
Busy Java Developer's Guide to Scala (Part One: Objects)

Ted Neward
Neward & Associates
<http://www.tedneward.com>

Credentials

- Who is this guy?
 - Independent consultant, architect, mentor
 - Instructor, PluralSight (www.pluralsight.com)
 - BEA Technical Director, Microsoft MVP Architect
 - JSR 175, 250, 277 EG member
 - Founding Editor-in-Chief, TheServerSide.NET
 - Author
 - *Effective Enterprise Java* (Addison-Wesley, 2004)
 - *Effective .NET* (Forthcoming)
 - *Pragmatic XML Services* (Forthcoming)
 - *Pragmatic .NET Project Automation* (Forthcoming)
 - *Server-Based Java Programming* (Manning, 2000)
 - *C# in a Nutshell* (OReilly, 2003)
 - *SCLI Essentials* (w/Stutz, Shilling; OReilly, 2003)
 - Papers at <http://www.tedneward.com>
 - Weblog at <http://blogs.tedneward.com>

Objectives

- See a bit of Scala's syntax
- See some of the underlying principles of functional programming
- See why Scala is interesting

Java-Scala comparison

- Quicksort in Java

```
void sort(int[] xs) {
    sort(xs, 0, xs.length -1 );
}
void sort(int[] xs, int l, int r) {
    int pivot = xs[(l+r)/2];
    int a = l; int b = r;
    while (a <= b)
        while (xs[a] < pivot) { a = a + 1; }
        while (xs[b] > pivot) { b = b - 1; }
        if (a <= b) {
            swap(xs, a, b);
            a = a + 1;
            b = b - 1;
        }
    }
    if (l < b) sort(xs, l, b);
    if (b < r) sort(xs, a, r);
}
void swap(int[] arr, int i, int j) {
    int t = arr[i]; arr[i] = arr[j]; arr[j] = t;
}
```

Java-Scala comparison

- Quicksort in Scala

```
def sort(xs: Array[Int]): Array[Int] =  
  if (xs.length == 1) xs  
  else {  
    val pivot = xs(xs.length / 2)  
    Array.concat(  
      sort(xs filter (pivot >)),  
      xs filter (pivot ==),  
      sort(xs filter (pivot <))  
    )  
  }
```

Processing XML

- Prints XML to the console

```
object XMLTest extends Application {
  import scala.xml._

  val df = java.text.DateFormat.getDateInstance()
  val dateString = df.format(new java.util.Date())
  def theDate(name:String) =
    <dateMessage addressedTo={ name }>
      Hello, { name }! Today is { dateString }
    </dateMessage>;
  Console.println(theDate("Scott Davis").toString())
}

object XPathQuery(val xml) {
  import scala.xml._
  def main(args: Array[String]): Unit = {
    Console.println(xml \ "book" \ "title")
    Console.println(xml \\ "title")
    Console.println(xml \\ "_")
    Console.println(xml.descendant_or_self())
  }
}
```

Scala

"Scala is a Java-like programming language which unifies object-oriented and functional programming.

"It is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes. Classes can be composed using mixin composition. Scala is designed to work seamlessly with two less pure but mainstream object-oriented languages—Java and C#."

"Scala is a functional language in the sense that every function is a value. Nesting of function definitions and higher-order functions are naturally supported. Scala also supports a general notion of pattern matching which can model the algebraic types used in many functional languages."

Scala

- **Developed at EPFL from 2001**
 - by Martin Odersky (of Pizza & GJ fame)
- **Six years of development**
 - Version 1.0 released in November 2003
 - Version 2.0 released in March 2006
 - Latest (Oct 2007) is 2.6.1
- **Growing community**
 - www.scala.org

Scala

- Object-oriented
 - every value is an object
 - types and behavior described by classes and traits
 - abstractions extended by subclassing and mixins
- Functional
 - every function is a value
 - lightweight syntax for anonymous functions
 - functions can be nested
 - supports currying
 - case classes and built-in pattern matching

This naturally extends to XML, making XML processing significantly easier and powerful, and particularly useful for building XML services

Scala

- **Statically typed**
 - generic classes
 - variance annotations
 - upper and lower type bounds
 - inner classes & abstract types as object members
 - compound types
 - explicitly typed self references
 - views
 - polymorphic methods
- **Type-inferenced**
 - no redundant casts required

Example: Hello, Scala

- Self-explanatory output

```
object HelloWorld {  
  def main(args: Array[String]): Unit = {  
    System.out.println("Hello, Scala!")  
  }  
}
```

