Utrecht University

Concepts of Programming Language Design 2021-2022
INFOMCPD

Language Report
Apache Groovy Programming Language

Author:
Athanasios Tsiamis
Student Nr. 5223652
February 22, 2022

a course supervised by:
Prof. Dr. Gabriele Keller &
Tom Smelding

Figure 1: Apache Groovy's logo

Introduction

Apache Groovy is a Java-syntax-compatible object-oriented programming language for the Java platform. Groovy includes features found in Python, Ruby and Smalltalk, but uses syntax similar to the Java programming language making it the ideal choice for many Java programmers. It contains powerful features such as closures, functional programming and type inference and it is supported by the Apache Software Foundation & the Groovy community [1].

A brief history

Although in 2003 Java dominated the market with 24% according to the TIOBE index [2], Java wasn't the desired language for various scenarios such as scripting, applying an urgent patch or just programming without boilerplate code. A new language was needed which should maintain a Java-like structure while also reducing the complex and often unnecessary syntax imposed by Java.

James Strachan can be considered the creator and original envisionist of Groovy. He first wrote about the development on his blog in August of 2003. He quoted *"I've wanted to use a cool dynamically typed scripting language specifically for the Java platform for a little while.*[..]*I'd rather a dynamic language that builds right on top of all the groovy Java code out there & the JVM."*[3]

Hence the development of this language started and in March 2004 it was formally submitted to the Java Community Process as a Java Specification Request (JSR) Nr.241 where it was unanimously accepted by ballot [4]. However, this approval was the beginning of a difficult process for the Groovy community. For many, development took a turn towards the wrong direction and imposed a bigger workload than they were willing to do. James Strachan silently stepped down from his managerial position and Guillaume Laforge took his position [5]. Finally on January of 2007 Groovy 1.0 was released [6] [7].

In July 2012 Groovy 2.0 was released in which groovy code could be compiled statically offering type inference and performance near that of Java [8].

As of today, Groovy 3.0.9 is the latest stable release which includes a brand new parser (codenamed *Parrot*) and adds, among other things, as many as 80 extension methods to existing Java classes. Meanwhile Groovy 4.0, which is still in beta, adds powerful switch expressions and sealed types which restrict other classes or interfaces interacting with them.

Groovy is gaining more ground on various applications not only as a complementary language to a Java project but also as a standalone language in a project of its own(see section 4).

1 Main features

1.1 Groovy 2.0 - Static features in a dynamic language

As mentioned before, Groovy was created to be the dynamic supplementary of Java. Developers fluent in Java could switch back and forth between the two languages seamlessly and use the appropriate type system when needed. However, as time progressed, Groovy developers fancied Groovy's syntax more and more and preferred to use only one language (instead of two). The issue though was that they wanted to have a safety net for errors which only the static type check could provide.

Groovy's development team was struggling with this change and was hesitant to offer static type checking. Feedback was mixed with some supporting that a shift towards static typing would cause Groovy to be a "*failed Java 8 wannabe*" [9] while others strongly supported type checking even if that meant that Groovy would lose some of its dynamic features [10].

While this topic was still controversial, it was clear that the other languages were progressing while Groovy was still lacking behind; Kotlin (a static type Java like language) had been just unveiled by the JetBrains team while Java was preparing the next release [11][12]. Immediate action was needed if Groovy would to be kept as an alive programming language.

Using the right concept was no easy task for the Groovy development team. According to King (Groovy's chair PMC) various ideas were considered for combining static and dynamic typing [5]. One proposal would be *gradual typing* based on the work of *Siek* and *Taha* [13]. The idea behind that would be that variables declared with a type would be statically checked while others would remain dynamic. According to King, while this idea was favourable to them [5], it would break existing codebase. Hence, the concept of gradual typing was rejected.

Another option, as King suggested, would be a special switch that would set the compiler into static mode in which all code would be statically typed [5]. However, this option felt too crude as many existing codebases were already making use of its dynamic features in some parts and not use it at all in others. Enforcing this binary project-wide switch did not bide well with the developers. Therefore, the compiler switches were rejected as well.

