

A History of the Groovy Programming Language

PAUL KING, Object Computing, Australia

Shepherd: Ralph Johnson, Metaefficient, USA

This paper describes the history of the Groovy programming language. At the time of Groovy's inception, Java was a dominant programming language with a wealth of useful libraries. Despite this, it was perceived by some to be evolving slowing and to have shortcomings for scripting, rapid prototyping and when trying to write minimalistic code. Other languages seemed to be innovating faster than Java and, while overcoming some of Java's shortcomings, used syntax that was less familiar to Java developers. Integration with Java libraries was also non-optimal. Groovy was created as a complementary language to Java—its dynamic counterpart. It would look and feel like Java but focus on extensibility and rapid innovation. Groovy would borrow ideas from dynamic languages like Ruby, Python and Smalltalk where needed to provide compelling JVM solutions for some of Java's shortcomings.

Groovy supported innovation through its runtime and compile-time metaprogramming capabilities. It supported simple operator overloading, had a flexible grammar and was extensible. These characteristics made it suitable for growing the language to have new commands (verbs) and properties (nouns) specific to a particular domain, a so called Domain Specific Language (DSL). While still intrinsically linked with Java, over time Groovy has evolved from a niche dynamic scripting language into a compelling mainstream language.

After many years as a principally dynamically-typed language, a static nature was added to Groovy. Code could be statically type checked or when dynamic features weren't needed, they could be turned off entirely for Java-like performance. A number of nuances to the static nature came about to support the style of coding used by Groovy developers.

Many choices made by Groovy in its design, later appeared in other languages (Swift, C#, Kotlin, Ceylon, PHP, Ruby, Coffeescript, Scala, Frege, TypeScript and Java itself). This includes Groovy's dangling closure, Groovy builders, null-safe navigation, the Elvis operator, ranges, the spaceship operator, and flow typing. For most languages, we don't know to what extent Groovy played a part in their choices. We do know that Kotlin took inspiration from Groovy's dangling closures, builder concept, default *it* parameter for closures, templates and interpolated strings, null-safe navigation and the Elvis operator.

The leadership, governance and sponsorship arrangements of Groovy have evolved over time, but Groovy has always been a successful highly collaborative open source project driven more by the needs of the community than by a vision of a particular company or person.

CCS Concepts: • **Software and its engineering** → **Object oriented languages**; Multiparadigm languages; Abstract data types; Polymorphism; Inheritance; Data types and structures; Classes and objects; Recursion.

Additional Key Words and Phrases: Dynamic typing, Static typing, Object-oriented, Functional programming, Closure, Scripting, Domain Specific Languages, Metaprogramming, Extensibility

ACM Reference Format:

Paul King. 2020. A History of the Groovy Programming Language. *Proc. ACM Program. Lang.* 4, HOPL, Article 76 (June 2020), 53 pages. <https://doi.org/10.1145/3386326>

Author's address: Paul King, Object Computing, Brisbane, Australia.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).

2475-1421/2020/6-ART76

<https://doi.org/10.1145/3386326>

CONTENTS

Abstract	1
Contents	2
1 The Language Vision	4
1.1 Better for Script Writers	4
1.2 Better for Data Scientists	6
1.3 Better for Java Developers	8
1.4 Has the Vision Changed?	9
2 The Journey to 1.0	9
2.1 The JSR Activity	10
2.2 Inspiration from Other Languages	11
2.3 Similarities with Java	11
2.4 Differences to Java	11
3 Interesting Aspects of the Language	12
3.1 Operator Overloading	12
3.2 Command Chains	13
3.3 Builders and GPath	15
3.4 Script Integration	16
4 Language Governance, Hosting, and Community	17
4.1 Hosting	17
4.2 Governance	17
4.3 Sponsorship and Contributors	18
5 Abridged History of Language and Library Changes since 1.0	19
5.1 Groovy 1.5	19
5.2 Groovy 1.6	21
5.3 Groovy 1.7	22
5.4 Groovy 1.8	23
5.5 Groovy 2.0	24
5.6 Groovy 2.1	24
5.7 Groovy 2.2	24
5.8 Groovy 2.3	24
5.9 Groovy 2.4	25
5.10 Groovy 2.5	25
5.11 Groovy 3.0	25
5.12 Groovy 4.0	26
5.13 Lessons Learned in Evolving the Language	26
6 The Evolution of the GDK	27
7 The Evolution of AST Transforms	28
7.1 Moving to Meta-annotations	30
7.2 Macros and Macro Methods	30
7.3 Feature Interaction	31
8 The Evolution of Java Compatibility	31
8.1 Supporting Generics	33
8.2 Implicit SAM Coercion	33
8.3 Lessons Learned Maintaining Java Compatibility	33
9 A Static Nature for a Dynamic Language	34
9.1 Exploring Options	35

9.2	Supporting Existing Coding Idioms	36
9.3	Type Checking Extensions	37
9.4	Type Checking Groovy’s Closure	39
10	The Importance of Testing	40
10.1	Testing as Part of Community Building	41
10.2	Testing Lessons Learned	41
11	Health Status	41
12	On-going Challenges and Missed Opportunities	43
12.1	On-again Off-again Focus on Performance	43
12.2	Traits and AST Transforms	43
12.3	Joint Compiler	44
12.4	Android Development—a Missed Opportunity	44
12.5	Learning to Say “No”	44
12.6	Coping with Big Tasks in Open Source	45
13	Influencing Other Languages	45
14	What’s Next for Groovy?	46
	Acknowledgments	47
A	Non-overridable Operators	47
B	Supplementary Listings	47
C	Release Information	49
	References	49
	Non-archival References	53

1 THE LANGUAGE VISION

In 2003, Java was stable and a broad range of quality libraries existed within the Java ecosystem. Yet, Java wasn't the language of choice in numerous scenarios:

- Shell scripting or using the Perl language was deemed more appropriate for operating system level activities, text processing tasks or file munging.
- Data scientists preferred the simplicity of Python or R over Java's static typing.
- Smalltalk programmers forced to use the more dominant Java language lamented the perceived loss of power and fun in their programming activities.
- Developers of web frameworks like Ruby on Rails or Emacs tinkerers felt they could achieve more innovation using languages where extensibility was in the hands of language users.

Groovy was created as a complementary language to Java to provide a compelling JVM solution for a broad range of scenarios including those outlined above. The initial concept was to produce a new language for the JVM that was "*just like Java only better*". It might simplistically and superficially be considered a "*Java-flavored Python*". The language should:

- have a Java "*look and feel*"—making it easy for Java developers to learn
- reduce complexity and other pain points experienced by Java developers—borrowing ideas from other languages if needed but adapting those ideas to ensure consistency among Groovy's features
- integrate well with Java—allowing traditional library development to continue in Java with Groovy acting as the glue to make using those libraries easy
- be extensible—allowing users to grow their own glue languages, or Domain Specific Languages (DSLs), suitable for their particular domains
- elevate code to be a first class construct via closures—supporting various functional programming idioms and styles

By 2009, Groovy was becoming more popular and was being used in a much broader range of contexts than perhaps originally envisaged. Groovy was being used for whole codebases not just the occasional script. While Groovy's runtime metaprogramming capabilities were powerful, adding a compile-time metaprogramming capability allowed alternative ways to reduce complexity without incurring runtime overheads. Many Java design patterns and idioms were complex or involved writing lots of boiler plate code. Compile-time metaprogramming provided a declarative approach to simplify applying such design patterns and idioms. The language vision was largely unchanged but Groovy had a new weapon to fight complexity.

By 2012, Groovy continued to grow in popularity and was now being frequently used in contexts which might more typically be regarded as suited to Java development. Users were searching for additional compile-time type checking and additional performance. Groovy added a static nature. Groovy wasn't diverting the language from dynamic to static and would continue to allow polyglot development mixing Java and Groovy, but would now also support static typing and related performance enhancements in contexts where it would be beneficial. In being "*just like Java only better*", it could also be more like Java when needed.

The fuzzy concept of "*just like Java only better*" is perhaps clearer if we give a few examples.

1.1 Better for Script Writers

As a scripting language, Groovy should be good for writing scripts. While Java's libraries were feature rich, system administrators and other script writers didn't find using Java or those libraries particularly attractive. They were used to the terseness of Perl, Awk, Sed, Grep and Bash scripts, and perceived static typing as of little value for these kind of scripts. Groovy's goal: re-purpose those Java libraries to meet script writers needs.

Let's consider a regular expression example. Suppose we wish to extract whitespace separated key/value pairs from the string "a:1 b:2 c:3" and, even though we have simplified this particular example, let's assume we don't know in general the size of the keys or values or whitespace in between. This seems like an ideal task for regular expressions. This is not the place for a regex tutorial but the example uses a regular expression with groups that allow us to find 3 matches in our sample string. For illustrative purposes, we'll check the details of the second match. Java's libraries can easily handle this as shown in Listing 1.

```

1 import org.junit.Test;
2 import java.util.regex.Matcher;
3 import java.util.regex.Pattern;
4 import static junit.framework.Assert.assertEquals;
5
6 public class RegexTest {
7     @Test
8     public void testRegexMatching() {
9         Pattern regex = Pattern.compile("(\\S+):(\\S+)");
10        Matcher matcher = regex.matcher("a:1 b:2 c:3");
11        int match = 0;
12        while (matcher.find()) {
13            match++;
14            if (match == 2) {
15                String full = matcher.group(0);
16                assertEquals("b:2", full);
17                String key = matcher.group(1);
18                assertEquals("b", key);
19                String value = matcher.group(2);
20                assertEquals("2", value);
21            }
22        }
23        assertEquals(3, match);
24    }
25 }

```

Listing 1. Java regular expression matching

The details aren't important but basically in the code we create a matcher from our regex pattern and call `find()` for each match. For the second match, we'll extract group 0 (the total match), group 1 (the key) and group 2 (the value). The code is not complex but doesn't match the terseness of Grep or Perl.

