Author: Andre Amirsaleh
Language: Scala
Course: CSCI 208 (1:00PM Section)
Professor: Benoit Razet
Date: 12/5/16

Abstract

This paper analyzes and demonstrates some of the core design choices of Scala, the purely object-oriented yet functional programming language.

I. Paradigms at a Glance

Scala is object-oriented and functional. In Scala, every value is an object, and objects are represented by classes and traits [1-a]. Scala supports its classes with subclassing as well as a "mixin-based composition mechanism," which is effectively a form of multiple inheritance – allowing programmers to easily reuse classes [1-a, 1-b]. Scala also implements the functional programming paradigm with features such as nested functions, currying, case-classes, and pattern-matching. Scala code is written in an imperative style with loops and assignments [1-a].

II. Background

Martin Odersky's work on Java generics and the javac compiler is what led him and his team at Sun Microsystems to the development of Scala, the scalable language – hence the name – that adapts to specific needs via user-defined libraries [4, 5-a, 5-b]. Scala was publically released in 2004, but its development continues to this day [4, 5-b]. According to The Redmonk Programming Language Rankings of 2016, Scala is ranked as the 14th-most popular programming language of 2016 [8-a]. It is most notably used by Twitter, Netflix, LinkedIn, Sony, and Apple.

Scala is designed as a general-purpose language that seamlessly interoperates with Java programs. In fact, Scala utilizes some of Java's libraries as well as all of Java's primitive types. Moreover, Scala, like Java, is compiled into Java bytecode, which is then ran via a Java virtual machine [4]. Thus, Scala's performance is on par with that of Java [5-b]. However, Scala glaringly differs from Java in that Scala has a built-in interpreter in addition to its built-in compiler. Scala is also less verbose and less restrictive in most cases [4].

III. Hello, World

This section explains how to create and run a hello world program in Unix. First, create an empty file and name it HelloWorld.scala. Then, insert the following code [1-c].

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

Compile the source file and output the resulting executable file to a specified directory using the command, scalac, in the terminal as follows [1-c].

```
scalac HelloWorld.scala –d <some_directory>
```

Run the compiled code using the scala command in the terminal as shown below [1-c].

```
scala HelloWorld
```

Admittedly, this hello world program is intentionally syntactically reminiscent of Java code and is thusly slightly longer than it needs to be using only default libraries. Regardless, such a simple program is best scripted in the built-in interpreter. Launch Scala's interpreter from the terminal with the following command [4].

```
scala
```

Inside the interpreter, enter the following command [4].

```
println("Hello, World!")
```

Exit the interpreter using the following command [4].

```
:q
```

IV. Scaladocs

Scaladocs are used to clearly and concisely display crucial information regarding all packages, classes, traits, methods, and other members [1-e]. See the code snippet below for a standard issued.

```
/** Short description here
 *
 * More info here…
 * As needed…
 */
def someFunc = {}
```

At first glance, scaladocs appear syntactically identical to javadocs. However, scaladocs offer more functionality yet take less room in the code [1-e].

V. Math Operations

The following code – copied from the interpreter – demonstrates some of the basic math operations that are possible in Scala.

```
scala> 1 + 4
res3: Int = 5
scala> 1 – 4
res4: Int = –3
scala> 1 * 4
res5: Int = 4
scala> 16 / 4
res6: Int = 4
scala> 'a' + 'a'
```

```
res7: Int = 194
scala> 'a' * ('a' + 1)
res8: Int = 9506
scala> 'b' * ('a' + 1)
res9: Int = 9604
```

VI. Primitive Types

Scala's primitive types and arrays are mapped directly to those of Java [4]. Thus, Scala's primitive types are identical to those of Java in terms of their memory footprints and ranges [2-a, 4]. The table below lists Scala's primitive types and their corresponding sizes in memory and their minimum and maximum values.

Table 1. Scala's primitive types and corresponding sizes in memory and minimum and maximum values [2-a, 3-a, 9-b].

Type	**Size [bits]**	**Min. & Max. Values (inclusive)**
Byte	8	-128 & 127
Short	16	-32768 & 32767
Int	32	$(-2)^{31}$ & $2^{31} - 1$
Long	64	$(-2)^{63}$ & $2^{63} - 1$
Float	32	$\pm 3.40282347 \times 10^{38}$
Double	64	$\pm 1.79769313486231570 \times 10^{308}$
Char (unsigned Unicode)	16	0 to 65,536
Boolean	1 (not precisely defined)	true or false

Table 1 neglects to mention that the String type is imported from a Java library by default [4].

VII. Variables

```
scala> myVal1 = 11
<console>:13: error: not found: value myVal1
val $ires0 = myVal1
             ^
<console>:11: error: not found: value myVal1
       myVal1 = 11
```

VIII. Multidimensional Arrays

Scala allows for multidimensional arrays with up to five separate dimensions [4].

