Apache Groovy Programming Language

**Tristan Lotivio**

Middle Tennessee State University

Murfreesboro, Tennesee  
tcl3c@mtmail.mtsu.edu

# INTRODUCTION & HISTORY

* 1. **Introduction & Overview**

The Apache Groovy Programming Language is a dynamic and powerful language based off the Java Virtual Machine (JVM), which began production in the early 2000’s [5]. The language sees a lot of use in the realms of Web Development as the basis of many frameworks, app development as it is a supported language by the Android Software Development Kit, all the way to simple scripting and automation. Evidently, this decision to be based off the JVM, also opens the language to be Object Oriented, but this philosophy to coding is not the only one, present with the language, as one can even code without using classes entirely in the language [6].

This speaks to the philosophy of the language which is simply to design a language that is accessible and flexible, such that an experienced programmer, and novice can find comfort and use in the language. This level of flexibility was equally inspired by the now Python and Ruby languages with their respective usages of concepts such as dynamic type checking and metaprogramming, both of which are available tools in Apache Groovy. Along with this there are also multiple ways to do a multitude of tasks lending to the beginner friendly aspect of the language, which can be both good and bad, as it might entice a new programmer into programming though, render him/her ignorant to the machinations of computer science, as Groovy can hide meanings that can be derived in more classical structures and instructions. This along with the sporadic popularity can build a mediocre case against learning the language, but it doesn’t excuse the languages capabilities to appease multiple demographic in the computing community with general ease.

* 1. **Timeline of Groovy**

Apache Groovy was originally invented in 2003 by Bob McWhirter & James Strachan with intentions of developing a language that converts well into bytecode, plays well with other Java-based programs, but is also malleable such that another language doesn’t have to be invoked to do scripting or automation tasks [5]. After a year, the language was commissioned to be added to the next update for Java, as Java Specification Request (JSR) 241, however at the time both McWhirter and Strachan stepped down as project leads[5]. Development of the language was then restarted the next year under the leadership of Jeremy Rayner & Guillaume Laforge, inspired by the idea and determined to see the rest of the language come to fruition [5]. Apache Groovy version 1.0 released in ’07 and has since received growing and steady support throughout the years. Main differences from the Java included the following:

**Figure 1.2.1 [14]**

* Variables within a string can be flagged to be substituted using the ‘$’ token.
* Getters and Setters are automatically set for data members within an Apache Groovy class.
* Quality of life changes were added to make the language more concise (i.e. System.out.println() vs. println()).
* Data members of a class are immediately accessed without having to call a getter or setter, through the ‘.’ operator.

In 2014, with the release of Apache Groovy 2.3, support for the language was bolstered when Apache Groovy was officially supported with the then current JDK 8. The most recent sign of support to the language comes with its addition to the Apache Software Foundation in 2015. Currently, the stable version of the language is at 3.0.3.

**1.3 Current Use of Apache Groovy**

Groovy sees a great portion of its use as a web development language. While it is strong as a beginner’s tool to learning coding, it still remains a language that is compiled as bytecode, therefore any new framework developer who might be looking into programming languages that he or she can work on, has Groovy available to them just as they would able to select any other. Because of this, the language sees use on many companies’ applications, including but not limited to:

**Figure 1.3.1 [11]**

* National Cancer Institute
* French Nuclear Safety Organism
* JPMorgan
* MasterCard
* Netflix
* LinkedIn
* ETC…

Virtually all of these are used in Web Framework environments that use Groovy as it it’s main language, such as Grails, Gaelyk, Glide, and Spring-Boot [1]. Lastly, it is still important to consider that the language as last reported by Paul King on his blog, that the Apache Groovy which is annually sees 12 million downloads a year as of 2016 [12], meaning that the language sees a lot of use outside of these companies in smaller/open source projects throughout the world.

# PHILOSOPHY

* 1. **Basic Theory**

The general theory is Apache Groovy is that of accessibility, in that it intends to as accessible. Java while powerful in context of what the language can do, is also strictly structured syntactically which then begs the question of how one might be able to break the mold of Java’s more confining features while keeping all the parts of it that make it great, such as it’s bytecode formatting, plentiful library.