Groovy offers a more terse way of calling that same library using the regex match operator (`=~`). It was loosely inspired by Perl's similar operator but it is backed by a normal Java regex `Matcher` and yields code as shown in Listing 2.

```

1 def (first, second, third) = 'a:1 b:2 c:3' =~ /(\\S+):(\\S+)/
2 def (full, key, value) = second
3 assert full == 'b:2' && key == 'b' && value == '2'

```

Listing 2. Groovy regular expression matching

We gather our first, second and third matches, and then examine the details for the second match. It's the same library calls under the covers that Java developers would use but the code is much more succinct and closer to what the script writers will be familiar with. Groovy's regex match operator is crucial to this example but other language features all work together in achieving

the terseness including optionally typed variables, multi-assignment, the slashy string to avoid escaping the backslashes in the regex and the greater emphasis on assertions.

Groovy offers similar levels of simplification for working with files, streams, collections, processes, dates, XML, SQL, JSON, command line processing, templating, the Swing graphics classes, and many other areas. And, while presented as a feature for script writers, these improvements are more broadly applicable since Java developers use many of these technologies in the business logic or integration logic of their enterprise applications.

1.2 Better for Data Scientists

Let's consider the scenario of offering data scientists an environment for doing matrix calculations on the JVM. A useful Java library in this scenario is the Apache Commons Math library [CommonsMath 2020a]. Let's use the example from that library's documentation [CommonsMath 2020b] but extend it slightly. The example is written in Java as shown in Listing 3.

```

1 import org.apache.commons.math3.linear.*;
2
3 public class MatrixMain {
4     public static void main(String[] args) {
5         double[][] matrixData = { {1d,2d,3d}, {2d,5d,3d}};
6         RealMatrix m = MatrixUtils.createRealMatrix(matrixData);
7         double[][] matrixData2 = { {1d,2d}, {2d,5d}, {1d, 7d}};
8         RealMatrix n = new Array2DRowRealMatrix(matrixData2);
9         RealMatrix o = m.multiply(n);
10        // Invert p, using LU decomposition
11        RealMatrix oInverse = new LUDecomposition(o).getSolver().getInverse();
12        RealMatrix p = oInverse.scalarAdd(1d).scalarMultiply(2d);
13        RealMatrix q = o.add(p.power(2));
14        System.out.println(q);
15    }
16 }
```

Listing 3. Java use of RealMatrix

Which has this output:

```
Array2DRowRealMatrix[{15.1379501385,40.488531856},{21.4354570637,59.5951246537}]
```

What are some of the pain points for the data scientists?

- The code is a little verbose
- The output isn't particularly friendly
- They need to add the library to Java's classpath

One option would be to contemplate writing a matrix library from scratch in Groovy but there is no need. Groovy offers many capabilities to enhance existing libraries providing a revamped friendlier experience. This is sometimes referred to as the *pimp my library* [Odersky 2006] idiom.

Possible improvements using Groovy might include:

- Remove the need to set up the classpath or use a build tool by using Groovy's @Grab annotation
- Use operator overloading and metaprogramming capabilities to make the code more concise
- Optionally provide a custom OutputTransforms script for the GroovyConsole

The OutputTransforms script is an extensibility feature of the GroovyConsole to aid custom visualization of script results. It overrides how specific classes will be rendered using Java's Swing classes. The details are not important (see Listing 28 if interested) but we should note that it might be as simple as a few lines of Groovy code to define.

This results in an improved experience for our data scientists as illustrated in the GroovyConsole (Figure 1).

```

1 @Grab('org.apache.commons:commons-math3:3.6.1')
2 import org.apache.commons.math3.linear.*
3
4 RealMatrix.metaClass {
5     plus << { RealMatrix ma -> delegate.add(ma) }
6     plus << { double d -> delegate.scalarAdd(d) }
7     multiply { double d -> delegate.scalarMultiply(d) }
8     bitwiseNegate { -> new LUdecomposition(delegate).solver.inverse }
9     constructor = { List l -> MatrixUtils.createRealMatrix(l as double[][]) }
10 }
11
12 def m = [[1d,2d,3d], [2d,5d,3d]] as RealMatrix
13 def n = [[1d,2d], [2d,5d], [1d,7d]] as RealMatrix
14 def o = m * n
15 def p = (~o + 1) * 2
16 o + p ** 2
  
```

Result:

```

[ 15.137950138504156 40.48853185595568 |
 21.43545706371191 59.59512465373961 ]
  
```

Execution complete. 16:11

Fig. 1. Matrix programming environment: Take 1

There is now no need to set up the classpath and the actual matrix calculations in the bottom half of Figure 1 are more succinct. But we can make further improvements if we wish:

- Groovy allows a base script to be defined for scripts
- Compiler customization options allow specifying source file extensions, base classes, and automatic imports so the code for activating our enhancements can be hidden away from the data scientists

The details aren't critical, but it's easy enough to define a base script class as shown in Listing 4 which will allow these definitions to be reused across many scripts.

```

1 import org.apache.commons.math3.linear.*
2
3 abstract class MatrixBase extends Script {
4     MatrixBase() {
5         super()
6         RealMatrix.metaClass {
7             plus << { RealMatrix ma -> delegate.add(ma) }
8             plus << { double d -> delegate.scalarAdd(d) }
9             multiply { double d -> delegate.scalarMultiply(d) }
10            bitwiseNegate { -> new LUdecomposition(delegate).solver.inverse }
11            constructor = { List l -> MatrixUtils.createRealMatrix(l as double[][]) }
12        }
13    }
14 }
  
```

Listing 4. Defining a BaseScript class

We could stop here but Groovy provides many options for further evolving our DSL environment. We'll add some compiler customization to hide away the base script and have a friendly alias for

the matrix class (see Listing 30 for details). Finally, at the cost of having an additional library on the classpath, and a slightly more sophisticated OutputTransforms script (see Listing 29 for details), we can have prettier output in the Groovy Console as can be seen in Figure 2.

The screenshot shows a window titled "MatrixSimple.mgroovy - GroovyConsole". The code in the editor is as follows:

```

1 def m = [[1d,2d,3d], [2d,5d,3d]] as Matrix
2 def n = [[1d,2d], [2d,5d], [1d,7d]] as Matrix
3 def o = m * n
4 def p = (~o + 1) * 2
5 o + p ** 2

```

The output, labeled "Result:", is a 2x2 matrix of floating-point numbers, displayed in a formatted style:

$$\begin{pmatrix} 15.137950138504156 & 40.48853185595568 \\ 21.43545706371191 & 59.59512465373961 \end{pmatrix}$$

At the bottom of the console, it says "Execution complete." and "1:1".

Fig. 2. Matrix programming environment: Take 2

Our data scientists have a more productive programming environment but are still making the same calls to our selected matrix library as the original Java code. In this environment, they can go far with minimal Java or Groovy knowledge. We can give them a quick cheat-sheet showing them how to define and use matrices with the same terseness they might expect in R or Matlab. If however, the data scientists are in fact familiar with Java they can, for the most part, copy and paste the original Java example from the Matrix library's website—avoiding Groovy shortcuts to start with and gradually converting over to use additional Groovy idioms at their own convenience if they choose. Alternatively, if they are familiar with Groovy, they can use some of Groovy's more advanced capabilities like memoization and potentially end up with a script that executes faster than the Java example.

This motivating example might seem a little strange. The example targets data scientists as an audience and shows that Groovy is well suited to writing DSLs. But what about serious programmers tackling some general-purpose programming problems? Arguably, every program is trying to solve a problem in a particular domain; i.e. every program is a DSL. We aren't advocating writing such output and compiler customization scripts for every program. But perhaps thinking carefully about our nouns and verbs, and pondering whether to grow our language just a little to meet the problem at hand, might perhaps be a useful technique in striving for clear maintainable code.

1.3 Better for Java Developers

For a traditional Java developer, perhaps improved regex functionality or DSL looking code might not rank highly on their list of desirable language features. The bulk of their coding might involve using Java design patterns and idioms, and following guidelines such as those espoused in Effective Java [Bloch 2017]. Let's consider a more typical task for such a developer—the task of writing an immutable Book class. The class might have some properties such as a title (a String), a list of authors (a List) and the date published (a Date). We might also want a copy constructor for making a new book instance based on an existing one, and the ability to externalize book instances, and finally we might want to sort lists of such books by title or publication date. Following the practices in Effective Java [Bloch 2017, Item 17] and other common Java idioms, we would write:

- private backing fields for the properties
- getter methods for accessing the properties which would do defensive copying to maintain immutable integrity
- a tuple constructor and copy constructor doing defensive copying as needed
- optionally a toString method
- an appropriate equals and hashCode method
- a compareTo method
- a ComparatorByTitle and ComparatorByPublicationDate class
- readExternal and writeExternal methods

Rather than writing all of this code by hand, or even letting an IDE generate some for me, Groovy allows a declarative alternative as shown in Listing 5.

```

1 @Immutable(copyWith = true)
2 @Sortable(excludes = 'authors')
3 @AutoExternalize
4 class Book {
5     String title
6     List<String> authors
7     Date publicationDate
8 }
```

Listing 5. The declarative way to define an immutable class in Groovy

While the Groovy code is orders of magnitude shorter to write with over 500 lines of injected equivalent Java code, the key improvement is that the code has a clearer intent making it easier to read and understand. Also, we can be confident that the injected code will be based on current best practices for each aspect and the maintenance burden is reduced since we can let the Groovy library maintainers worry about evolving those implementations over time if needed.