VI. Variable Scoping and Access Modifiers

VII. Statically Typed

Scala is statically typed [4]. To demonstrate this, the second line in the following code in Scala's interpreter throws an exception because a variable's type cannot change once the variable has been declared.

```
scala> var myVar: Int = 2
scala> myVar = "Hello, World!"
```

The string “Hello, World!” does not implicitly coerce to an Int, so the interpreter catches the error, displays an error message, and then continues to run like normal.

VIII. Type Strength and Coercion

Although Scala is strongly typed, it will implicitly coerce types whenever possible. The following code demonstrates the implicit coercion of a literal value when it is bound to a variable of a different primitive type [4].

```
scala> 2
res3: Int = 2
scala> // Thus, the literal value 2 is recognized as an Int (by default)
scala> var myVar: Float = 2
myVar: Float = 2.0
scala> // Now, 2 is printed as 2.0 because it coerced from an Int to a Float
```

Scala implicitly coerces the Int 2 to the Float 2.0; but, as demonstrated in the previous section, implicit coercion from one type to another is not always possible depending on the two types and the direction of the coercion. Each primitive type may be implicitly coerced to a very specific set of other primitive types.

IX. Implicitly or Explicitly Typed

Scala is both implicitly and explicitly typed depending on how a variable is declared [4]. As the following code demonstrates, a variable’s type can be either explicitly defined or inferred.

```
scala> var myVar1: Float = 2.0
myVar1: Float = 2.0
scala> var myVar2 = 2.0
myVar2: Double = 2.0
```

X. Polymorphism

Scala implements parametric polymorphism [11-a].

XI. Overriding Methods

X. Garbage Collection and Memory Hazards

Unless

XI. Function Overloading

XII. Dangling Else

The dangling else is prevalent, but optional curly braces eliminate the problem as long as they are properly used [notes].

XIII. Regular Expressions

XV. Anonymous functions
Anonymous functions are a core part of the language.

XVI. Exceptions and Try-Catch Blocks
Technically speaking, Scala does not have built-in exceptions despite its ability to throw exceptions. How? Scala imports Java's exceptions by default [4]. The following code demonstrates throwing an exception.

```
try {
      throw new Exception
} catch {
      Exception => println("Exception caught")
} finally {
      println("try-catch completed");
}
```

XVII. Short Circuit Evaluation
Short circuit evaluation is implemented by default [13-a].

Works Cited:

1. Scala Documentation (http://docs.scala-lang.org)
    a. http://docs.scala-lang.org/tutorials/tour/tour-of-scala
    b. http://docs.scala-lang.org/tutorials/tour/mixin-class-composition.html
    c. http://scala-lang.org/documentation/getting-started.html?_ga=1.106534539.905291005.1473008976#your_first_lines_of_code
    d. http://docs.scala-lang.org/tutorials/tour/implicit-conversions
    e. http://docs.scala-lang.org/style/scaladoc.html
2. Tutorials Point (https://www.tutorialspoint.com)
    a. https://www.tutorialspoint.com/scala/scala_data_types.htm
3. CS Fundamentals
    a. http://cs-fundamentals.com/java-programming/java-primitive-data-types.php
4. Odersky, Martin et. al. Programming in Scala: A Comprehensive Step-by-Step Guide (2nd edition)
5. Artima Developer (http://www.artima.com)
    a. http://www.artima.com/scalazine/articles/goals_of_scala.html
    b. http://www.artima.com/scalazine/articles/origins_of_scala.html

6. Stack Exchange (http://softwareengineering.stackexchange.com)
    a. http://softwareengineering.stackexchange.com/questions/130335/why-is-scala-more-scalable-than-other-languages
7. Toptal (https://www.toptal.com)
    a. https://www.toptal.com/scala/why-should-i-learn-scala
8. Redmonk (http://redmonk.com)
    a. http://redmonk.com/sogrady/2016/07/20/language-rankings-6-16/
9. Java Documentation (https://docs.oracle.com/)
    a. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
    b. https://docs.oracle.com/javase/7/docs/api/java/lang/Float.html
10. Safari Books (https://www.safaribooksonline.com)
    a. https://www.safaribooksonline.com/library/view/scala-cookbook/9781449340292/ch11s12.html
11. Twitter.Github (https://twitter.github.io)
    a. https://twitter.github.io/scala_school/type-basics.html
12. Alvin Alexander (http://alvinalexander.com)
    a. http://alvinalexander.com/scala/whos-using-scala-akka-play-framework
13. Stonybrook (https://sites.google.com/a/stonybrook.edu)
    a. https://sites.google.com/a/stonybrook.edu/functional-programming-scala/lecture-1-4