* 1. **Differences from Java**

The best way to understand the perceived necessity is to analyze the differences between the Java and Groovy further. At the core of Groovy, the idea existed to simply improve on what the language lacked and nothing more. This allows for the language to be close and accessible to Java, as it also made the development of the language equally interesting. When changes were made to the native Java language which were in favor of the goal of Apache Groovy, the changes were simply integrated with the language. Since the goal of the language is to simply improve on the shortcoming of Java, there is no reason to reinvent the wheel. This decision process leaves Groovy language developers simply to work on the any new issues that their own language might incur, as well as new problems that Java might incur.

* 1. **Choice in Instructions**

To analyze the design flexibility found in Apache Groovy, we can use the for-loop in its more common iterations in both Groovy and Java, to understand more of the design philosophy by sheer volume of instructions and from that intention.

In Java the two more common styles of the for loop are as follows:

**Figure 2.3.1 [10]**

* for(stmt1;stmt2;stmt3){/\*do stuff\*/}
* for(type item:arrName){/\*do stuff\*/}

Firstly, we have a very classical format of for loop which though clunky explains and reads simply enough with time in the following statement. Where some value starts at index 0, while that value is less than some other value, add by one until preceding statement is false. The beauty of this for loop is in its simplicity as it the entire instruction is laid bare for the programmer to use. Any adjustments to the counter can be made, other datatypes can be made in place, and the Boolean expression in the second statement can be modified to fit whatever needs some arbitrary program might require.

The second is a more simplified version which is specific to array operations. Often if a programmer wished to iterate and operate on a list, they could use the counter declared in the for loop the process each element individually, until the end of list is reached. This is possible with the first format however an interesting thing to note is that the code can become noticeably superfluous as more and more of the same instruction is written to complete similar tasks. It is in the transition from the first form to the second form that we see a spark of the same design philosophy in Groovy, as the English format of the second iteration is the following. At datatype some item in list arrName, do the following.

One can see that a lot of the fat from the previous statement has been trimmed as the new overloaded format simply just needs a place holder for the element being processed at that point in the loop as well as the array. We don’t need anymore information available to us, as we the length is determined as is generally the step length which, can be further modified within the loop if need be.

Apache Groovy expands on this idea of necessity to give a multitude of ways to handle the same type of information. By limiting the length of an instruction, and simultaneously handling how the compiler or the instruction might handle the instruction, we can see that though superfluous can save time on a lot of the iteration needed for each. Following are some common forms of lists that are found in Groovy:

**Figure 2.3.2 [4 , 9]**

* 3.times {//it for counter}
* 0.step 3, 1 {//it for counter}
* 0.upto(4) {//it for counter}
* lstName.each {//it for element}
* lstName.eachWithIndex{val , idx -> //val for element & idx for counter}

To generally analyze the structure of each instruction one can essentially see that due to the inherently object oriented nature of Apache Groovy, values of any datatype natively are objects, of which certain Groovy’s dot operator flexibility is utilized such that methods available to that datatype’s class, allowing the programmer to seamlessly use any given value to perform a task. A good example of this can be found with the Integer datatypes as they have access to the ‘times {}’, ‘.step max,idx {}’, and ‘.upto(x) {}’ methods which all have optimal functions based on the scenario that the programmer is dealing with. The same is present for lists, found with the ‘.each {}’ and ‘.eachWithIndex {ele, idx -> }’methods as it very commonly used to access the counter, but should the programmer need access to certain elements of the list, the last form could be utilized such that ‘val’ can be used to perform other more precise operations.

This goes without saying however, that the same classical forms of the for loop are also available, which final addresses better the power of choice within the language. Any experienced programmer should they not like the given solution or instruction present to them, often has and makes the choice to instead use what is simple and easiest to him or her. This often takes time a way from solving the problem at large. Through the example of the large quantity of ways to iterate and use the concept of the for loop, the Apache Groovy is able to present the idea such that here is a large quantity of items that are made present here and now, that will be able to solve a multitude of problem. Essentially, the more one learns groovy, the less one time one spends finding the exact solution to a problem, and more time they spend simply using what is most efficient in the language to continue solving whatever bigger problem they might be dealing with.