1.4 Has the Vision Changed?

Groovy's compile-time metaprogramming and static nature were not features being considered in early versions of Groovy. You might think the language's direction has changed dramatically. In reality, that's not the case. It was never a case of Groovy plugging a few holes in Java and then stopping innovation. The goal has always been to address pain points experienced by Java developers and as those pain points shift, it is natural for Groovy to adapt accordingly.

An early guiding principle in Groovy's design was to remain close to Java. You might ask whether, after many versions, is it time to diverge in more extreme ways from Java? This has certainly been a question continually revisited by Groovy's team, but to date, the answer has been no. Groovy remains valuable to its users by being responsive to their pain points. As research, technology and industry practices evolve, Java adapts to remain a compelling option. By remaining close to Java, Groovy can piggy-back on those improvements and proactively focus on pain points which have not yet been a focus for Java.

2 THE JOURNEY TO 1.0

Groovy was the brainchild of James Strachan. In August 2003, he announced an effort to work on a new language for the JVM [Strachan 2003b]:

So I've been musing a little while if it's time the Java platform had its own dynamic language designed from the ground up to work real nice with existing code; creating/extending objects normal Java can use and vice versa. Python/Jython's a pretty good base—add the nice stuff from Ruby and maybe sprinkle on some AOP features

and we could have a really Groovy new language for scripting Java objects, writing test cases and who knows, even doing real development in it.

We're starting simple with the nice tuples, sequences, maps from python & closures from ruby and being concise & dynamically typed with a java-look-and-feel though where we end up is anyone's guess right now.

Strachan started the early work but soon a community was rallying behind the language. Betas started being produced with a plethora of new features inspired by other languages which seemed to have nice solutions to some of the problems Groovy was trying to address. Initially languages like Ruby, Python, [Nice \[2020\]](#) and [Beanshell \[2020\]](#) were sources of inspiration [[Strachan 2003a](#)]. Specific features of Smalltalk and to a lesser extent Eiffel, Dylan and Objective-C also influenced early decisions. These formative stages included lots of exploration and prototyping, and continued until early 2004. Strachan with input from the community set direction while Bob McWhirter carried out early parser work [[McWhirter 2003](#)].

At that time, the opportunity arose to potentially have Groovy recognized as an official JDK language and perhaps even be incorporated into the JDK at some point in the future [[König et al. 2015](#)]:

[... Soon after the project started ...] Richard Monson-Haefel met James, who introduced him to Groovy. Richard immediately recognized Groovy's potential and suggested the submission of a Java Specification Request (JSR-241). To make a long story short, this JSR passed "without a hitch," as Richard puts it in his blog, thanks to additional support from Geir Magnusson, Jr. and the foresight of the folks at Sun Microsystems. They don't see Groovy as Java's rival but rather as a companion that attracts brilliant developers who might otherwise move to Ruby or Python and thus away from the Java platform.

2.1 The JSR Activity

In March 2004, Groovy was formally submitted as a Java Specification Request (JSR) [JSR 241 \[2004\]](#) and accepted by ballot. But initial acceptance is just the start of the battle. For the JSR to reach final approved status would require not only a reference implementation (which they had) but a specification and compatibility test kit.

The JSR process was an acid test for the Groovy community. As well as being a lot more work than some were prepared for, it showed where contributors were pushing in different directions and it imposed more structure on the development than some (including the founder) were willing to accept. Strachan stepped aside to let others take on the task. Guillaume Laforge stepped in to lead the project.

The JSR deliverables never reached the point where they could be voted upon but the process itself proved valuable. Without it, the myriad of features emerging from Groovy's early development while useful and interesting may not have worked seamlessly together. And, while the Groovy Language Specification (GLS) never reached 100% completeness, what was done now forms part of the Groovy documentation. Similarly, a complete compatibility test suite wasn't completed to JSR standards, but what was done was added to Groovy's large test suite.

The culmination of the beta prototypes and JSR activities was version 1.0 which was released in January 2007 [[Laforge 2007a](#); [Rayner 2007](#)]. Version 1.0 could perhaps best be described as "most of Java" plus:

- Closures, scripts, builders, GStrings (interpolated Strings), named parameters
- Properties (similar to JavaBean properties but with less boilerplate and supporting the Uniform Access Principal [[Meyer 1997](#)])
- Native regular expression support, operator overloading

- Native literal syntax for collective types (like lists, maps, ranges)
- GPath expressions (simplified access to aggregate data types)
- Runtime metaprogramming, optional typing, ranges

As well Groovy's own library classes, about 430 enhancements to Java classes had been added in what became known as the Groovy Development Kit (GDK).

For those involved in the language, the release of 1.0 had become more important than JSR acceptance. At this point though, there was still some hope that time could be found later to complete the JSR activities. It wasn't until April 2012 after eight years of inactivity, and not long before the Groovy 2.0 release, that the Spec Lead changed the status of JSR 241 to dormant.

2.2 Inspiration from Other Languages

Groovy's early string support and literal list and map notations borrowed heavily from Python. Named parameters and collection method names (`collect`, `inject`, `findFirst`, `reverse`) came from Smalltalk. Ruby influenced numerous aspects of the metaprogramming mechanism and both Ruby and Smalltalk influenced to a lesser extent the closure syntax.

2.3 Similarities with Java

Groovy, like Java has many syntax characteristics by virtue of being a member of the C family of languages. It uses curly braces to delimit blocks rather than whitespace indentation. It follows Java's syntax for class, method, field and local variable definitions, and offers the familiar control structures, e.g. `if/else`, `switch/case`, `for/while` loops.

The similarities with Java don't end there. Groovy uses the same syntax and keywords as Java in many scenarios. In fact, nearly all of Java syntax is valid Groovy syntax. Groovy shares a nearly identical conceptual model for a developer. It uses the same packaging system, has similar definitions for classes and methods, and borrows heavily from Java's exception handling approach—though Groovy coalesces all exceptions to be unchecked. It uses the same object, security and threading models, and, for the most part, the same data types.

2.4 Differences to Java

So, what is different and why?

- A *dynamic execution model* means decisions made at compile time in other languages are sometimes delayed until runtime. This allows, for instance, method dispatch based on runtime type, and interacting with an object's lifecycle, such as intercepting calls. Code fragments can also be evaluated at runtime.
- A *metaprogramming model* allows instances or classes to be enhanced with additional methods and properties.
- *Flexible syntax* allows various syntactic elements to be elided in contexts where there is no ambiguity including semicolons, certain imports, certain brackets, and the `return` keyword in some circumstances. Groovy also supports operator overloading, friendly literal list and map syntax, and special minimal notation for scripts. Optional typing means that language users can optionally elide type declarations in places where having an explicit type isn't important.
- *Interactive tooling* offer the developer interactive modes of interaction allowing them to experiment and prototype typically associated with interpreted languages. The `groovysh` tool provides a familiar Read-Eval-Print Loop (REPL) style of interaction within a terminal window¹. The `groovyConsole` offers a “mini-IDE” Swing application for typing and running programs.

¹JDK9 and above now also offers the JShell REPL tool

- *Extensibility* is supported via runtime and compile-time metaprogramming capabilities. Various tools including the type checker are extensible.

Combining these aspects makes Groovy well suited for writing embedded DSLs.

3 INTERESTING ASPECTS OF THE LANGUAGE

Since Groovy followed Java closely and took inspiration from other languages, you could argue it has few truly novel features. Nonetheless, the language certainly has some strengths and interesting features that users of the language find appealing.

3.1 Operator Overloading

Groovy doesn't support arbitrary custom operators—there was concern that the language would become too Perl or APL-like [Laforge 2007c]. Instead, it provides a fixed set of operators to be overridden for your types such as we did earlier in the matrix example. The fixed set of operators are the simple ASCII-based operators familiar to Java users for primitives and Strings plus a few additions.

Supporting a wider range of operator symbols using Unicode characters was rejected at the time to keep barriers to language usage low. Even a non-Unicode text editor will be good enough for someone wanting to write a few lines of Groovy for scripting. Groovy does support using Unicode characters in method, field and local variable names, and so far this has proved sufficient for Groovy's users.

For the common operators, the shorthand syntax is available for a particular object if that object has the operator method available either directly or via an extension method. The *Works with* column indicates which variants are provided by the GDK but you can write your own if not shown in that column.

Operator	Name	Operator method	Works with
a + b	Plus	a.plus(b)	Number, String, StringBuffer, Collection, Map, Date, ...
a - b	Minus	a.minus(b)	Number, String, List, Set, Date, Duration
a * b	Star	a.multiply(b)	Number, String, Collection
a / b	Divide	a.div(b)	Number
a % b	Modulo	a.mod(b)	Integral number
-a	Unary minus	a.unaryMinus()	Number, ArrayList
+a	Unary plus	a.unaryPlus()	"
a ** b	Power	a.power(b)	Number
a b	Numerical or	a.or(b)	Number, Boolean, BitSet, Process
a & b	Numerical and	a.and(b)	Number, Boolean, BitSet
a ^ b	Numerical xor	a.xor(b)	"
~a	Bitwise complement	a.bitwiseNegate()	Number
a[b]	Subscript	a.getAt(b)	Object, List, Map, CharSequence, Matcher, ...
a[b] = c	Subscript assignment	a.putAt(b, c)	Object, List, Map, StringBuffer, ...
a << b	Left shift	a.leftShift(b)	Integral number; "append" for StringBuffers, Writers, Files, Sockets, Lists
a >> b	Right shift	a.rightShift(b)	Number
a >>> b	Right shift unsigned	a.rightShiftUnsigned(b)	"
a as type	Enforced coercion	a.asType(typeClass)	Any type

Compound operators are also available (e.g. +=, *=, &=, etc.) but can't be separately overridden. If you override plus, that change will work for both + and +=.