The solution came eventually from a Java style annotation[1]. A flexible alternative to the compiler being in one of two modes was to indicate at the method level whether code within that method was static or dynamic. A class could then be annotated (if needed) as a shorthand for annotating all methods. For example a static class could have one or more dynamic methods and a dynamic class one or more static methods. Thus two new annotations were created: *@TypeChecked* [15] which methods binded with this would go through the Meta Object Protocol (MOP) and *@CompileStatic* in which all of the MOP would be bypassed [16]. More about metaprogramming can be found in section 3.

1.2 The Groovy way - Syntax style and new features

Groovy adds various features to the Java Virtual Machine ecosystem. Groovy's vibrant and rich open source community has been a factory of new ideas. The following section provides a short overview of some important new features of the language.[2]

[1]Java annotations, a form of metadata, consist of an at-sign (@) and provide data about a program that is not part of the program itself [14].

[2]The list is not complete as it only contains a number of features the author found interesting. The full list of features can be found at https://groovy-lang.org/style-guide.html

1.2.1 Semicolons and return keywords are optional

Even though Groovy uses the JVM and Java code can be used directly into a groovy program, semicolons are optional. The same principle applies for the *return* keyword as well. However, for short methods and for closures, it's nicer to omit it for brevity. The code in section 1.2.5 provides a simple example of this.

1.2.2 Optional Typing - Def keyword

One of the most important keyword in the Groovy programming language is the *def* keyword. The def keyword is used to define an untyped variable or a function in Groovy, as it is an optionally-typed language. It can make the life of a developer easier and drive up the productivity by reducing the need to specifically think about the variable needed. Instead they can just declare it as "def" and let the compiler infer its type. (See line 1 at section 1.2.5)

1.2.3 Groovy Strings (GStrings)

One of the most cumbersome tasks in Java can be string manipulation in which users have to either use external libraries or complex notation with the help of "+" sign. Groovy reduces this need by introducing *Groovy Strings* (GStrings). Gstrings represent a String which contains embedded values such as:

```
"hello there ${user} how are you?"
```

which can be evaluated lazily.

Advanced users can iterate over the text and values to perform special processing, such as for executing SQL operations, where the values can be substituted for ? and the actual value objects can be bound to a JDBC statement[3][18]. For example:

```
def firstName = "Thanos"
def lastName = "Tsiamis"
sql.execute("INSERT INTO contactBook (firstName, lastName)"+"values (?,?)", [
    firstName, lastName])
```

A more simple example of the case of GStrings can be found at section 1.2.5

1.2.4 Switch statement

Groovy's switch statement accepts almost all types simultaneously making it a very powerful statement in comparison to other languages (such as C) which only accept primitive types.

1.2.5 Example

```
1       def version = "1.2"
2       switch(GIT_BRANCH) {
3           case "develop": \\case is of type String
4               result = "dev"
5               break
```

[3]According to IBM documentation: "Java™ database connectivity (JDBC) is the JavaSoft specification of a standard application programming interface (API) that allows Java programs to access database management systems."[17]

```
        case ["master", "support/${version}".toString()]: \\case is of type
    Arraylist
            result = "list"
            break
        case "support/${version}": \\case is of type GString
            result = "sup"
            break
        default:
            result = "def"
            break
    }
    print("${result}")
```

2 Drawbacks of Groovy

Even though this report greatly praises Groovy, it is by no means a perfect language. Optional typing (i.e. *def* keyword), even though sometimes can be a handy feature especially when scripting, it can make the program unnecessary difficult to understand. Strong typing makes the contract more concrete, especially in public APIs, avoids possible passed arguments type mistakes and also helps the IDE with code completion.

Additionally, progress in the Java language tends to obliterate the strong points made by Groovy. *Var* keyword which was introduced in Java 10 [19] started to look like the *def* keyword (optional typing) of Groovy and pattern matching for the switch keyword introduced in Java 17 [20], although still more cumbersome than Groovy, may under certain circumstances reduce the need for a new language.