# INFLUENCE

Like all languages, Apache Groovy Particularly stands on the shoulders of many significant programming languages. Outside of the language obviously being based on Java, the necessity for Groovy to exist as a separate language stems from the present strengths of other languages in an attempt to combine them with the same strengths of Java, thus creating Groovy. The most obvious imported strengths are demonstrated in Groovy in the following lines of code:

**Figure 3.0.1[8]**

1. def widen ={src->
2. def newStr =””
3. for(c in src){newStr+=c+” ”}
4. return newStr
5. }
6. println widen(“test”)

output: t e s t

* 1. **Java**

Evidently, the very first language that should be noted is Java in that Java is the very language being improved upon. What is occurring above is possible in Java, in that the output is being modified by function widen at runtime, but the way that it is occurring simply is not. For one, at line one we have the ‘def’ operator attached to widen, which denotes that dynamic typing is being used, which is not legal in java, in this format. We also have on top of that closures for the method which resembles a function call in its implementation and declaration throughout the program, however this is also not the case, as this ‘widen’ is being called handled at runtime, which denotes that metaprogramming is occurring. Understandably this is a very simplistic implementation of the possibilities present for this concept, as in theory one could add classes, and methods to a language at runtime, which is a very common problem in Web Development tasks. It is in this that we see where Java falls short and the influences of other languages fall short.

* 1. **Python**

From Python, we see the first and most obvious influence in the form of dynamic typing. Dynamic typing revolutionized the accessibility and flexibility of the Groovy, such that specifically in Java as a language cannot be interpreted and compiled at runtime. This can be considered a good quality from the position of speed, and knowledge of that arbitrary software, it is a system that does not account for malleability. Should a programmer need to be able handled varying datatypes quickly and effectively without overly verbose code, the ability for the program to process and handle that unknown data is required, and subsequently unavailable in Java, and available in Python . Consider the following small examples:

**Figure 3.2.1 [15]**

1. def x = 2
2. println x
3. x = “2”
4. println x

**Figure 3.2.2**

1. int x = 2
2. println x
3. x = “2”
4. println x

Above a simple example is presented Groovy plays well with both dynamic and static typing due to the nature of the simple tasks being executed [2]. Above is the usage of ‘x’ as a dynamic data type at line which then allows for the Groovy compiler to be able to handle and reassign the datatype as necessary, thus printing both forms of ‘2’ in lines 2 and 4.

In the second code block we see equally viable, Groovy code with the under the conditions that the code compiles into bytecode properly, with the obvious error that in line 1 ‘x’ is statically typed, as an int datatype. This isn’t natively a problem, as Groovy intends to allow for this to occur, thus the code compiles and executes until the interpreter sees the datatype mismatch and then causes the error. Regardless this also shows that a great deal of flexibility is made available to a Groovy Programmer as they have the ability to do have both dynamic and statically typed variables for any task.

* 1. **Ruby**

Ruby’s influence on Groovy can be seen in the presence of metaprogramming in the language [13]. The best part of this concept can be seen in Figure 3.0.1 from lines 2,6 where ‘src’ pointer is created in order to create a new method which is handled at runtime of the actual language. While simple in this example consider its application in web development. A website that has a button object, that when clicked triggers an action, that pulls up another window, to display more information. Without metaprogramming, this action would all have be entered and compiled with that source file, and in some case might not be possible, as instances can occur where items are not being currently being executed on. However, because of metaprogramming, new information can be seamlessly compiled and handled. This is what makes Ruby such a powerful language in frameworks like Ruby on Rails, and what makes Grails a worthy adversary for those who wish to use Groovy instead.

# CRITICISM

Many of the criticisms directed at Groovy are more or less drawback of the language that are intended solely in that these topics are more or less not the focus Groovy as a programming language.

**4.1 Choice Overload**

Ambiguity in the choice of the amount of instructions and methods is can be seen as a criticism, in terms that new and experienced programmers will become placated by these shortcuts, and might be shocked to see that the computer might not be handling their loop, or their program the way they assumed it might have been. For an experienced programmer who understands what they are getting into by using a method shortcut is assumedly ok as they should understand the context of their instruction.

**4.2 Speed**

Speed is also another great example of a pitfall of Groovy. Because of dynamic typing, Groovy does gain issues in terms of the speed of the language in comparison to Java, though this might later be an unimportant factor in the language [7]. Often, the efficiency of a language only matters for select cases, which often aren’t the case for web development tasks. Nevertheless, it doesn’t change the drawback of lag which can be felt on smaller programs which can take seconds to compile simple programs that do nothing more than arithmetic, input and output, during compilation.