Next are the equality and identity operators; only the `compareTo` method needs to be overridden for all the equality operators to be supported:

Operator	Name	Shorthand for	Works with
<code>a == b</code>	Equals	<pre>if (a instanceof Comparable) { a.compareTo(b) == 0 } else { a.equals(b) }</pre>	Object; consider <code>hashCode()</code>
<code>a != b</code>	Not equal	<code>!(a == b)</code>	Object
<code>a <=> b</code>	Spaceship	<code>a.compareTo(b)</code>	<code>java.lang.Comparable</code>
<code>a > b</code>	Greater than	<code>a.compareTo(b) > 0</code>	
<code>a >= b</code>	Greater than or equal to	<code>a.compareTo(b) >= 0</code>	
<code>a < b</code>	Less than	<code>a.compareTo(b) < 0</code>	
<code>a <= b</code>	Less than or equal to	<code>a.compareTo(b) <= 0</code>	
<code>a === b</code>	Reference equality	alias for <code>a.is(b)</code> ²	Object
<code>a !== b</code>	Reference inequality	<code>!(a === b)</code>	Object

For the pre/post inc/decrement operators, the next and previous methods are overwritten:

Operator	Name	Shorthand for	Works with
<code>a++</code>	Post-increment	<code>def b = a; a = a.next(); b</code>	Iterator, Number, String, Date, Range
<code>++a</code>	Pre-increment	<code>a = a.next(); a</code>	"
<code>a--</code>	Post-decrement	<code>def b = a; a = a.previous(); b</code>	"
<code>--a</code>	Pre-decrement	<code>a = a.previous(); a</code>	"

Groovy also has a number of non-overridable operators (see [Appendix A](#)).

3.2 Command Chains

Command chain syntax allows chained method calls to appear without brackets and dots. Most developers have no problems understanding the extra symbols but for some DSL writers, those symbols contribute a lot of unnecessary noise to the DSL.

Command chains improve Groovy’s suitability for writing particular kinds of natural language DSLs. The expression “`a b c d`” is parsed as “`a(b).c(d)`”.

The general form is:

```
method_1 args_1 [method_n args_n]*
```

which naturally lends itself to writing expressions containing an even number of terms. Special conventions apply when using comma separated or named arguments. The last method/argument pair in any chain can be replaced by a single property access term, allowing expressions with an odd number of terms to be handled. This feature allows the examples contained in [Listing 6](#).

```
1 turn(left).then(right)
2 move(forward).at(3.getKm()).div(h))
3 take(2.pills).of(chloroquine).after(6.hours)
4 paint(wall).with(red, green).and(yellow)
5 paint(wall).with(red, green, and).yellow
6 check(that: margarita).tastes(good)
7 given({}).when({}).then({})
8 select(all).unique().from(names)
9 take(3).cookies
```

Listing 6. Some expressions before command chains

²But while you can override `is`, you can’t override reference equality when used directly

To be re-written in a more natural language style as shown in Listing 7.

```

1 turn left then right
2 move forward at 3.km/h
3 take 2.pills of chloroquine after 6.hours
4 paint wall with red, green and yellow
5 paint wall with red, green, and yellow
6 check that: margarita tastes good
7 given { } when { } then { }
8 select all unique() from names
9 take 3 cookies

```

Listing 7. The expressions with command chains

Command chains are often used in combination with Groovy’s dynamic capabilities to create simple DSLs³. One approach might be to combine nested maps of closures in conjunction with command chains as shown in Listing 8.

```

1 show = { println it }
2 square_root = { Math.sqrt(it) }
3
4 def please(action) {
5     [the: { what ->
6         [of: { n -> action(what(n)) }]
7     }]
8 }
9
10 please show the square_root of 100
11 // ==> 10.0
12 // equivalent to: please(show).the(square_root).of(100)

```

Listing 8. Command chains for DSLs

An alternative might be to use the `ExpandoMetaClass`. The day after we released the version of Groovy supporting command chains, a member of the Japanese Groovy community blogged [[uehaj 2010](#)] about how to use the feature using non-English keywords as shown in Listing 9.

```

1 // Japanese DSL using GEP3 rules
2 Object.metaClass.を =
3     Object.metaClass.の = { clos -> clos(delegate) }
4
5 まず = { it }
6 表示する = { println it }
7 平方根 = { Math.sqrt(it) }
8
9 まず 100 の 平方根 を 表示する // First, show the square root of 100
10 // => 10.0

```

Listing 9. A Japanese DSL

Scala has also introduced similar DSL-friendly syntax to support natural language looking expressions of the form:

```
object method_1 arg_1 [method_n arg_n]*
```

Such expressions have an odd number of terms and different argument handling conventions

³Interested readers may wish to read further about “chloroquine” [[Antao 2008](#); [Antao et al. 2008](#)] and unit manipulation DSLs [[Laforge 2008](#)]

compared with Groovy. Each language implements its DSLs using slightly different techniques but the kind of natural language expressions Scala and Groovy can support are similar.

3.3 Builders and GPath

Constructing and processing hierarchical data structures is another area where Java's libraries are powerful but somewhat clunky to use compared to what is possible with dynamic languages. As an example, consider the code to generate some XML as shown in Listing 10.

```

1 import groovy.xml.MarkupBuilder
2
3 def writer = new StringWriter()
4 def builder = new MarkupBuilder(writer)
5 builder.root {
6     firstLevel(attr: 'foo') {
7         secondLevel(attr: 'bar')
8     }
9     firstLevel(attr: 'baz')
10 }
```

Listing 10. Generating XML

At first glance, it appears that the MarkupBuilder class must have methods named root, firstLevel, etc. But instead the attempts to call these methods are intercepted by the builder and used to build the appropriate XML nodes as per Listing 11.

```

1 <root>
2   <firstLevel attr='foo'>
3     <secondLevel attr='bar' />
4   </firstLevel>
5   <firstLevel attr='baz' />
6 </root>
```

Listing 11. The resulting XML

Similarly, reading such code is an area where dynamic styling can lead to succinct coding as shown in Listing 12.

```

1 def slurper = new XmlSlurper()
2 def root = slurper.parseText(writer.toString())
3 assert root.firstLevel.secondLevel.@attr == 'bar'
4 assert root.firstLevel[1].@attr == 'baz'
```

Listing 12. Parsing XML

Again, it appears that the slurper.parseText call returns an object with a method or property called firstLevel. In reality, method calls and property access are intercepted and used to look up the appropriate XML nodes.

The Groovy code for processing XML is more concise but have we given up type safety by using such dynamic code? At one level, we have. But in reality, the Java APIs for XML processing take Strings as the arguments for element and attribute names. So there isn't really any useful extra checking being carried out. What we'd really like is to check aspects like whether we have nested the elements in the correct order. We can't do this in general but it is possible in special cases such as if the type of XML is known, e.g. a known type like XHTML, or if the XML has a DTD or Schema which captures the valid elements and attributes. For XML, our APIs might have some

functionality to test some of these special cases but Groovy's extensible type checker provides a powerful additional approach for type checking such code.

Groovy offers similar libraries for dealing with other hierarchical data structures such as JSON, YAML, Swing GUI definitions, or a binary tree. In fact, this style is so common, Groovy provides numerous helper classes to simplify writing your own builders.

3.4 Script Integration

Groovy has been used for writing test scripts in Spock [Niederwieser et al. 2020] and SoapUI [SoapUI 2020], build scripts in Gradle [Gradle 2020] and Jenkins [Jenkins 2020b], and numerous declarative configuration scripts. It has also been used as a glue language in Grails [Jenkins 2020a] and Micronaut [Micronaut 2020], in numerous templating and content generation scenarios, and for capturing business rules. Such scenarios need special integration support.

Script Engines. Java developers will be familiar with compiling their source files to class files which can then be executed or packaged in a jar file for later use. Groovy can be used in this way too or some additional integration might be required if, for example, the source files exist on a web server or within a database and perhaps need to be compiled on the fly. To deal with various integration scenarios, Groovy offers numerous capabilities and classes, e.g. `Eval`, `GroovyClassLoader`, `GroovyShell` and `GroovyScriptEngine`. The classes offer different capabilities depending on whether caching of scripts is required, whether both scripts and classes must be supported, whether scripts or dependent scripts should be reloaded if their source files change on the file system, etc. As well as these native classes there is support for Java's in-build scripting [JSR 223 2006] and the Bean Scripting Framework engine [BSF 2020], both which support multiple languages, Spring Integration [Spring 2020], Greengine [Greengine 2020] and others.

Compiler Customization. We already saw some customization of the `groovyConsole` and compiler in the earlier matrix example. We saw customization using helper scripts but the compiler also has its own API which also supports the same customization options.

Customization is a powerful feature of the language both in terms of the kinds of extensibility it provides and how it makes those customizations available for tooling. We saw in the matrix example, it allowed us to add imports and static imports, define different behavior for different file extensions and define a base script class. You can also restrict the syntax of the language to a subset of Groovy (maybe forbid imports or `System.exit()` calls), and add AST transformations (e.g. automatically add static type checking).

Focus on Extensibility. The language designers of Groovy have always had a strong focus on extensibility. The motivating matrix example from Section 1.2 is a case in point. Not only could the language accommodate matrix operators but the console could be extended to better display matrices.

Initially this was seen as simply responding to an area of weakness in Java. It latter became a necessity to better manage evolution of the language and it's libraries. Groovy was becoming more popular, and there was an abundance of possible bells, whistles and potential extensions that could be added. Having standard extension mechanisms allowed the team to better structure the codebase, keeping special case logic to a minimum. More importantly, because we strived to make extension mechanisms available to users of the language, it allowed proposed extensions to be pushed back into user code or into external projects.

