}
```



Cloud Computing DSL in Java

- Builder pattern - Method Chaining Fluent Interface
- Issues
  - Syntax restrictions, verbosity – parenthesis, dot, semi-colons
  - Non Domain complexity – Builder
  - No Inline strategy – no higher order functions

```
new Machine.Builder()
                .cpus(8)
                .os("Linux")
                .atSpotPrice(30)
                .priceStrategy(new StandardPriceStrategy())
                .build();
```

```
public class Machine {
  static class Builder {
    ...
    public Builder cpus(int _cpus) {
      this._cpus = _cpus;
      return this;
    }
    public Builder os(String _os) {
      this._os = _os;
      return this;
    }
    ...
    public Machine build() {
      return new Machine(this);
    }
  }
  ...
  private Machine(Builder b) {
    _cpus = b._cpus;
    _os = b._os;
    ...
  }
}
```

DSL Classification

- Internal DSL
  - Embedded in a host language like Ruby, Scala, Groovy – use their features
  - Bound by host language syntax and semantics
- External DSL – standalone developed ground up
  - Define syntax and semantics as a grammar
  - Use tools like lexical analyzers, parsers, interpretation, code generators

Internal DSL Classification

- Internal DSL
  - Generative - Ruby, Groovy Techniques like runtime metaprogramming
    - Meta Objects – inject new behaviour at runtime
  - Embedded in host language
    - Smart API – Method chaining – Java etc.
    - Syntax tree manipulation – Groovy, Ruby libraries
    - Type Embedding – Scala – statically typed, type constraints, no invalid operations on types

Scala Language

- Scala is a “scalable” language
- JVM based – leverage libs, JVM perf, tools, install base etc
- Mixed Paradigm – Object Oriented + Functional Programming
    - Object Oriented Programming - Improves on Java OOP (Traits, no statics, advanced types)

Scala Language

- Functional Programming - Functions
  - No side effects and immutable variables
  - “First-class” citizens - Can be assigned, passed around, returned
  - Higher order functions promote composition using other more primitive functions
- Lots of powerful features – discussed later
- Statically Typed – Type Safe DSL

Scala – Readable style

- Type inference minimizes the need for explicit type information – still Type safe
- Elegant, succinct syntax unlike verbose Java
  - Optional dots, semi-colons and parentheses
  - Operators like methods
  - Syntactic sugar method takes one/zero argument, drop period and parentheses

```
// Java
Map<Integer, String> intToStringMap =
        new HashMap<Integer, String>();

// Scala Type Inference
val v: Map[String, String] = new HashMap()

// Scala Type Inference - the return type of the
// method can be inferred
def timesTwo(v: Int) : Int = 2 * v

// Omit the return
def timesTwo(v: Int) = 2 * v
```

```
// + is the Operator name
1 + 2

// Equivalent to - optional dot and parenthesis
1 .+(2)

// Readable style - optional dot and parenthesis
List(1, 2, 3, 4) filter ( 2 > ) foreach println

// x is the only argument - can be left out
List(1, 2, 3, 4) filter (x =>  2 > x)

List(1, 2, 3, 4) filter ( 2 > )
```

Scala - Implicits

- Implicits
  - Conversions – return a wrapped original type e.g. Integer
  - Implicit argument to function – don't need to pass – Concise syntax
- Generative expression - Implicit conversion converts the 1, an Int, o a RichInt which defines a 'to' method
- Lexically scoped – Unlike other languages like Groovy where such modifications are global in scope

```
// Generator Expressions - Int does not have the 'to' method
for (i <- 1 to 10) println(i)

// Int wrapped as RichInt which defines method 'to' and
// returns Iterable data structure
implicit def intWrapper(i: Int) = new RichInt(i)

val i = 6
i.toBinaryString
```

Scala - Higher order functions

- Functions which take other functions as parameters or return them as results are called higher-order functions
- Flexible mechanism for composition
- Currying

```
// 'filter' is a higher order function which takes a predicate
// function
List(1, 2, 3, 4) filter (x =>  2 > x)

// Summing up a series of Int from a to b after applying a
// function 'f' to each Int
def sum(f: Int => Int, a: Int, b: Int): Int =
if (a > b) 0 else f(a) + sum(f, a + 1, b)