# OBJECT ORIENTED PROGRAMMING

One of the strong suits of Groovy, is that the language retains a lot of the same strengths in terms of object orientation that are present in Java. Due to the modularity of class manipulation in Java, all being preconceived strong suits of the language, all these tools and even the program design in Java are available in Groovy for anyone to use. Better yet, all the same rules with Object Orientation are even made easier for less experienced programmers as getters and setters are defined automatically, and Groovy also allows for a lot of flexibility once again with the ‘def’ and ‘.’ datatype and operator respectively. This not only allows for objects to change variable names, but also allows for the same new benefits of Groovy to have an impact on Java. Consider the following code:

**Figure 5.0.1 [3]**

1. class Student {
2. int StudentID;
3. String StudentName;
5. int Marks1;
6. int Marks2;
7. int Marks3;
9. int Total() {
10. return Marks1+Marks2+Marks3;
11. }
13. static void main(String[] args) {
14. Student st = new Student();
15. st.StudentID = 1;
16. st.StudentName = "Joe";
18. st.Marks1 = 10;
19. st.Marks2 = 20;
20. st.Marks3 = 30;
21. }
22. }
24. println(st.Total());

The most interesting part of the code snippet in Figure 5.0.1, is in the simplicity of the value setting without getters and setters. Because of the naturally malleable nature of Groovy, this code works without consequence, and strictly from a development standpoint is better for it, because of that same freedom that the language provides. Groovy is freedom as a programming language.

# REFERENCES

1. Anon. 2015. 6 Web Frameworks for Groovy. (November 2015). Retrieved April 19, 2020 from <https://codecondo.com/6-web-frameworks-for-groovy/>
2. Anon. 2016. Dynamic typing vs. static typing. (October 2016). Retrieved April 19, 2020 from <https://docs.oracle.com/cd/E93962_01/bigData.Doc/studio_users_onPrem/src/csu_transform_typing.html>
3. Anon. Groovy - Object Oriented. Retrieved April 19, 2020 from <https://www.tutorialspoint.com/groovy/groovy_object_oriented.htm>
4. Anon. 2015. Groovy For Loop Examples - Grails Cookbook. (April 2015). Retrieved April 19, 2020 from <http://grails.asia/groovy-for-loop-examples>
5. Anon. Groovy Script Tutorial for Beginners. Retrieved April 19, 2020 from <https://www.guru99.com/groovy-tutorial.html>
6. Paul King. 2017. Groovy, an Open Source Success Story. (January 2017). Retrieved April 19, 2020 from <https://www.sitepoint.com/groovy-open-source-success-story/>
7. Jonathan Gros-Dubois. 2020. Statically typed vs dynamically typed languages. (April 2020). Retrieved April 19, 2020 from <https://hackernoon.com/statically-typed-vs-dynamically-typed-languages-e4778e1ca55>
8. Intertech. 2013. Groovy Metaprogramming. (May 2013). Retrieved April 19, 2020 from <https://www.intertech.com/Blog/groovy-metaprogramming/>
9. Sunil Prakash Inteti. 2009. Why and When to use Groovy. (January 2009). Retrieved April 19, 2020 from <https://xebia.com/blog/why-and-when-to-use-groovy/>
10. Anon.Retrieved April 19, 2020 from <https://www.w3schools.com/java/java_for_loop.asp>
11. Guillaume Laforge. 2013. Who's using Groovy in production? (October 2013). Retrieved April 19, 2020 from <https://www.quora.com/Whos-using-Groovy-in-production/answer/Guillaume-Laforge>
12. Guillaume Laforge. 2014. Groovy crosses the 3 million downloads a year mark. (January 2014). Retrieved April 19, 2020 from <https://glaforge.appspot.com/article/groovy-crosses-the-3-million-downloads-a-year-mark>
13. Dean Nasseri. 2015. Ruby Metaprogramming by Example. (December 2015). Retrieved April 19, 2020 from <https://buildingvts.com/ruby-metaprogramming-by-example-612526d0b72>
14. Pratik Kolapkar 2019. Difference between Groovy and Java. (August 2019). Retrieved April 19, 2020 from <https://www.geeksforgeeks.org/difference-between-groovy-and-java/>
15. Anon. 2012. Static vs. dynamic typing of programming languages. (April 2012). Retrieved April 19, 2020 from https://pythonconquerstheuniverse.wordpress.com/2009/10/03/static-vs-dynamic-typing-of-programming-languages/