3 Runtime & compile-time metaprogramming An in depth analysis

Groovy like various others programming languages can support metaprogramming. Groovy has two modes of metaprogramming; runtime and compile-time metaprogramming.

3.1 Runtime metaprogramming

With runtime metaprogramming it is possible to intercept, inject or even interact with other methods at runtime. This concept (also known as *monkey patch*) modifies run time code but does not alter the original source code. *Metaclasses* play a key role on this process as the compiler and the Groovy Runtime Engine interact with methods of a Metaclass class.

3.1.1 Expando Metaclass

Expando is a special case of Metaclass that allows for dynamically adding or modifying properties, constructors or even borrowing methods from other classes by using closure syntax. It is particularly useful for software testing and in particular simulating the behavior of a program (i.e. stubbing & mocking).

However, with great power comes great responsibility. Using the expando metaclass means that in the same JVM all objects of that class will have that method added to Metaclass which could maybe lead to problems. Thus, it is advised to use Categories, a feature borrowed from Objective-C which addresses the issue of additional methods by limiting them in a block of code (closure). An example of expando metaclass can be found in appendix section A.1.

3.2 Compile-time metaprogramming

In contrast to runtime metaprogramming, compile-time metaprogramming allows the code to be generated at compile time. Those transformations are altering the Abstract Syntax Tree (AST) of a program, which is why in Groovy are called AST transformations.

AST transformations are very important especially when dealing with a mix of Groovy and Java code. For example an AST transformation can add methods to a class and be visible to Java; a thing which wouldn't be visible if runtime metaprogramming was used.

Groovy comes prepacked with various AST transformations to reduce boilerplate code (e.g.*@TupleConstructor*), improve coding efficiency (e.g.*@BaseScript*) and other. Every developer can also make their own project wide or method specific AST transformation.

AST transformations are of two categories; global and local. Global are applied globally on the code wherever the transformation can apply(field, method, class etc.). Local transformations are applied locally by using the annotation notation. The compiler will discover them and apply the transformation on these code elements. Every developer can write their own AST (global or local) transformations. For the simplicity of this report in section A.2 a simple prepackaged example of @Lazy annotation can be found which delays field instantiation until the time when that field is first used.

4 Groovy applications

Having set some basic Groovy rules and syntax it's time to see where this language is being used. The following section covers a large part of a system's development lifecycle (Development, Testing [see section 4.1] & Integration[see sections 4.2 - 4.3]) in which Groovy is the primary language. Section A contains three example applications; more specifically section A.3 contains a simple Groovy program, which in turn is then built into bytecode by a Gradle program (which uses groovy scripting) (section A.4) and then deployed in a (hypothetical) pipeline in section A.5 which uses Groovy as well.

4.1 Testing

Groovy has been widely used for writing test scripts in Spock and Geb. Groovy's easy to learn syntax and Spock's highly expressive language makes them a top choice in the testing industry for Software and Quality Assurance Engineers alike. As seen in section A.3 the absence of boilerplate code and its "narrative" nature are one of the reasons Groovy is preferred for this job.

4.2 Gradle

Gradle is an open-source build automation tool that is designed to be flexible enough to build almost any type of software. It is high performance, fully extensible and fully supported by numerous major IDEs such as IntelliJ IDEA, Android Studio and Eclipse. Gradle is heavily based upon Groovy in its core implementation and testing.[4] It can be also used to either script, build the application or just chain tasks with the keywords *shouldRunAfter* and *dependsOn*.

4.3 Jenkins

Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.

Groovy scripts are written in gdsl type files which are executed by the pipeline in a remote server such as an AWS environment. Jenkins makes also heavy use of Gradle (see section 4.2) to combine the whole pipeline into one seamless process.

5 Conclusion

Groovy was created as a Java supplementary and as time progressed it is considered a great Java alternative. It vastly improves various features found in many object oriented languages while it adds many more due to its vibrant community of developers. Meanwhile, its applications in various parts of the product's development cycle make it the ideal choice for many.

