SCALA

Hector Sanchez
sanchezh@.email.arizona.edu

Samantha Felzien
sfelzien@email.arizona.edu

April 30, 2020

The Scala Programming Language is an interesting and challenging new take on many aspects of a programming language that we have come to know and so dearly love; Java. With a more complex and rough-around-the-edges syntactical appearance that draws inspiration from other popular languages, Scala can take some getting used to, but once you do you'll be glad you did! In the following examination of the Scala programming language we'll observe some of the various key features that make Scala unique and what makes it a language you will definitely want to get your hands on.

1 Introduction

2 History

Scala was developed by Martin Oderksky in 2001 and first released in 2004. Scala was designed to fuse both functional and object oriented programming [3]. While Scala is not an extension of Java, it is interoperable and translates to Java bytecode [4]. The name is derived from the word scalable, meaning it can grow the demand of users. Scala is widely used today, as teams on Twitter, Apple Inc, The New York Times, Google, and Walmart Canada use it [3].

3 Control Structures

The Scala programming language is one that features all of the control structures we are familiar with from other more-common programming languages such as Java. Among these control structures we find the essential "if", "while", "for", and "case" structures. Contrary to the syntax found in languages such as Python but similar to Java, Scala requires the use of curly-braces to define the body and procedures within the control structure, such as the following:
while (true)

println ("this is true")

Much the same as Java, Scala also allows for the omission of these curly braces in certain structures if they are for functional use [7]. We often think about this structure in Java when we have a condition that must be met and a one-line instruction to follow if that condition is met, such as the example below implies. As you might imagine however, not all control structures can fit this simplified syntax, such as for loops and while loops [7]. In the case of for-loops, the omission of the curly braces is dependent on the inclusion of a "yield" clause in order to avoid infinite loops.

```
if (boolVal)println("bool was true")
```

Unlike Java however, there are some control structure elements that are not present in Scala. One such example is the Java ternary conditions, in which the syntax is made easy by the use of colons and question marks to essentially represent an if/else statement in one line of code. Scala doesn't fall far behind in this concept however, but rather offers a less-elegant but still very functional approach. Syntax in Scala is as follows:

```
val b = if (x) "x" else "y"
```

We can note the similarities in structure to other programming languages, but with Scala taking a more literal approach while also implementing the previously mentioned omission of the curly braces.

Despite this however, Scala does offer some very elegant and useful features in their control structure syntax, one of which being "for-comprehension" [8]. This syntactical notation allows for the traversal of multiple List elements within the for-loop declaration while also allowing for other conditionals to dismiss certain elements, returning only the desired elements using only two lines of code or so. In essence, Scala for-loop declarations simplify the foreach functionality and allows it to be applied to multiple arguments [8]. The code below demonstrates this feature with a list of numbers ranging from 1 to 10, in which the for loop is responsible not only for iterating through the list, but also picking out numbers greater than 5 and appending those numbers to a very different list.

```
val s = for (n <- nums if (n > 5)) yield n
```

When compiled, list s will return all numbers greater than 5, saving us additional lines of code or at least make our code a tad easier to read. This example is of the most simplistic nature, and this feature can be a very outstanding tool when iterating through lists of objects in which individual values or attributes must be checked for each object, since Scala allows for class fields to be references within if statements. [8]

Finally, we arrive at case conditionals, which are also embedded with these short syntactical shortcuts that allow for one-line code segments useful in more complex case statements that could otherwise be quite messy. At it's simplest form, the Scala case conditional is similar to those found in other languages, however Scala shines when creating more complex case-statements which may also require embedded conditionals or pattern matching. [7]

```
x match{

case a if (x.status == "bored") => findSomethingToDo(x)
case a if (x.status == "tired") => rest(x)
case _ => doNothing()
}
```

In the above code snippet we see the implementation of these if-conditionals within the case statements as well as the final pattern-matching case similar to that in Haskell. Such syntax is useful for creating elegant and easy to follow code that is often characteristic within the Scala programming language.

4 Data Types

Scala has the same memory types as data types as Java, with the same memory footprint and precision. However, there are no primitive types as in Java, so you can call methods on Int, Long, etc.

Any is the supertype of all types and defines some universal methods, such as equals, hashCode, and toString. Any has two subclasses, AnyVal and AnyRef. AnyVal represents the nine value types: Double, Float, Long, Int, Short, Byte, Char, Unit, and Boolean. The unfamiliar Unit is a value type which has no meaning or value. There is one instance of Unit, (). As all functions in Scala must return a value, Unit can be used for what would otherwise be a function with a void return type. AnyRef represents reference types, including non-value types. If Scala is used in a Java runtime environment, AnyRef corresponds to java.lang.Object.

Types can be based unidirectionally within their AnyVal or AnyRef distinction. Nothing is a subtype of all types and is thus the "bottom type". Nothing is commonly used to signal non-termination, such as a thrown exception or program exit. Null is a bottom subtype of any AnyRef subtype.