// Use anonymous functions as 'f'
def sumInts(a: Int, b: Int): Int = sum(x => x, a, b)
def sumSquares(a: Int, b: Int): Int = sum(x => x * x, a, b)
```

```
/*
Currying
*/
def sum(f: Int => Int): (Int, Int) => Int = {
  def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)
  sumF
}

def sumInts = sum(x => x)
def sumSquares = sum(x => x * x)
```

Scala - Functional Combinators

- Calculate the total price of all Linux machines – uses several combinators – filter, map, foldLeft – all take other functions as predicates

```
val mc1: Machine =
   new Machine having (8 cpus "64bit") with_os "Linux" at_spot_price 30

val mc2: Machine =
   new Machine having (4 cpus "32bit") with_os "Win" at_spot_price 25
....
val machines = List(mc1, mc2, ...)

machines
   .filter(_.os == "Linux")
   .map(_.price * 100)
   .foldLeft(0)(_ + _)
```

Scala – Cloud Computing DSL - Implicits

- Consider excerpt - *8 cpus "64bit" – Using Implicit conversion we get the object representing the CPU - Cpu(8, 64bit)*

```
new Machine having (8 cpus "64bit") with_os "Linux"
  at_spot_price 30 with_price_strategy defaultPriceStrategy

new Machine having (8 cpus "64bit") with_os "Linux"
  at_spot_price 30 with_price_strategy {(x) => 1.15 * x}
```

```
// Cloud Computing DSL
val machine =
  new Machine having (8 cpus "64bit") with_os "Linux" at_spot_price 30

case class Cpu(cpus: Int, arch: String)

class CpuInt(qty: Int) {
 def cpus(arch: String) = {
  Cpu(qty, arch)
 }
}

implicit def cpuInt(i: Int) = new CpuInt(i)

// Take this excerpt of above DSL - '8 cpus "64bit"'
val cpu = 8 cpus "64bit"
Cpu(8, 64bit)
```

Scala – Cloud Computing DSL – E2E

DSL - *new Machine having (8 cpus "64bit") with_os "Linux"*

- Implicit Conversion
- Method Chaining – Builder pattern – without the cruft
- Syntactic sugar no parenthesis, dot, brackets

```
val machine =
   new Machine having (8 cpus "64bit") with_os "Linux"

machine.cpu = Cpu(8, "64bit")
machine.os = "Linux"

--> In CpuInt implicit conversion : Creating Cpu(8,64bit)
--> In Machine.having Cpu(8,64bit)
--> In Machine.with_os Linux
```

```
case class Cpu(cpus: Int, arch: String)

class CpuInt(qty: Int) {
  def cpus(arch: String) = {
    Cpu(qty, arch)
  }
}

implicit def cpuInt(i: Int) = new CpuInt(i)

class Machine {
   var cpu: Cpu = null
   var os: String = null

   def having(_cpu: Cpu) = {
     cpu = _cpu
     this
   }

   def with_os(_os: String) = {
     os = _os
     this
   }
}

val machine =
   new Machine having (8 cpus "64bit") with_os "Linux"
```

Scala - Cloud Computing DSL - Functions

- Using Higher Order Functions – Flexible pricing
- Spot Price Threshold - Inline strategy

```
new Machine having (8 cpus "64bit") with_os "Linux"
  at_spot_price 30 with_price_strategy defaultPriceStrategy

new Machine having (8 cpus "64bit") with_os "Linux"
  at_spot_price 30 with_price_strategy {(x) => 1.15 * x}
```

```
class Machine {
  ....
  var spotPrice: Int = 0
  var threshold: Double = 0

  def having(_cpu: Cpu) = {
    ...
  }
  def with_os(_os: String) = {
   ...
  }
  def at_spot_price(_spotPrice: Int) = {
   spotPrice = _spotPrice
   this
  }
  def with_price_strategy(_pricing: (Int) => Double) = {
    threshold = _pricing(spotPrice)
    this
  }
}
val machine =
  new Machine having (8 cpus "64bit") with_os "Linux"
    at_spot_price 30 with_price_strategy {(x) => 1.15 * x}