It wasn't until years later, that the team took onboard some additional vocabulary [Gabriel 1994; Steele 1998] to describe what they had been trying to achieve through Groovy's extensibility. Groovy was being designed neither to be a small or large language. It would be impossible to create

a perfect language, but by being extensible, users could grow Groovy to meet their needs and alter or tweak any deficiencies. More examples of extensibility will be seen in more detailed examination of the history.

4 LANGUAGE GOVERNANCE, HOSTING, AND COMMUNITY

There would be very few, if any, other languages that have been as successful as Groovy despite what could be perceived as a somewhat turbulent history. The founder left the project early on and project leadership has not remained constant since then either. The governance and hosting of Groovy also had to adapt when its initial hosting organization shut down. There have been a rotating set of organizations providing various levels of sponsorship. Despite this apparent turbulence, Groovy development has remained fairly stable. At all stages of its development it has remained a highly collaborative open source project driven more by the needs of the community than by a vision of a particular company or person.

4.1 Hosting

Initially the project established itself within The Codehaus open source project hosting site where it remained until that service closed its doors in 2015 [InfoQ 2015; Marx 2015]. Codehaus provided a source code repository, issue tracker and wiki. In 2011, the project also created a mirror of the repo on github. When Codehaus did shut down, the project had to find a new home. It was investigating several possible alternatives including The Apache Software Foundation [ASF 2020a], The Eclipse Foundation [Eclipse 2020], and The Software Freedom Conservancy [Conservancy 2020]. It decided The ASF was the best fit and upon being accepted, most infrastructure migrated as well. A mirror of the project repo is still maintained at github but now under apache.

While at The Codehaus, the source code repository technology changed from cvs to svn and then to the git distributed version control system. We'll see shortly in Figure 3, that switching from central to a distributed version control technology coincided with a significant increase in contributors. It made it easier for the project maintainers to track and merge contributions but also lowered the amount of work required to make a contribution.

4.2 Governance

When establishing itself within Codehaus, the project also adopted the Codehaus governance structure and manifesto [Codehaus 2015]. The Codehaus approach encouraged friendly collaborative decision making. Input and feedback was widely sought but there was recognition that contributors who had been involved more deeply with the project should have more say when deciding language direction and evolution.

After showing sufficient "merit", key contributors could be given committer rights to the repo and eventually considered for promotion to despot. Ultimate decision making lay in the hands of "project despots" with a project manager to adjudicate if consensus couldn't be reached, though that was extremely rare. Committers have full access rights to the repo but there was always an expectation that major changes would be discussed prior to being committed with the understanding that commits might be adjusted or reverted if subsequent review dictated the need.

The project would welcome input from the community for language improvements and new features. These suggestions might be ideas discussed on the mailing list, patches, and later Pull Requests (PRs). For more elaborate changes a Groovy Enhancement Proposal (GEP) is created. These can be separate documents or specially marked issues within the issue tracker [GEPs 2020]. In addition, Groovy developer conferences would allow the core team members to get together to discuss proposals and direction, brainstorm ideas, and hack prototypes.

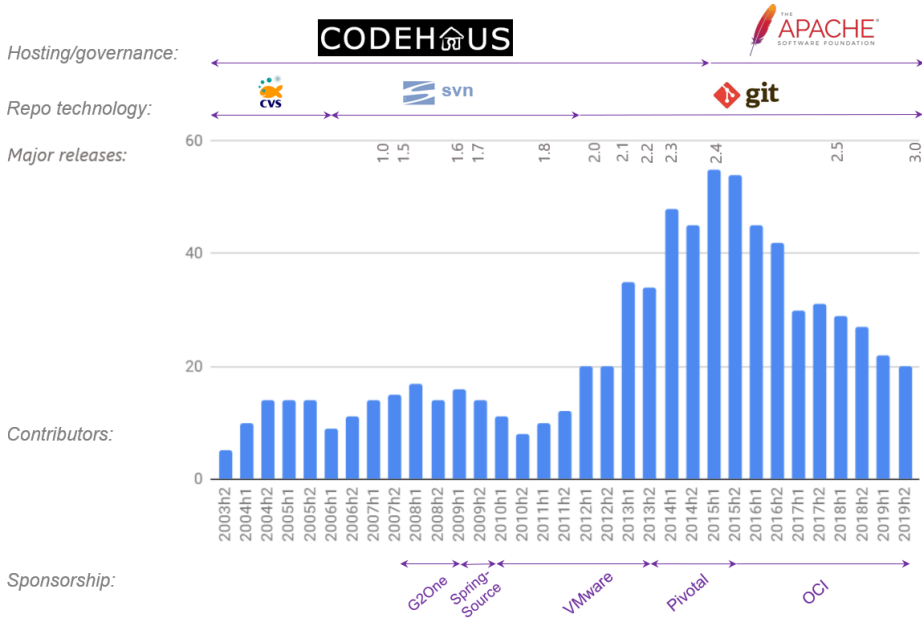


Fig. 3. Contributors to Groovy

If a new feature was suggested, the cost of implementing and maintaining the suggested feature would be weighed up against how much benefit it offered and how many users would be affected. If the majority of the despots thought the idea was worthwhile, that was a green light for the idea to be progressed. Sometimes the original proposer would progress the idea and others times one or more of the committers would also work on or even takeover implementation. Before being fully accepted, any new feature would need to potentially fit into the language specification and test suite, as well as comply with project coding conventions and quality standards.

When moving to The ASF, some aspects were quite different but many things remained largely unchanged. Apache has a way of working often referred to as “The Apache Way” [ASF 2020b]. As it turns out, the Groovy community was already following many of its aspects.

The role of committer has a more formal meaning with The ASF and some additional levels of structure apply to ASF projects [ASF 2020c]. Projects are lead by a Project Management Committee who are committers who have been elected to the PMC. There is an official Chair of the committee who reports to The ASF Board. Guillaume Laforge was the Chair when Groovy entered until November 2018. Paul King is the current Chair.

The ASF also has more structure around how releases are made and has an official process that needs to be followed including voting by PMC members. There is also a well-defined voting process that can be used for day to day decisions of the project.

4.3 Sponsorship and Contributors

Numerous organizations have been involved in sponsoring development activity within Groovy at different times. The major sponsors for regular contributors to Groovy are G2One, SpringSource, VMware, Pivotal and OCI (see Figure 3). ASERT and Canoo have also been major sponsors but on an ad-hoc basis. Corporate sponsorship has accounted for the involvement of 1–3 people at various times but as can be seen in Figure 3 there have always been a much wider range of contributors.

It's perhaps worth asking what is similar about the companies sponsoring Groovy. They were offering products, services or training related to Groovy or related frameworks like Grails and Micronaut which layer upon Groovy. They made money by helping customers build and test systems using these frameworks. Groovy has always been known as a language that helps people get things done and perhaps the large number involved in the project who spend time building systems explains why.

Apart from perhaps the first few weeks, the project has never been in a mode where all of the activity was being done by one person or one organization—even during periods of sponsorship. That being the case, the open source community model was always respected. One aspect of “The Apache Way” which also existed during the Codehaus era, but without a label, was that of “do-ocracy”. This means that while the project may have a rough roadmap, the people doing the work have some choice in which things they work on. Having sponsored developers doesn't mean that the sponsor can take over the project, but conversely, things which a developer or sponsor value are more likely to be included in the list of things which get done.

Lessons Learned about Community.

- Groovy has always maintained a friendly and helpful community spirit and culture. It's a wonderful community to be involved with and we should not ignore how that came about.
- Even when projects are evolving, if developer advocacy stops, the perception becomes that the language is stagnant or obsolete. Marketing needs to be done by anyone who can do it. This is certainly harder with lower levels of sponsorship but social media can be a low cost way to keep visibility and momentum.

5 ABRIDGED HISTORY OF LANGUAGE AND LIBRARY CHANGES SINCE 1.0

The Groovy language has continued to evolve since 1.0 and over a dozen major releases have been made (see Section C). We'll examine specific aspects of that evolution in Sections 6–9, but in this section, let's look at some of the main language and library changes, and the driving forces behind those changes.

This section isn't intended to be a detailed description or tutorial on the language. For that we suggest reading Groovy in Action [König et al. 2015] or the online documentation [Groovy 2020].

5.1 Groovy 1.5

Some highlights were:

- Various enhancements to target Java compatibility especially with new features in Java 1.5 (discussed in Section 8)
- The Elvis operator, where “`x ? : y`” is a shorthand for “`x ? x : y`” was added. It was inspired by the `||` operator from JavaScript and Ruby. The name Elvis was suggested by Jörg Staudemeyer, and endorsed by Laforge and others [Laforge 2007b]⁴.
- Bracket-free named parameter calls for improved DSL support, allowing Smalltalk-like syntax, e.g.:
`copy from: srcDir, to: destDir`
- Metaprogramming improvements to allow frameworks to more easily define extensions (see Section 5.1)
- A joint compiler to allow mixed Java/Groovy project to be easily compiled (see Section 5.1)
- A Maven plugin to compile Groovy code within a Maven build

⁴We later discovered that the gcc compiler also supports leaving out the middle argument of the ternary operator, resulting in similar experience, e.g. `x ? : y` being a shorthand for `x ? x : y`

Metaprogramming Improvements—or Grails as Groovy’s first Killer Application. It is often said that some *killer* application built using a language can bring with it the success of the language—perhaps Grails would do that for Groovy.

Ruby on Rails, or just Rails, was rapidly gaining mindshare in large sections of the web development community. It emphasized the use of other well-known software engineering patterns and paradigms, including convention over configuration and metaprogramming to allow rapid application development. This style of development was the inspiration for the Grails web framework and the Griffon desktop application framework both of which built upon Groovy. Graeme Rocher, creator of Grails, talks about Grails and its relationship to Groovy at the time [Rocher 2007]:

Groovy and Grails feel natural to a Java programmer. Groovy allows you to use a blended approach by mixing statically-typed Java for the complex tasks and dynamic Groovy for the easier things without requiring a huge mental shift as you switch from one paradigm to the other.

