Parmveer Dayal (301282695)

CMPT 383 Assignment 2

July 7, 2023

Exploring Scala

Introduction

Scala (or Scalable Language), designed by Martin Odersky, is a multi-paradigm programming language [1]. Scala was created to combine both object-oriented programming (OOP) and functional programming into one language [2]. It is OOP in the sense that it fully supports classes and objects, allowing users to take advantage of concepts such as inheritance and polymorphism, while also, treating functions as values, which means they can be assigned to variables or returned as values like a functional programming language [1]. By combining both OOP and functional programming together, Scala allows developers to use familiar OOP concepts, while also leveraging functional programming techniques.

Scala was designed to improve upon the limitations of Java [3]. Issues present in Java, such as the heavy emphasis on mutable states, concurrency model being difficult to work with, or no built-in support for higher-order functions are overcome in Scala, while also being the much more concise language [2]. Scala allows for Java libraries to be incorporated into the code, so it not only benefits from its own libraries but also takes advantage of Java's much larger ecosystem [3]. This incorporation of the Java ecosystem allows for Scala to be compiled in Java bytecode and executed in the Java Virtual Machine (JVM) [3]. Scala is able to incorporate key Java concepts, while improving its shortcomings, in order to create a much more versatile programming language.

Scala is commonly used for machine learning, data science, or even finance applications [3]. Although, Scala is nowhere near the most popular language, ranking as the 20th language on the TIOBE index in 2018 [4], many big companies continue to program with it. Companies such as Twitter, Tesla, Apple, and Airbnb have incorporated Scala into their codebase in some way to improve services for their customers [5, 6]. It is worth mentioning, that many consider Scala to have a steep learning curve, which may be the biggest reason for why it is not more popular with developers. Both Twitter and LinkedIn stated they may decrease the amount of their code in Scala, due to the difficult learning curve for new team members [5]. Although, the programming language is very versatile and concise, there are not very many developers that are familiar with it, compared to more popular languages such as Java and C#.

Type System

Built-in Types

Scala offers a set of built-in types, where many of them are inherited from Java. As shown in figure 1, there are seven numeric data types: Char (2 bytes), Byte (1 byte), Short (2

bytes), Int (4 bytes), Long (8 bytes), Float (4 bytes), and Double (8 bytes). There are also two non-numeric datatypes: Boolean and Unit. These built-in types are identical to Java's, which allows for some familiarity. However, the difference is in Scala the types wrap around the primitive types and any operation on them is a function call [7]. All nine of these datatypes are considered direct subtypes of AnyVal, which is one of the two direct subtypes of the root class Any [2]. The other child of Any is AnyRef and is the superclass for all reference types, similar to Java's "Object" class [2]. It is responsible for objects, iterables, sequences, lists, strings, other Scala classes, other Java classes, and Null. Finally, as shown at the bottom of the hierarchy in figure 1, is the Nothing class, which is a subtype of every other type and is empty so no object of this type can be created. It is used for rare cases, where an expression or function does not produce a value or cannot be reached [2].