Scala, as with other JVM languages, has types that are erased on compile time. Scala uses Java's runtime reflection. Reflection is the ability to inspect an object at runtime. This includes inspection of generic types, instantiating new objects, and invoking members of said objects. Using TypeTag, an object x can carry the information from compile time to runtime. Using context bounds, TypeTag can be obtained. With Java's reflection, a type can be instantiated at runtime using invoking their constructor

5 Sub Programs

Scala has many useful subroutine features that may often seem familiar to more common languages such as Java or C. First, we start with what is regarded to as a trait in Scala. A trait in Scala is similar in appearance to a struct in C programming, however they are not similar beyond the appearance. A trait in Scala serves as a form of encapsulation for methods and fields, and unlike classes in Java which can only inherit from a single superclass, multiple traits can be inherited at once. Traits are somewhat similar to abstract classes in Java.

```
trait Derp
val phrase: String
def isDigit(x: Int): Boolean
def isNotDigit : (x: Int): Boolean =
!isDigit(x)
```

Above we can see the unique syntax of the Scala trait in which phrase simply defines a

feature of the trait Derp, while the following lines define two functions. A trait can be invoked within the function declaration by simply adding extends trait name to the end of the function. From here you can use the above defined "isDigit" method (for example) just as any other method extension available to you (i.e. val valWithinFunc = i.isDigit). Coming back around to the essential

subprogram features, we have the function. In Scala the function is much the same as functions in just about any other programming language, it requires a function declaration, a body, and some kind of invocation from somewhere else.

```
def addInt(a: Int, b: Int): Int =  var sum: Int = 0
sum = a + b
return sum
```

Here we have a simple declaration of a function that adds two integers. We can note the function declaration and its syntax is very familiar to us in the sense that we must declare our parameters and their types and what type of response we will send. Scala, allows us to name these incoming variables within this declaration for later use within the body. As for the declaration of what we will return, this can be found right after our parameters, in this case we have :Int, therefore we can expect and integer output. When left blank, this function will return nothing. As for calling a function it is the standard syntax (addInt(1,2), for this example). Finally, we come to the Scala class. In Scala class we find many useful features such as the ability to create a blueprint for objects and define a method within the class, thus allowing you to use this method as an extension of any object you implement of this class. Similar to the Java class, classes in Scala also have constructor declarations which must be set upon invoking a new object.

```
class Example(val x: Int, val y: Int)
var a: Int = x
var b: Int = y
def add(c: Int, d: Int)
a=a+c
b=b+d
println ("a: " + a);
println ("b: " + b);
```

Here we see the class “Example” in which we define the two parameters x and y, later assigned to a variable and furthermore used in the add function available when you implement this object. The “Example” object can later be created using the following syntax: val ex = new

Example(1, 2); From here we can then invoke the “add” method found within “Example” by saying ex.add(10,10);. Extending and inheriting from a class in Scala is also identical to Java since we must also use the extends keyword within our function declaration. Within Scala classes you can also create singleton objects and implicit classes by declaring them as such or following the necessary syntax

6 Who uses Scala and for what?

Stemming from the desire to better than the already existing and vastly-popular Java programming language, Scala as mentioned previously, was created in the early 2000’s but not officially released until 2004. Having said that, much of what makes Scala unique and for many a difficult language to learn is exactly what inspired the creation of it in the first place; a language designed for the JVM that tackled many of the undesirable aspects of Java. Scala has never really been known as

the easiest language to learn or even the first language you should encounter as you first learn to code, but what many find to be quite difficult to read or more difficult to understand is exactly what allows this language to be a more organized programming language. Scala provides the ability to often times minimize Java code into one more-complex but significantly shorter statement, saving substantial lines of code and even some complexity, while also teaching fundamental programming skills for immaculate structure and organization amongst other languages.

Having touched upon just a small subset of the many capabilities within Scala, we come to the inevitable question, who uses Scala and for what? While clearly not being a household name or industry standard like other languages, Scala is used by many individuals looking to hone their programming etiquette and large firms alike. For example, Scala is used by companies such as LinkedIn, Twitter, and FourSquare [6]to name a few. Being a fairly new language (respectively), Scala is still gaining support and a greater community following, as new IDE's and new frameworks are constantly emerging.

7 Summary

Overall, Scala was a fun programming language to explore. It had many quirks that often times made the learning curve a tad steep, but they weren't totally foreign or useless features, and often times added a cool factor to our project. Many of these unique features are highlighted within our project as we bake a loaf of sourdough, a fitting activity in these times of quarantine! Nonetheless, we witnessed first-hand how elegant Scala can be and even came to realize what the purpose behind Scala was, as we often found one-line solutions to what would've otherwise taken us multiple lines of Java or even Python code. In this, Scala adds an extra layer of elegance and skill to what we would've otherwise skipped over.

References

[1] Community.

[2] Documentation.

[3] History of scala - javatpoint.

[4] The scala programming language.

[5] Gavin Bisesi. Daenyth/taklib.

[6] Matt Hicks. Scala vs. java: Why should i learn scala?, Sep 2014.

[7] Heather Miller and Tyler Nienhouse. Control structures.

[8] Seth Tisue and Heather Miller. For comprehensions.