Most Java people who use Grails (or Rails for that matter) find it a breath of fresh air. If they are using common alternatives on the JVM like Struts and EJB, they will be facing a configuration-heavy approach. With Grails you can build an entire application and only ever have to configure the data source. Groovy and Grails are easy on the surface and allow you to be expressive and creative as a developer. Yet you can still piggyback on the underlying power of the JVM and frameworks like Spring and Hibernate.

Because of the amount of metaprogramming needed by Grails, Groovy’s metaprogramming capabilities were being put through their paces. As an example, consider the Book domain class given in Listing 13.

```
1 class Book {
2     String title
3     String author
4     Boolean paperback
5 }
```

Listing 13. A Grails domain class

To match the same kind of productivity offered by frameworks like Ruby on Rails, Grails needed to add automatic finder methods to the Book class as illustrated in Listing 14.

```
1 Book.findByTitleAndAuthor("The Sum of All Fears", "Tom Clancy")
2 Book.findByTitleLike("%Hobbit%")
3 Book.findNotPaperbackByAuthor("Douglas Adams")
```

Listing 14. Automatic finder methods

Groovy’s existing missing method hooks allowed attempts to call such methods to be intercepted and responded to appropriately. However, intercepting in this way was slow compared to if there was a real method in the class or a cached method known by the Book metaclass. Unfortunately, Groovy’s metaclass extensions relied on a pre-compiled class being on the classpath.

In September 2006, Graeme Rocher, lead designer for the Grails web framework, and also a Groovy committer, proposed an enhanced mechanism for defining class extensions via the `ExpandoMetaClass` [Rocher 2006]. It provided a convenient syntax for adding methods, constructors and properties to a class. For example, it is possible to add a `wordCount` property to any `File` object as shown in Listing 15.

```
1 File.metaClass.getWordCount = { delegate.text.split(/\w/).size() } // definition
```

```
2 new File('myFile.txt').wordCount // usage
```

Listing 15. Adding a wordCount property to File

Rocher first added the capability to Grails and offered for it to be moved to Groovy. The capability proved very popular within the Grails community, so after more discussions at a Groovy developer conference later that year in Paris [Devcon3 2007], the feature was moved from Grails to Groovy. This feature certainly provided a big boost to Groovy’s extensibility story.

Grails usage examples influenced other improvements within Groovy such as the ability to introspect available methods and properties including those added through metaprogramming using methods like `hasProperty` and `respondsTo`.

The Joint Compiler. Groovy recognised Java classes on the classpath and produced standard class files that could be made available to Java, also via the classpath. Hence Groovy and Java could be used together in hybrid projects but with some limitations. Handling complex inter-dependencies between Java and Groovy source files, might involve invoking the compilers in a particular order or splitting the codebase into smaller subsets or might not even be possible in some mutual dependency scenarios.

An early trick to handle inter-dependency scenarios was to define the boundary between Java and Groovy parts of the system via Java interfaces. Java code could reference the interfaces and the corresponding Groovy implementation classes could be injected during execution.

The joint compiler allowed for arbitrary and more flexible mixing and matching between the languages. Users could use both Java and Groovy in places which best suited each language, and easily change their mind and swap an arbitrary class from one language to the other if their initial choices didn’t pan out the way they expected. It worked by adding an extra preliminary step whereby Groovy source files were scanned, and class and method signature information was extracted into skeleton files called stubs. These stubs would be available to the Java compiler so as far as it was concerned it appeared that the Groovy files had been already compiled. Once the Java compiler was finished, the stub files were thrown away and the Groovy compiler produced the real Groovy class files.

This approach is a vast improvement over what existed previously but has some limitations which will be discussed later in Section 12.3.

5.2 Groovy 1.6

The main new functionality was the introduction of a compile-time metaprogramming capability also known as AST Transformations (covered in 7). As well as new methods in the GDK, the release provided 10 initial AST transformations making use of the new capability. Other highlights were:

- Multi-assignment support allowing expressions like:
`def (len, angle) = cartesianToPolar(x, y)`
- Support for writing annotation definitions in Groovy
- The grape packaging system (see Section 5.2)
- Improvements to `ExpandoMetaClass` when defining multiple extensions at once
- Various miscellaneous improvements including some enhancements for writing build-like steps in scripts (see Section 5.2)

The Grapes Package Management System. At the time, a pain point of Java was the complexity of managing dependencies on the classpath compared with the relative ease of using Python eggs or Ruby gems. We could make Groovy simple enough for non-developers to code in but they may choose not to use it if the classpath setup was too difficult. The grape packaging system was created using the `@Grab` annotation to declare a dependency using Maven or Ivy coordinates.

The dependency would automatically be downloaded and added to the classpath if needed during compilation and execution.

This capability wasn't meant to replace using a full-blown build system like Ant, Maven or Gradle for large projects but was great for making a single class or script be self-contained. You could email a script to a business analyst, database or system administrator, or tester, and they could cut and paste the script into the Groovy console, and run it without further setup.

The Influence of Build Systems. Another pain point in the Java ecosystem at the time was build systems. Ant and Maven were common JVM build tools at the time. They were XML based which was both a blessing (forcing a declarative style) and a curse (no way to script when needed). Groovy scripts provided excellent support for manipulating files and invoking operating system commands. This combined with Groovy's AntBuilder (and related classes) gave some real flexibility. But the scripting and DSL capabilities of Groovy could also help grander solutions being worked on at the time.

Hans Docktor, the founder of Gradle describes his interest in Groovy [NA Dockter 2019]:

I had been a Java developer for all my professional career. With the ascendancy of Ruby, I became interested in dynamic languages. At the time, I was not happy with the current build system landscape in the Java ecosystem and I decided to create a new build system. Groovy offered me everything I was looking for at that point in time. It was a language that had excellent interoperability with Java to integrate well with the Java tooling ecosystem and allowed for concise internal DSLs. Its dynamic capabilities allowed me to experiment with this concept in a serious project.

Gradle was initially 100% written in Groovy. For performance and tooling reasons, much of the code base of Gradle's core classes was later migrated to Java. During that process, I learned a lot about the pros and cons of dynamic languages; when to use them and when not. Groovy remained as the DSL layer for Gradle and many of the thousands of Gradle plugins are still written in Groovy. In recent times, we have also written a Kotlin DSL layer above the Gradle core and now also support writing Gradle plugins in Kotlin.

Early Gradle experiments acted as another driver into the design of Groovy's DSL capabilities.

5.3 Groovy 1.7

Some highlights of the release were:

- Further AST transformation improvements to support frameworks like Spock (see Section 5.3)
- Power asserts—originally a widely admired feature from Spock that provide great feedback when assertions fail. The feature was deemed to be of wide appeal and brought back into core Groovy.
- An AST browser was added to the groovy console allowing AST transform users and writers to inspect the output of transformation steps at every phase in the compilation process

The Influence of Spock. The Groovy-based Spock testing framework was innovating in many areas related to testing. It was also putting Groovy's AST transformation mechanism through its paces. As an example, consider the script shown in Listing 16.

```

1      @Unroll
2      def "maximum of #a and #b is #c"() {
3          expect:
4          Math.max(a, b) == c
5
6          where:
7          a | b || c
8          3 | 5 || 5
9          7 | 0 || 7
10         0 | 0 || 0
11     }

```

Listing 16. A data-driven Spock test

It looks like a mixture of declarative keywords `expect:` and `where:` and tabular data intermingled with an expression representing a desired condition. As far as the Groovy AST is concerned, it is initially one method having two labels, an expression involving `Math.max` and four expressions each involving the *binary or* operator (`|`) and the *logical or* operator (`||`). After the Spock AST transformation has taken place, the labels will be thrown away, there will be three standard JUnit method calls, each having a descriptive method name (for example “maximum of 3 and 5 is 5” for the first one) and containing an assertion involving `Math.max` with the appropriate parameters substituted⁵.

5.4 Groovy 1.8

This release included the largest number of new AST transformations added in any single release. As well as rounding out many design patterns and common idioms in use within Groovy code, some of the transforms were inspired by project Lombok [Kimberlin 2010] which also aimed to remove boilerplate code but through a Java pre-processor approach.

Other highlights for the release were:

Command chains discussed further in 3.2 allowed the language to support a wider range of DSL expressions.

Functional enhancements including forward and backward closure composition, trampolining, improvements to partial function application and memoization. Industry thought leaders were espousing the benefits of functional programming at the time so we believed it would be ideal to round out Groovy’s functionality in that area. In Groovy’s fight against boilerplate code, functional programming gives it a host of techniques which make many long-winded and cumbersome traditional OO design patterns fall away to simple Closure usage [Gibbons 2010; Norvig 1998].

JSON support was added since JSON was becoming a ubiquitous interchange format for applications

String improvements allowing multi-line slashy strings and a new dollar slashy string

GVars bundling to add dataflow, actors, parallel array processing, asynchronous functions and other functionality to the standard distribution. At the time, languages like Scala were debating whether to incorporate parallel and concurrent features into their core. Groovy decided not to merge GVars into the core but to keep it as an external library. Bundling in the standard distribution made access to that library a little simpler. Scala eventually took a similar stance with Akka being their equivalent library.

Performance improvements for primitive operations and some direct method calls.

⁵Explanation slightly simplified to ignore inconsequential details.