-> In Cpu implicit conversion : Creating Cpu(8,64bit)
-> In Machine.having Cpu(8,64bit)
-> In Machine.with_os Linux
-> In Machine.at_spot_price 30
-> In Machine.with_price_strategy computing threshold 34.5
```

Scala - Pattern Matching

- Pattern Matching – Switch Case on Steroids
- Cases can include value, types, wild-cards, sequences, tuples, deep inspection of objects

```
// Matching on Sequences

for (l <- lists) {
 l match {
   case List(_, 3, _, _) => ... // 4 elements, 2nd element being 3
   case List(_*) => ... //Any other list with 0 or more elements
 }
}

--

// Matching on Tuples (and Guards)

val tupA = ("m1", 8)
val tupB = ("m2", 4)

for (tup <- List(tupA, tupB)) {
 tup match {
   case (machine, cpus) if machine == "m1" =>
     println("A two-tuple for m1")
   case (machine, cpus) =>
     println("Not m1")
 }
}
```

Scala - Pattern Matching &Case Classes

- Case Classes – simplified construction and can be used in pattern matching
- Pattern matching on Case Classes
  - Deep pattern matching on object contents
  - Make good succinct powerful DSL

```
case class Machine(name: String, cpus: Int, arch: String)

val m1 = new Machine("m1", 8, "64bit")
val m2 = new Machine("m2", 4, "64bit")
val m3 = new Machine("m3", 2, "32bit")
val m4 = new Machine("m4", 2, "sparc32")

for (machine <- List(m1, m2, m3, m4)) {
 machine match {
  case Machine(name,  _, "64bit") => println(name + " : 64bit")
  case Machine(name,  _, "32bit") => println(name + " : 32bit")
  case Machine(name, cpus, arch) =>
   println("Machine " + name + " non standard arch : " + arch)
 }
}
```

Scala - Pattern Matching – Visitor Pattern

- Pattern match and case classes – extensible visitor
- Different operations on tree
  - Expression Evaluation
  - Prefix Notation
- Very expressive, flexible and concise code

```
abstract class Tree

case class Sum(l: Tree, r: Tree) extends Tree
case class Const(v: Int) extends Tree

def eval(t: Tree): Int = t match {
   case Sum(l, r) => eval(l) + eval(r)
   case Const(v) => v
}

val exp: Tree =
   Sum(Sum(Const(5),Const(5)),Sum(Const(7), Const(7)))
println("Expression: (5 + 5) + (7 + 7) : " + eval(exp))

> Expression: (5 + 5) + (7 + 7) : 24

def prefix(t: Tree): Unit = t match {
   case Sum(l, r) => print("+");prefix(l); print(" "); prefix(r);
   case Const(v) => print(v)
}

prefix(exp)

> ++5 5 +7 7
```

Scala – For Comprehensions

- Loop through Iterable sequences and comprehend/compute something
  - E.g. Filter 32, 64 bit architectures

```
val archs = List("32bit", "64bit", "sparc32")

for (arch <- archs) {
  arch match {
    case "32bit" => println("32bit")
    case _ => println("Other ")
  }
}

// Yielding results accumulate with every run, and the resulting collection
// is assigned to the value filteredArchs
val filteredArchs = for {
 arch <- archs
 if arch.contains("bit")
} yield arch
```

Scala – For Comprehensions + Option

- Wrap vars & function returns as Option – Null Checks, resilient programming
- Option sub classes : None and Some
- Options with for comprehensions, automatic removal of None elements from comprehensions

```
case class Machine(name: String, os: String)

val machines = List(
  Map("name" -> "m1", "os" -> "Linux"),
  Map("name" -> "m2", "os" -> "Win"),
  Map("name" -> "m3")
 )

var validMachines = for {
  machine    <- machines
  name <- machine get "name"
  os     <- machine get "os"
} yield Machine(name, os)

> List(Machine(m1,Linux), Machine(m2,Win))
```

```
val services = Map(
 "Amazon" -> "IAAS",
 "Azure"  -> "PAAS"
)

// Wrapped in Option - Some(IAAS)
println( "Amazon: " + services.get("Amazon") )

// Get Value - IAAS
println( "Amazon: " + services.get("Amazon").get )

// Wrapped in Option - None
println( "Unknown: " + services.get("Unknown") )