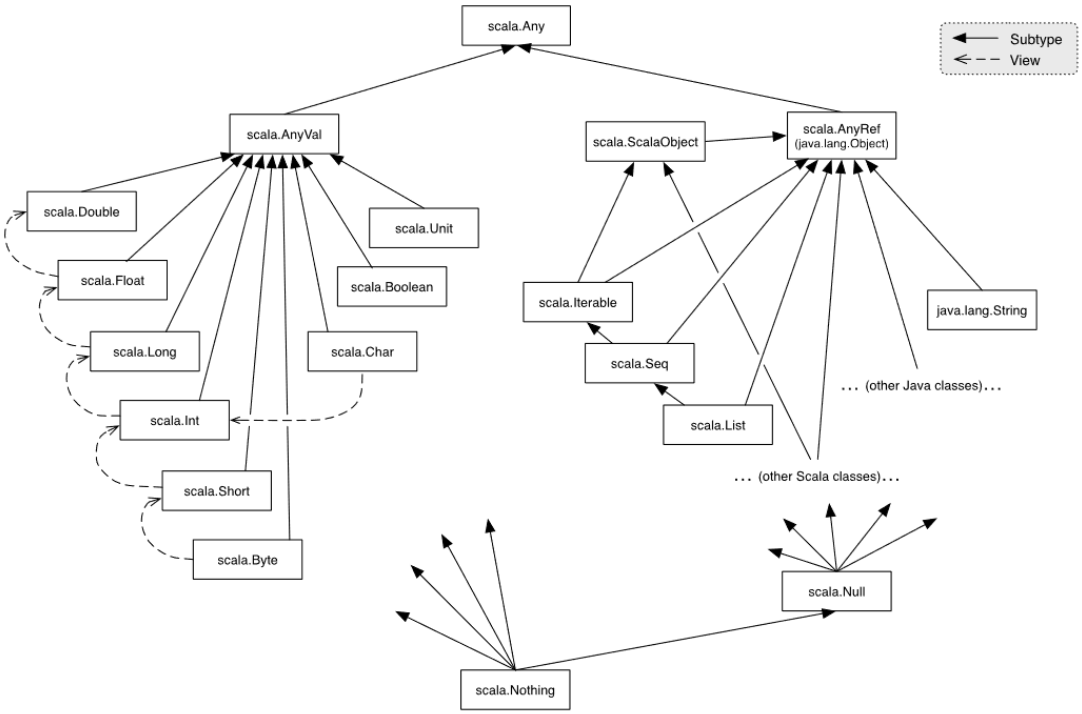

Figure 1: Scala Class Hierarchy. Adapted from [2]

Objects

Since Scala is an OOP language, it allows developers to take advantage of objects, similar to other OOP languages such as Python, Java, or C++. Creating objects in Scala is done with the keyword "object". Using this keyword creates a singleton instance of that object, which means the object can only be instantiated once [8]. If a programmer would like to create multiple instances, then they can define a class and use the "new" keyword create it. Figure 2 shows the differences between the "object" and "class" keywords, as only a singleton instance can be created with MyObject, and multiple instances can be created with MyClass.

```
object MyObject {
  def test(): Unit = {
    println("Testing!")
  }
}

class MyClass(name: String) {
  def hi(): Unit = {
    println(s"Hi, $name!")
  }
}

val obj = MyObject
obj.test()

val instance = new MyClass("John")
instance.hi()
val instance2 = new MyClass("Mike")
instance2.hi()
```

Figure 2: Object vs. Class.

A companion object in Scala is an object that shares the same name and source file as a class [9]. Since both the object and class are considered companions, they have access to each other's private members. This allows programmers to create multiple instances of certain classes without having to use the "new" keyword. Adding "apply" and "unapply" keywords in the companion object will construct and de-construct object instances respectively [9]. Figure 3, in the appendix, shows the close relationship between the companion objects and how it is a convenient way to define classes and objects in a cohesive manner.

In Scala, methods are values and from the previous section we know values are objects, therefore, methods can be treated as objects as well [2]. Figure 4 shows class MyClass has a method MyMethod and a main object which contains MyFunction that wraps MyMethod. By adding '_' after MyMethod, it has been converted to a function object, which allows the method reference to be stored in the MyFunction value and be called like a normal function. This is just a small example to showcase how methods can be treated as functional objects.

```
class MyClass {
  def myMethod(x: Int): Int = x * 2
}

object Main {
  def myFunction: Int => Int = (new MyClass).myMethod _

  def main(args: Array[String]): Unit = {
    val result: Int = myFunction(3)
    println(result)
  }
}
```

Figure 4: Methods as Objects.

Type System Features

Scala is strongly typed ensuring the operations are performed on compatible types, in order to prevent the risk of type-related errors [10]. It is also statically typed meaning the type checking occurs at compile time [10]. However, Scala's type system offers some flexibility with dynamic typing through type inference. This allows the compiler to type check automatically, making the code more concise [10]. Scala also supports more advanced type system features such as parametric polymorphism, higher-kinded types, pattern matching and implicit parameters [11, 12]. Overall, the flexibility and versatility of Scala's type system allows developers to solve complex problems with a concise codebase.