A Coding Examples

A.1 Runtime metaprogramming Example

The following example is taken from "Groovy in Action" Konig et al [21].

```groovy
class Spy {
    static {
        def mc = new ExpandoMetaClass(Spy, false, true)
        mc.initialize()
        Spy.metaClass = mc
    }
    String name = "James"

    void methodMissing(String name, args) {
        if (name.startsWith('changeNameTo')) {
            println "Adding method $name"
            String newName = name.substring(12)
            def newMethod = { delegate.name = newName }
            Spy.metaClass."$name" = newMethod
            newMethod()
        } else {
            throw new MissingMethodException(name, this.class, args)
        }
    }
    static void main(String[] args) {
        def spy = new Spy()
        assert "James" == spy.name
        spy.changeNameToAustin()
        assert "Austin" == spy.name
        spy.changeNameToMaxwell()
        assert "Maxwell" == spy.name
```

[4]More information about the source code can be found at Gradle's repository hosted at Github gradle/gradle

```
        spy.changeNameToAustin()
    }
}
```

The output of this program if we use main as an entry point is:

```
Adding method changeNameToAustin
Adding method changeNameToMaxwell
```

Notice that even though there are two calls *spy.changeNameToAustin()* there is only one print to the console as the second one was already registered as a method and therefore *methodMissing* was not used.

A.2 Compile time metaprogramming Example

The following example is taken from "Groovy in Action" Konig et al [21] as well. It shows how an expensive resource (e.g. database) can be lazily evaluated (i.e.@Lazy) to save costs.

```
class Resource {
 private static alive = 0
 private static used = 0
 Resource() { alive++ }
 def use() { used++ }
 static stats() { "$alive alive, $used used" }
}
class ResourceMain {
 def res1 = new Resource()
 @Lazy res2 = new Resource()
 @Lazy static res3 = { new Resource() }()
 @Lazy(soft=true) volatile Resource res4
}
new ResourceMain().with {
 assert Resource.stats() == '1 alive, 0 used'
 res2.use()
 res3.use()
 res4.use()
 assert Resource.stats() == '4 alive, 3 used'
 assert res4 instanceof Resource
 }
```

A.3 Groovy test file

```
import spock.lang.Specification
class UserJourneySpec extends Specification {
    ArrayList shoppingCart = []

    def setup() { \\setup runs automatically before every test
        println("Setting up test data...")
        assert shoppingCart.empty \\the assert keyword could be omitted
    }

    def cleanup() { \\cleanup runs automatically after every test
        print("Deleting everything from the basket")
        shoppingCart.clear()
    }

    def "Users buys groceries from the website"() {
        when: "User logs in the website"
        userLogsIn() \\ an external method that logs in the user into the website

        and: "Adds some products to the basket"
        shoppingCart += "banana"
        shoppingCart += "apple"
        shoppingCart += "orange"

        then: "Cart is not empty"
        shoppingCart.size() == 3

        when: "User decides to remove the apple from his cart"
        shoppingCart -= "apple"

        then: "Cart does not contain apple but still has the other fruits"
        !shoppingCart.contains("apple")
        shoppingCart.size() == 2
    }
}
```

A.4 Gradle file

```
[...] \\ gradle specific commands are omitted for brevity

tasks.register('testSuiteSizeValidation') {
    doLast {
        print("Validating that no extra test classes were added by a developer")
        def path = "..\\GroovyTesting\\src\\test\\groovy\\"
        def list = []
        def dir = new File(new File(path).getAbsolutePath())
        dir.eachFileRecurse { file ->
            list << file
        }
        assert list.size() == 1
    }
}
```