// Alternate way when it is None
println( "Unknown: " + services.get("Unknown").getOrElse("Not Present") )
```

Scala - For Comprehensions + Option

- Validate and audit machines
- Using Options with for comprehensions eliminate the need for most "null/empty" checks.
- Succinct, safe DSL with uncluttered API

```
// Wrap in Option
def validate(machine: Machine): Option[Machine] = isValid(machine match {
    case true => Some(machine)
    case _ => None
}

// Wrap in Option
def audit(machine: Machine): Option[Machine] = Some(machine) //..stubbed

// stub
def isValid(machine: Machine) = true


val mc1: Machine =
    new Machine having (8 cpus "64bit") with_os "Linux" at_spot_price 30

val mc2: Machine =
    new Machine having (4 cpus "32bit") with_os "Win" at_spot_price 25

val machines = List(mc1, mc2)

// For comprehension
val validMachines =
    for {
        machine <- machines
        mcValidated <- validate(machine)
        mcFinal <- audit(mcValidated)
    }
    yield mcFinal
```

Scala - Traits

- Traits are collections of fields and behaviors that you can extend or mixin to your classes.
- Modularize these concerns, yet enable the fine-grained "mixing" of their behaviors with other concerns at build or run time – Callbacks & Ordered
- Traits can be mixed-in at class level or at instance creation
- AOP Pervasive concerns - Logging, Ordering, Callback Handling

```
trait Ord {
   def < (that: Any): Boolean
   def > (that: Any): Boolean = !(this <= that)
...
}
class Date(y: Int, m: Int, d: Int) extends Ord {
   ..
   def <(that: Any): Boolean = {
      ...
   }
}
```

```
trait Subject {
 type Observer = { def receiveUpdate(subject: Any) }

 private var observers = List[Observer]()
 def addObserver(observer:Observer) = observers ::= observer
 def notifyObservers = observers foreach (_.receiveUpdate(this))
}

class Button(name: String) extends Subject {
 def click() = {
  notifyObservers
 }
}

class ButtonObserver {
 def receiveUpdate(subject: Any) = println("Called with " + subject)
}

val button = new Button("Okay")
val buttonObserver = new ButtonObserver
button.addObserver(buttonObserver)
button.click()
```

External DSL in Scala

DSL - having (8 cpus "64bit") with_os "Linux" at_spot_price 30

- Parser Combinator library – available as a library on host language – Scala
- External Parser Generators like Javacc – use tools to generate code for tokenizing, parsing
- Parser Combinator Specification is like a BNF grammar

```
expr      ::= machine
machine ::= "having" tech_spec price_spec
tech_spec  ::= cpu_spec os_spec
cpu_spec  ::= "(" numericLit "cpus" stringLit ")"
os_spec ::= "with_os" stringLit
price_spec ::= "at_spot_price"  numericLit
```

External DSL in Scala

- Each function is a parser - works on a portion of the input, parses it and may optionally pass on the remaining part to the next parser in the chain via the combinator
- Several combinators provided by the library like '~' the sequencing combinator composes two parsers sequentially.
- Optional function application combinator (^^) can work, applying the function on the result of the sequencing combinator.

```scala
// having (8 cpus "64bit") with_os "Linux" at_spot_price 30

object MachineDsl extends StandardTokenParsers {
  lexical.reserved +=
    ("having", "cpus", "with_os", "at_spot_price")
  lexical.delimiters += ("(", ")")

  lazy val machine =
    "having" ~ tech_spec ~ price_spec

  lazy val tech_spec =
    cpu_spec ~ os_spec

  lazy val cpu_spec =
    "(" ~> numericLit ~ "cpus" ~ stringLit <~ ")"

  lazy val os_spec =
    "with_os" ~> stringLit

  lazy val price_spec =
    "at_spot_price" ~> numericLit
}
```

```scala
lazy val cpu_spec =
    "(" ~> numericLit ~ "cpus" ~ stringLit <~ ")"
            ^^ { case n ~ "cpus" ~ a=> Cpu(n.toInt, a) }
```

Thanks

abhijit.sharma@gmail.com

Twitter : sharmaabhijit

Blog : abhijitsharma.blogspot.com