5.5 Groovy 2.0

This release included perhaps the biggest change ever in Groovy's history with the introduction of a static nature (see Section 9). It also included some JDK 7 inspired compatibility changes (see Section 8). Other highlights for the release were:

- Modular jar packaging so Android users, for instance, needing a small memory footprint, could exclude parts of Groovy they didn't need such as Groovy's Swing classes.
- *Extension modules* introduced some new conventions around the enhancements in the GDK allowing such enhancements to be placed within normal jar files—in fact Groovy's modular jars are packaged using this mechanism. The extension methods defined in extensions modules are available for both the static and dynamic natures.

5.6 Groovy 2.1

Highlights of the release include:

- Completion of invoke dynamic support (see Section 8).
- Improved Closure type checking allowing a greater range of DSLs and builder logic to be type checked
- Static type checker extensions (see Section 9.3)
- A meta-annotation facility (also known as annotation collectors) as a way to create one annotation as a combination of other annotations. This not only allowed us to simplify how we built annotations but offered a great extension point for the language as discussed in 7.1.

5.7 Groovy 2.2

This release contained many small improvements and enhancements as well as the usual bug fixes. Some highlights were:

- Automatic coercion of Closures to single abstract method (SAM) types (see Section 8.2)
- Scripting improvements for improved DSLs
- Type checking improvements
- Groovysh repl enhancements to perform completion in additional places like imports and filenames inside strings

5.8 Groovy 2.3

This release included the addition of traits (see Section 5.8). Other highlights were:

- Official JDK8 support
- NIO.2 (Java's New Input Output package) equivalents for all the classic IO extensions [GEP-62010]
- Improved type inference capabilities for Closure parameters

Traits. Traits are a structural construct of the language which allow composition of behaviors. Groovy's trait implementation was somewhat inspired from Scala. The implementation includes both a `@Trait` AST transformation for creating trait classes and trait composition functionality embedded as its own transformation step within the compiler. For most AST transformations, their impact is finished after compilation but the extra composition step made traits special. For this reason, it was decided to introduce the `trait` keyword. It better reflected that traits were an integral object-oriented capability of the language and also gave us more flexibility to change how it was implemented down the track if we wanted, and we did have some potential changes in mind (discussed further in 12.2).

Graeme Rocher, talks about re-writing Grails to make use of traits [NA Rocher 2019]:

In earlier versions of Grails, components like Controllers or Domain classes received added functionality through Groovy's metaprogramming capabilities. In Grails 3, the added functionality is obtained using the traits mechanism introduced in Groovy 2.3. Certain traits are automatically attached to the various kinds of components in Grails though you can also add them to your own classes. Using traits improves IDE support within frameworks like Grails.

5.9 Groovy 2.4

The headline feature for this release was Android support and it might seem at first glance that not much else was included. But under the covers, it was one of the biggest releases with many improvements across all parts of Groovy. The work on Android support led to many optimizations in bytecode generation; reducing the quantity of bytecode produced, lowering memory consumption of internal data structures and fine tuning bytecode for better performance.

Some other highlights were:

- Trait definitions could be constrained to apply to particular subtypes
- GDK additions and enhancements to make certain operations less eager
- AST transform improvements to improve consistency and fill in missing gaps in functionality
- Groovysh repl ease-of-use improvements

During the span of the 2.4 releases, the Grails project becomes a great source of feedback on traits usage as substantial parts of Grails is rewritten to use traits in many places where it previously used dynamic metaprogramming. Grails was now more competing with Spring boot applications rather than Ruby on Rails applications and the Grails re-write allowed it be a more compelling offering to that audience.

5.10 Groovy 2.5

This release included some packaging changes, some significant additions and reworkings of AST transformations (see Section 7.1) and the introduction of macros (see Section 7.2). Due to JDK9+ module restrictions, we removed the frequently used groovy-all jar and replaced it with a groovy-all pom. This had a big impact on our users but most have now adapted to use the component jars.

The Grails and Micronaut projects act as key users of the Groovy language and drivers of innovation during this time period. The Frege [2013] Haskell variant acts as an additional language influence on Groovy's functional features.

5.11 Groovy 3.0

The big ticket item for Groovy 3.0 is a new parser based on the Antlr4 parsing library. The earliest betas of Groovy used hand-crafted parsing code mostly created by Bob McWhirter. It was soon apparent that a proper parsing library would be key to evolving the parser and Jeremy Rayner was instrumental in converting the parser to use Antlr2. Antlr2 was state of the art at the time but it hasn't been updated since 2006. We had considered upgrading the underlying parser technology but it wasn't a straight forward upgrade and no-one had found the time to pursue the porting work.

In 2011, we initiated a Google Summer of Code project where a student would look at doing the port to Antlr3. The team discussed potential followup action at the Groovy developer conference Copenhagen in 2011 [Devcon6 2011] should the porting work prove successful but it didn't progress far enough to be pursued. In hindsight, the allotted summer break time given to such a student project was probably too short for the task at hand.

In 2014, we initiated a followup Google Summer of Code project to finish the work. This proved a useful proof of concept but again wasn't completed. In fairness to the student involved, the goal posts had moved a little further away since it was now Antlr4 that was the appropriate parser technology to target for the port.

In 2015, Jesper Steen Møller and later Sun Lan (Daniel) attempted to finish the remaining work [Møller 2015] but found the codebase was inefficient and hard to evolve and maintain.

In 2016, Daniel decided to re-write the parser from scratch and based it on a highly-optimized fork of Antlr4 and a corresponding mature and optimized Java grammar. Daniel gave the codename "Parrot" to the parser, since we had set a goal that when looking at the AST produced by the new parser it should look like it's repeating exactly what the old parser would produce.

Using the more flexible Parrot parser has allowed additional Java compatibility changes (see Section 8). The most important compatibility requirement was support for lambda and method reference syntax. In addition, the Parrot parser supports some new Groovy operators including Elvis Assignment, identity, and negated forms of `in` and `instanceof`.

Another highlight is improved functionality for embedded documentation which can now be accessed through the AST API and optionally embedded in class files.

5.12 Groovy 4.0

Planning is well underway for Groovy 4.0. Two big themes are consolidation and proper JDK9+ module support. There are some experimental activities including exploration of a Groovy equivalent to C#'s Language Integrated Query (LINQ) mechanism.

5.13 Lessons Learned in Evolving the Language

- Groovy has mostly relied on organic growth of its feature set. While this has mostly been a good thing (we are solving someone's real world problem) there have been times when in hindsight we could have designed a particular feature with a little more care and the feature could have been made more flexible.
- Our choice to disallow custom operator definitions is still one which we re-ask ourselves from time to time. We still feel that if programmers invent too many unreadable symbols that unmaintainable "ASCII art" code can be the result.
- Some parts of project memory are hard to maintain. Sometimes long discussions occur around a particular design decision. Some aspects of those discussions might appear in the documentation or within tests or are easily searchable in mailing lists. This might be when we chose a particular direction for a good reason. At other times, some aspect of the discussion may not be readily preserved - we might have avoided some other direction for an equally valid reason. Later when discussions ensue, we find it difficult to convey those lost discussion points. It must frustrate some new members on our mailing lists when we say we've tried some new suggestion before and it hasn't worked but we aren't really able to articulate the reasons clearly anymore.
- Command chains support nice looking DSLs but are currently one of the biggest causes of ambiguities in the language grammar. They add some complexity to the parser requiring detailed consideration of precedence rules and the need to turn to semantic predicates. We currently turn off command chains in just a few places like literal list and map declarations but if we had started command chains at the same time as introducing Antlr4, perhaps we would have been even more restrictive in where we allowed them.
- Groovy offers many extension points. We don't regret making most of these available to language users but each one has the potential to complicate tooling support. We work hard

to provide conventions and mechanisms to ease tool support but the extra flexibility doesn't come for free in all cases.

6 THE EVOLUTION OF THE GDK

Groovy offers a large number of enhancements or extensions to many Java classes. These are now known as extension methods or collectively as the GDK. There is a metaClass for every class that knows about the available extension methods. Execution of dynamic Groovy code involves method dispatch logic that will route execution to either a native class method or one of the extension methods. For statically compiled Groovy code, calling the extension method is woven directly into the bytecode.

Each new version of Groovy typically brings with it some new additions to the GDK. These might arise when new classes have been added to Java or when Groovy is being used in large scale projects in new ways, e.g. if a codebase is using many functional idioms or making heavy use of data science manipulation. While a crude measure, the number of extension methods in the GDK represents some of the value added by Groovy over Java. Figure 4 shows the count of those extensions over time.

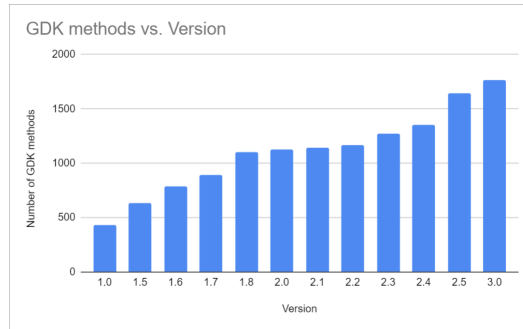


Fig. 4. The number of extension methods in the GDK per version

Some highlights in this history are:

Groovy 1.6 Extension methods could be flagged as requiring a minimum JDK version. They would be placed with a special plugin classes which would only be loaded when running on a particular JDK version or higher.

Groovy 1.8 The core extension methods were split into multiple classes.

Groovy 2.0 The classes were organized into modular jars.

Groovy 2.3 NIO extension methods were added.

Groovy 2.4 Many extension methods working on collective types were made less eager.

Groovy 2.5 This version was impacted by JDK8 and 9 changes:

- many extension methods were added for the `java.time` package
- modular jars gain Automatic-Module-Name information in the Manifest for JDK9+
- numerous modular jars become optional

Groovy 3.0 Includes part 1 of a two part process to redress split packaging issues.