Memory Management

The memory management in Scala is very similar to Java's because it is primarily handled by the Java Virtual Machine (JVM) [3]. Thus, developers do not need to worry about freeing memory before the program exits because the garbage collector will automatically do it for them. Although, the garbage collector may take a longer time to free up code, it decreases the likelihood of memory leaks happening. It is the garbage collector's job to determine which

objects are reachable and eligible for garbage collection. The collector will start with the root objects and work its way down, marking all the reachable objects. Any object that was not marked, will be considered unreachable and be collected by the garbage collector [13]. There are various JVM garbage collection algorithms to optimize memory management and will be selected based on many factors such as the JVM configuration, heap size, application behavior and performance monitoring [14].

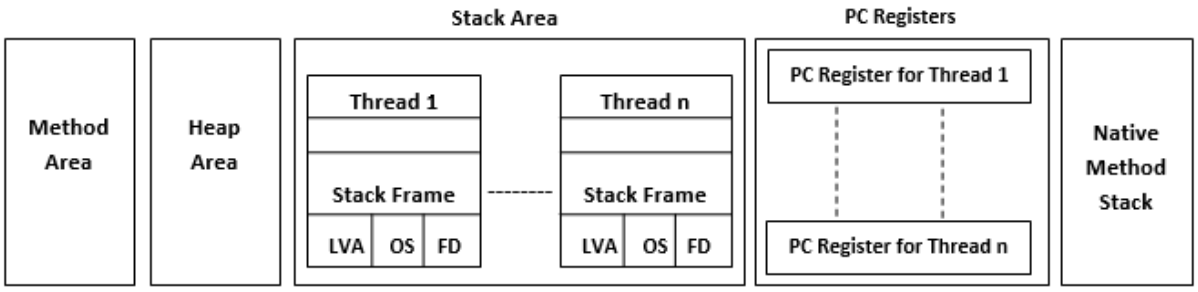

Figure 5: JVM Memory Structure. Adapted from [15]

JVM memory consists of a heap, stack and other area. The heap is where objects are allocated in Scala and is created when JVM starts up [13]. This dynamically allocated memory area grows and shrinks as objects are created and garbage collected. Various types of objects can be stored here such as instances of classes, arrays, and dynamically allocated objects. Also, the garbage collector only runs on heap memory and automatically deallocates any memory no longer in use [13]. The stack region is used to store local references such as local variables or method parameters since each thread will have its own stack. Each stack is last-in first-out (LIFO) and is faster to allocate and deallocate memory compared to the heap [13]. Finally, the other area stores all JVM specific code and data, and is made up of the method area and native memory [13]. The method area stores class-level structures, bytecode, and other metadata, while native memory stores implementations of methods written in other languages [15]. Figure 5 shows a diagram of the JVM memory structure.

Other Interesting Features

Scala seamlessly blends OOP and functional programming paradigms. It supports the creation of classes and objects like any other OOP language, which includes concepts such as inheritance and polymorphism [2]. However, unlike a traditional OOP language, it also provides support for functional programming features such as lambda expressions and tail recursion [2]. With functional programming, Scala can take advantage of higher-order functions by treating functions as first-class citizens, allowing them to be assigned to variables, passed as arguments, and returned as results [16]. Higher-order functions enable other functional programming techniques

```
def add(a: Int, b: Int, c: Int): Int = a + b + c

val addTwo = add(_: Int, 2, _: Int)
val result = addTwo(1, 3)
println(result)
```

Figure 6: Partially Applied Function.

like currying and function composition. Functional programming also allows programmers to take advantage of partially applied functions, which are functions that can have some arguments fixed, while others remain unfixed [16]. As shown in figure 6, the second argument is fixed and in the very next line, the partially applied function is called without having to enter the second argument. This creates additional flexibility and makes functions easier to work with.

Scala's compiler is really good at inferring the types of variables, expressions, and function return values due to type inference [17]. There is no need to write what type of datatype a variable will hold because the compiler will automatically figure it out. The same works for methods as well. Figure 7, in the appendix, shows a function that will multiply two numbers and return them. There is no need to specify what datatype should be returned because the compiler will figure it out and know two integers being multiplied should return an integer. This is another way that Scala allows programmers to write code without having to explicitly specify types in most cases.