Compile with scalac and execute with scala, in parallel fashion to what we see with javac and java, or pass in to the Scala interpreter, scalaint, for direct execution

object keyword indicates a Singleton instance

Line terminators can be either ; or newline

Symbols (main) defined in "<keyword> <name>: <type>" patterns throughout the language

unit roughly equal to void

Example: JVM Interop

- Calculating factorial

```
object BigFactorial {  
  import java.math.BigInteger, BigInteger._  
  
  def fact(x: BigInteger): BigInteger = {  
    if (x == ZERO) ONE  
    else x multiply fact(x subtract ONE)  
  }  
  def main(args: Array[String]): Unit = {  
    System.out.println("fact(100) = " +  
      fact(new BigInteger("100")))  
  }  
}
```

import package.type._ brings in all symbols (a la static import)

Equality tested with == operator, a la equals()

functions (like fact) implicitly return last value calculated

x multiply y is another syntax for x.multiply(y)

Example: Objects throughout

- Simple interpreted Scala expression

`1 + 2 + 3 / x`

`1.+(2.*(3./(x)))`

All values are objects, even constants

"Operators" are, in fact, methods

"+", "=", "-", and others, are valid identifiers in Scala

this will be tremendously useful in working with XML in Scala

Example: Objects throughout

- Functions are objects

```
object Timer {  
  def main(args: Array[String]): Unit = {  
    oncePerSecond(timeFlies)  
  }  
  
  def oncePerSecond(callback: () => Unit): Unit =  
    while (true) { callback(); Thread.sleep(1000) }  
  
  def timeFlies(): Unit =  
    Console.println("time flies when you're having fun")  
}
```

callback is a function object taking () returning unit

oncePerSecond calls callback() every second

Methods need not be composite blocks if they are simple

Console is Scala stdin/stdout wrapper

Example: Anonymous functions

- Define functions at point of usage

```
object Timer {  
  def main(args: Array[String]): Unit = {  
    oncePerSecond(() => Console.println("Beep"))  
  }  
  
  def oncePerSecond(callback: () => Unit): Unit =  
    while (true) { callback(); Thread.sleep(1000) }  
}
```

timeFlies isn't used more than once—why waste the space?

() => <code> syntax defines anonymous method

Example: Classes

- Classes define types

```
class Complex(real: Double, imaginary: Double) {  
  def re() = real  
  def im() = imaginary  
  override def toString() =  
    "" + re + (if (im < 0) "" else "+") + im + "i"  
}  
  
object Application {  
  def main(args: Array[String]): Unit = {  
    val c = new Complex(1.5, 2.3)  
  }  
}
```

On construction, Scala executes whole body of defined class

Scala infers types for re, im, and c, based on available type information already present

Can also rewrite re and im to be methods without arguments (remove parens)

Must use override to override def: defines a method

def this(...): defines an alternate constructor

val: initializes a constant (immutable) value

var: defines a (mutable) variable

object and class may nest

Example: POJOs v1

- Creating a POJO-compatible Scala class

```
class Person(firstName:String, lastName:String, age:Int)
{
  def getFirstName = firstName;
  def getLastName = lastName;
  def getAge = age;

  def setFirstName(value:String):Unit = firstName = value
  def setLastName(value:String):Unit = lastName = value
  def setAge(value:Int):Unit = age = value

  override def toString =
    "[Person firstName:"+firstName+" lastName:"+lastName+
    " age:"+age+]"
}
```

Example: POJOs v2

- Creating a POJO-compatible Scala class

```
class Person(fn:String, ln:String, a:Int) {  
  @scala.reflect.BeanProperty  
  var firstName = fn  
  
  @scala.reflect.BeanProperty  
  var lastName = ln  
  
  @scala.reflect.BeanProperty  
  var age = a  
}  
  
object App {  
  def main(args: Array[String]): Unit = {  
    val p = new Person("Neal", "Ford", 39)  
  }  
}
```

Example: Comprehensions

- Comprehensions, not *for*-loops

```
object ComprehensionTest extends Application {
  def even(f: Int, t: Int): List[Int] =
    for (val i <- List.range(f, t); i % 2 == 0) yield i

  def twoNumsSum(n: Int, v: Int): Iterator[Pair[Int, Int]] =
    for (val i <- Iterator.range(0, n);
         val j <- Iterator.range(i+1, n);
         i + j == v)
      yield Pair(i, j);

  twoNumsSum(20, 32) foreach {
    case Pair(i, j) =>
      Console.println("(" + i + ", " + j + ")")
  }
}
```

Comprehension have the form for (enums) yield e, where:

enums = semicolon list of enumerators (a generator or a filter)

e = body of code evaluated for each binding generated by the enumerator

Omit "yield" to have the comprehension return unit

More

- Much, *much* more:
 - Currying
 - Upper and Lower Type Bounds
 - Actors (Concurrency model)
 - Traits
 - Views

Summary

- Scala represents...
 - ... a "pure" object-oriented language
 - ... a "impure" functional language
 - ... an interesting hybrid of the concepts
 - ... a powerful look at what may be "Java 2.0"

Questions