A.5 Jenkinsfile

```
pipeline {

    agent any

    triggers {
        cron('H */8 * * *') //regular builds
        pollSCM('* * * * *') //polling for changes, here once a minute
    }

    stages {
        stage('Checkout') {
            steps { //Checking out the repo
                checkout changelog: true, poll: true, scm: [$class: 'GitSCM',
    branches: [[name: '*/master']], browser: [$class: 'BitbucketWeb', repoUrl: '
    https://web.com/blah'], doGenerateSubmoduleConfigurations: false, extensions:
```

```
    [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'git', url: 'ssh://
    git@git.giturl.com/test/test.git']]]
            }
        }
        stage("Validating that the number of tests is correct") {
            steps {
                script {
                    try {
                        print("Gradle task starting now...")
                        sh "./gradlew -q testSuiteSizeValidation"
                    } finally {
                        print("Task finished!")
                    }
                }
            }
        }

        stage("Building the application") {
            steps {
                script {
                    try {
                        print("Gradle task starting now...")
                        sh './gradlew -q build' //run a gradle task
                    } finally {
                        print("Task finished!")
                    }
                }
            }
        }
    }
    stage("Notify in Slack") {
        buildStatus = result
        def subject = "${buildStatus}: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'"
        def summary = "${subject} (${env.BUILD_URL})"
      if (result == "FAIL") {
        color = "RED"
        colorCode = "#FF0000"
        slackSend (color: colorCode, message: summary)
        error("Job has failed")
      }else{
        color = "GREEN"
        colorCode = "#00FF00"
        slackSend (color: colorCode, message: summary)
      }
  }
}
```

References

[1] *The Apache Groovy programming language*. URL: https://groovy-lang.org/.

[2] *TIOBE index*. URL: https://www.tiobe.com/tiobe-index/.

[3] James Strachan. *Groovy - the birth of a new dynamic language for the Java platform*. 2003. URL: https://web.archive.org/web/20030901064404/http://radio.weblogs.com/0112098/2003/08/29.htm.

[4] *JSR 241: The Groovy Programming Language.* URL: https://jcp.org/en/jsr/detail?id=241.

[5] Paul King. "A History of the Groovy Programming Language". In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386326. URL: https://doi.org/10.1145/3386326.

[6] Guillaume Laforge. *Groovy 1.0 is there!* URL: https://markmail.org/message/qspd5ufq35v7yk3a.

[7] Jeremy Rayner. *Groovy 1.0 released!* URL: http://javanicus.com/blog2/items/204-index.html.

[8] Oliver Plohmann. *Groovy 2.0 Performance compared to Java.* URL: http://objectscape.blogspot.com/2012/08/groovy-20-performance-compared-to-java.html.

[9] Russel Winder. *static compilation for Groovy.* URL: https://markmail.org/message/g627wfm5au3emher.

[10] Endre Stølsvik. *Consider statically typed/compiled as default for Groovy 3.0.* URL: https://issues.apache.org/jira/browse/GROOVY-8329.

[11] Paul Krill. *JetBrains readies JVM-based language.* URL: https://web.archive.org/web/20190907161741/https://www.infoworld.com/article/2622405/jetbrains-readies-jvm-based-language.html.

[12] *JSR 336: JavaTM SE 7 Release Contents.* URL: https://jcp.org/en/jsr/detail?id=336.

[13] Jeremy Siek and Walid Taha. "Gradual Typing for Objects". In: Aug. 2007, pp. 2–27. ISBN: 978-3-540-73588-5. DOI: 10.1007/978-3-540-73589-2_2.

[14] *The Java™ Tutorials Lesson: Annotations.* URL: https://docs.oracle.com/javase/tutorial/java/annotations/.

[15] *GEP-8.* URL: https://groovy.apache.org/wiki/GEP-8.html.

[16] *GEP 10.* URL: https://groovy.apache.org/wiki/GEP-10.html.

[17] *What is JDBC?* URL: https://www.ibm.com/docs/en/informix-servers/12.10?topic=started-what-is-jdbc.

[18] James Strachan. *Class GString.* URL: https://docs.groovy-lang.org/latest/html/api/groovy/lang/GString.html.

[19] *JDK 10 Release Notes.* URL: https://www.oracle.com/java/technologies/javase/10-relnote-issues.html.

[20] *JDK 17 Release Notes.* URL: https://openjdk.java.net/projects/jdk/17/.

[21] Dierk Konig et al. *Groovy in Action.* 2nd ed. Manning, 2015. ISBN: 9781935182443; 1935182447.