Pattern matching in Scala is another interesting feature because it is really good at checking for values against a pattern and is a more powerful version of the "switch" keyword in Java [18]. It allows programmers to do tasks such as deconstructing data structures and handling different cases in a concise way. It is also very powerful for case classes, which are a special type of classes that are created for pattern matching [18]. As shown in figure 8, a case class Person is created, then an instance of Person is created and uses pattern matching to match against it. This helps to simplify the code and write it in a more concise manner.

```
case class Person(name: String, age: Int)

val person: Person = Person("John", 50)

person match {
  case Person(name, age) => println(s"Name: $name, Age: $age")
  case _ => println("Unknown person")
}
```

Figure 8: Case Classes.

Comparison with Other Languages

As stated many times before, Scala is not only very similar to Java, but was also created to overcome some of Java's shortcomings [2]. Scala compiles to Java bytecode, since the programming language runs on JVM just like Java. This makes it possible for Scala to run on any platform that is implemented with JVM, making it compatible with the Java ecosystem [2]. This also allows Scala to seamlessly work with Java code, allowing programs to call Java libraries and frameworks. JVM also ensures both language's memory is collected with a garbage collector, so there is no need for programmers to worry about deallocating their memory. Both languages support OOP, to allow programmers to develop classes and objects. Both languages are also statically typed, ensuring type safety at compile time [19].

However, there are some key differences between the two languages as well. Scala incorporates functional programming, which allows for type inference, pattern matching, and higher-order functions, to go along with a much more concise codebase. Scala also has additional features such as operator overloading and lazy evaluation, while Java does not. Scala incorporated an "Option" type for absent values, as a way to avoid the issue in Java where null values can lead to null pointer exceptions if not handled correctly [2]. Finally, Scala provides much better built-in support for concurrent and parallel programming through libraries such as Akka [2]. The strengths of Scala being a much more versatile language can also be viewed as a weakness at times, due to the programming language having a more difficult learning curve in comparison to Java. New programmers are much more likely to learn Java compared to Scala.

Scala shares some similarities to Haskell, since they are both functional programming languages. Just like Haskell, Scala is able to take advantage of concepts such as immutable data structures, type inference, currying, and lambda expressions. However, a key difference between the two languages is Haskell is much better with its use of monads to handle side effects compared to Scala [20]. Of course, Scala is much more than just a functional programming language and does more than Haskell, but the functional programming between the two is very similar.

Scala also shares some similarities to other languages as well. Scala took inspiration from C#; that is why concepts such as properties, implicit conversations, and methods closely resemble those in C# [2]. Scala's constructs for defining classes and methods, or working with collections are very similarly to those in Ruby [21]. Python can be seen as an inspiration for Scala when it comes to the conciseness and readability of the code, along with the collection of APIs. Scala also provides a Read-Eval-Print Loop (REPL) similar to the one in Python for experimentation and interactive coding [22].

Conclusion

Scala's ability to combine both OOP and functional programming gives programmers the flexibility to solve problems in more than one way. Now, programmers can take advantage of concepts such as inheritance and polymorphism, while still being able to treat functions as values and solve problems declaratively. Scala essentially incorporates Java and Haskell into one language, while still sharing similarities with other languages like C#, Python, and Ruby. These similarities with much more popular languages, allows programmers to be familiar with many Scala concepts before even learning Scala. Scala also being very closely related to Java means Java classes and libraries can be easily incorporated into the codebase, providing further flexibility for programmers.

Due to Scala incorporating so many different and complex concepts, it has a much steeper learning curve than languages such as Java or Haskell, for example. Scala is difficult to learn even for those with previous experience in Java, due to the different keywords and incorporation of functional programming. Although, it will take some time to learn the language, it is well worth the time. Scala offers flexibility and an abundance of complex concepts written in

a clean and concise manner, like very few programming languages can. Learning Scala after Java or C# is a great investment and way for programmers to learn new concepts and continue to evolve, even if they end up not programming in Scala again.

Appendix

```
class Person:
  var name = ""
  var age = 0
  override def toString = s"$name is $age years old"

object Person:

  // a one-arg factory method
  def apply(name: String): Person =
    var p = new Person
    p.name = name
    p

  // a two-arg factory method
  def apply(name: String, age: Int): Person =
    var p = new Person
    p.name = name
    p.age = age
    p

end Person

val joe = Person("Joe")
val fred = Person("Fred", 29)
```

Figure 3: Companion Objects. Adapted from [9]

```
def multiply(x: Int) = x * x
def fac(n: Int) = if n == 0 then 1 else n * fac(n - 1)
```

Figure 7: Type Inference.

References:

[1] “Introduction,” *Scala Documentation*. https://docs.scala-lang.org/tour/tour-of-scala.html
[2] M. Odersky *et al.*, “An Overview of the Scala Programming Language Second Edition.” Available: https://www.scala-lang.org/docu/files/ScalaOverview.pdf
[3] A. Fawcett, “Scala 101: A beginner’s guide to the scalable language,” *Educative: Interactive Courses for Software Developers*, Nov. 20, 2019. https://www.educative.io/blog/scala-101-a-beginners-guide
[4] “TIOBE Index,” *TIOBE*. https://www.tiobe.com/tiobe-index/scala/ (accessed Jul. 07, 2023).
[5] K. Greene, “The Secret Behind Twitter’s Growth,” *MIT Technology Review*, Apr. 09, 2009. https://www.technologyreview.com/2009/04/01/214481/the-secret-behind-twitters-growth/
[6] “Tesla Virtual Power Plant,” *InfoQ*. https://www.infoq.com/presentations/tesla-vpp/ (accessed Jul. 07, 2023).
[7] “Guide to Data Types in Scala,” *Baeldung*, May 20, 2022. https://www.baeldung.com/scala/data-types
[8] “Classes and Objects in Scala,” *Baeldung*, Oct. 11, 2022. https://www.baeldung.com/scala/classes-objects
[9] “Companion Objects,” *Scala Documentation*. https://docs.scala-lang.org/overviews/scala-book/companion-objects.html (accessed Jul. 07, 2023).
[10] “Heather Miller,” *heather.miller.am*. https://heather.miller.am/blog/types-in-scala.html (accessed Jul. 07, 2023).
[11] Baeldung, “Polymorphism in Scala | Baeldung on Scala,” *www.baeldung.com*, Aug. 20, 2020. https://www.baeldung.com/scala/polymorphism
[12] “Higher-Kinded Types in Dotty,” *dotty.epfl.ch*. https://dotty.epfl.ch/docs/internals/higher-kinded-v2.html (accessed Jul. 07, 2023).
[13] J. Dhiman, “Decoding Scala without the Code,” *Medium*, Nov. 09, 2020. https://towardsdatascience.com/decoding-scala-without-the-code-6db00f37c469
[14] L. Gupta, “Java Garbage Collection Algorithms [till Java 9],” *HowToDoInJava*, Apr. 12, 2018. https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/#mark-sweep (accessed Jul. 07, 2023).
[15] “Memory Management in Java - Javatpoint,” *www.javatpoint.com*, 2011. https://www.javatpoint.com/memory-management-in-java
[16] “Introduction to Functional Programming in Scala,” *Baeldung*, Mar. 08, 2021. https://www.baeldung.com/scala/functional-programming (accessed Jul. 07, 2023).
[17] “Type Inference,” *Scala Documentation*. https://docs.scala-lang.org/tour/type-inference.html (accessed Jul. 07, 2023).
[18] “Pattern Matching,” *Scala Documentation*. https://docs.scala-lang.org/tour/pattern-matching.html (accessed Jul. 07, 2023).
[19] “Scala Vs Java | Comparison Between Scala And Java,” *www.knowledgehut.com*, Jun. 14, 2023. https://www.knowledgehut.com/blog/programming/scala-vs-java (accessed Jul. 07, 2023).
[20] S. Louc, “Demystifying the Monad in Scala,” *We’ve moved to freeCodeCamp.org/news*, Oct. 22, 2019. https://medium.com/free-code-camp/demystifying-the-monad-in-scala-cc716bb6f534 (accessed Jul. 07, 2023).

[21] “Ruby under the influence [of Scala],” *thoughtbot*, Mar. 27, 2019. https://thoughtbot.com/blog/ruby-under-the-influence-of-scala (accessed Jul. 07, 2023).
[22] “Concurrency,” *Scala Documentation*. https://docs.scala-lang.org/scala3/book/concurrency.html (accessed Jul. 07, 2023).