Groovy 4.0 planning Part 2 of the process to redress split packaging with the modular jars targeted to become proper Java modules. Now that Java is evolving more quickly, it will be useful to also flag extension methods with a maximum JDK version.

What might not be apparent in this history is that over time, we have introduced steps to better manage and evolve the GDK. We have always had the ability to deprecate individual extension

methods that might no longer be of high value but we now also have a much improved ability to evolve module-size pieces of the GDK.

7 THE EVOLUTION OF AST TRANSFORMS

Groovy's compiler is built as a multi-phase pipeline with each phase made up of numerous internal transformations. The early phases of the compilation process involve parsing the source code into an Abstract Syntax Tree (AST) which is then transformed and enriched as each phase progresses.

The introduction of AST transformations, or AST transforms for short, involved providing standard hooks whereby user written transformations could participate in the compilation process. This capability was key in allowing us to scale the language [Steele 1998]. Users didn't have to wait for us to incorporate new ideas into the language and we felt less pressure to incorporate user proposed features; we knew they had a way to incorporate many such features themselves.

The motivation behind AST transformations was one of extensibility. Early prototypes of Groovy had included a property keyword and a `@Property` annotation to identify class properties before settling on the current conventions. The compiler automatically generates the necessary getters, setters and backing field boilerplate code associated with such properties (as per Java's JavaBean specification [Sun 1997]). This functionality was hard-coded into the compiler. In May 2007, Daniel Ferrin wanted to embellish property support by enhancing the compiler to generate further boilerplate code, in particular for the property change listener code associated with Bound properties [GROOVY-1884 2008]. The initial proposal was to add another annotation `@BoundProperty` and again have special logic within the compiler to handle the necessary processing. While there was agreement that the functionality was useful, the proposed implementation was rejected. Instead a more extensible solution was worked on for several months, including brainstorming at a Groovy developer conference in October 2007 [Devcon4 2007] and finally included into Groovy 1.6. `@BoundProperty` was discarded but AST transforms for `@Bindable`, `@Vetoable` and `@IndexedProperty` were added for bound, constrained and indexed properties respectively.

`@Bindable` is called a local transform. The presence of the annotation locally within the source code triggers the processing by the associated AST transformation. Provision was also made to support global transforms that always ran. These are typically used sparingly so as not to adversely affect performance.

Using annotations in the code provides a powerful declarative approach to constructing a program but has limitations. While most AST transforms can be configured using annotation attributes, like we indicated we wanted a copy constructor for our Book class in Section 1.3, not every scenario can be catered for and we don't really want to avoid writing many lines of code by having to write many lines of annotation attribute configuration. Also, you are limited to where you can place the annotation, e.g. typically on a class, method or field. Finally, there is the potential for adverse interactions when two transforms are injecting conflicting code into your class. We'll discuss this last limitation in more detail in Section 7.3.

A key motivating force behind AST transforms was as one strategy for managing the complexity of evolution of the Groovy language. During the 1.0 and 1.5 timeframes, many new features were proposed and discussed. Some of the features would require new keywords to be added to the grammar. We wanted a way to add new features that didn't impact the grammar or make the language overly complex. As well, we wanted a way for users to grow the language themselves if it didn't meet their needs. Figure 5 shows the count of those extensions over time.

The evolution of AST transforms from 1.6 through to 2.4 was mostly about adding in new transforms as new design patterns or idioms which could be expressed in this declarative way were found. During planning for Groovy 2.5, we wondered whether we were reaching the limits of what was possible using the declarative style of programming offered by transforms. The earlier

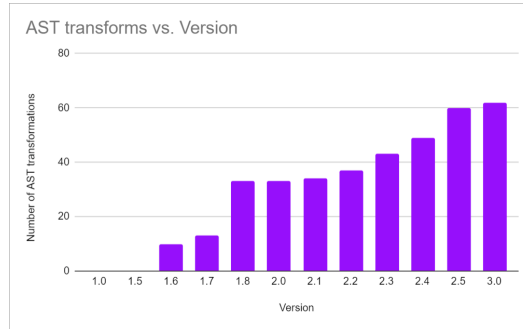


Fig. 5. The number of extension methods in the GDK per version

mentioned limitations appeared to be impacting potential further growth. But a major rework in 2.5 has made some significant improvements which we discuss shortly.

Groovy 1.6 included 10 initial transforms including:

- `@Immutable` annotation on classes which allows the declarative specification of classes with immutable properties. It generates bytecode corresponding to classes following the standard rules for creating immutable classes in Java [Bloch 2017, Item 17] (eliminating potentially 100s of lines of boilerplate code).
- `@Lazy` annotation on fields allowing declarative specification of deferred initialization of the field. It generates the best practice initialization code for the field in question [Bloch 2017, Item 83] potentially using double-checked locking, soft references (if needed) or the initialization on demand holder idiom.
- `@Delegate` annotation to automatically delegate part of the functionality of an owner class to the annotated delegate using the delegate pattern [Gamma et al. 1995, Introduction].

Groovy 1.8 introduced the largest number of new AST transformations added in any single release including:

- `@AutoClone` and `@AutoExternalizable` to simplify the boilerplate code often associated with cloning and externalization
- `@Log` and related annotations to support automatic addition of the common logging frameworks
- `@ToString` to generate a `toString()` method based on class properties (and potentially fields)
- `@EqualsAndHashCode` to generate `hashCode()` and `equals()` methods based on class properties (and potentially fields)
- `@TupleConstructor` which generated a constructor based on class properties (and potentially fields)
- `@Canonical` which combined the above three annotations

Groovy 2.2 added a `@Memoized` transform allowing caching of methods similar to the functionality already available for Closures

Groovy 2.3 include additions such as:

- `@Sortable` which injects a set of Comparators and sort methods
- `@TailRecursive` which provides similar functionality to Closure trampolines for recursive methods
- `@Builder` providing several common implementations for the builder pattern

Groovy 2.4 many improvements to improve consistency and fill in missing gaps in functionality

Groovy 2.5 meta annotations for fine-grained functionality, macros, macro methods

Groovy 3.0 Added @Groovydoc and @NullCheck

As well as Groovy's bundled transforms, framework requirements were a key driver for evolution. Graeme Rocher, creator of Grails and Micronaut talks about the impact of AST transformations [NA Rocher 2019]:

The use of AST transformations allowed us to move numerous features that were traditionally done at runtime to be done at compile time. For a metaprogramming heavy framework like Grails, this greatly reduces runtime overheads. For Micronaut, this capability is central to the compile time dependency injection capability which dramatically increases microservice throughput.

7.1 Moving to Meta-annotations

The @Immutable AST transform is powerful and injects much functionality. The project was receiving an increasing number of requests to make it handle further scenarios. We needed an improved approach to handle the evolution of that transform. Annotation collectors (meta-annotations) had been introduced in version 2.1 but was only used in a handful of places in Groovy itself. The approach proved useful in splitting out the functionality of @Immutable and @Canonical into their fine-grained pieces.

To handle some of the additional requested scenarios, we could combine the fine-grained pieces in new ways. Similarly it offered improved re-use for users of Groovy. If they have a special kind of immutable class and Groovy's @Immutable isn't quite suitable, they need now only write a small AST transform containing their unique logic. They can combine their special transform, perhaps some Spring or other framework annotations and a healthy collection of the fine-grained pieces that make up Groovy's @Immutable.

7.2 Macros and Macro Methods

An AST transform modifies a program by tweaking the AST, i.e. modifying the compiler's representation of the program. You could add a MethodNode to a ClassNode or add a Statement to the code for a MethodNode. Prior to macros, you needed to have a good knowledge of how Groovy code is mapped into the AST data structures to be able to write your transform. With Macros, you simply supply the Groovy code itself that you want to be added into the AST and it will be automatically converted into the needed AST data structures. There is provision for placeholders and various other options.

The goal is to open up AST transform writing to a bigger audience. An analogy can be made with Microsoft Office. There would be a limited number of people who would have the C++ background and necessary library knowledge to write a new native feature for Microsoft Excel but hopefully a much wider audience can enhance their spreadsheet using a VBA Macro.

There is also a related AST Matcher. While a macro lets you specify some replacement AST using Groovy code, the matcher lets you match an AST with a pattern specified as Groovy code (potentially with placeholders). Think regex but for AST writers. Writing your AST transform then becomes a search for places in the AST which look like one bit of Groovy code and replace it with some other bit of Groovy code.

Macro expansion is triggered by what can be thought of as a global macro method in the source code. It doesn't represent a method call at all but is fully expanded during compilation. Groovy users can also register their own global expansion methods, known as *macro methods*. This overcomes another limitation of the declarative annotation based approach of AST transforms. Macro methods can be used anywhere a method call could legally go not just anywhere that can be legally annotated.

7.3 Feature Interaction

Feature interaction [Calder et al. 2003; Wikipedia 2020], though often studied in the context of software features in telecommunications networks, has also been studied in the context of functional programming languages, for instance [Prehofer 1997] where feature interactions are resolved by lifting functions through monad transformers. The same issue affects Groovy programs including when using AST transforms and traits, and we believe is an interesting area for future innovation. The feature interaction problem overlaps with some of the concepts of macro hygiene in languages which support textual based macro substitution, but is more focused on behavioral interactions after various AST transformations are applied. Groovy's AST transformations work at the AST level, rather than source code level, so some aspects relevant to macro hygiene are less applicable.

As an example, suppose we create a `@Trace` AST transform which logs every method call with its parameters by injecting appropriate `println` or `log` statements at the start of every method. Now suppose we annotate a class with both `@Trace` and `@ToString` (which injects a `toString()` method with content determined by the class properties). Would we expect tracing information to be logged when the `toString()` method is called?

There is no correct answer to this question in general, different users of these transforms might want different behavior. What will happen? To answer you need to understand some of the workings behind transforms. AST transforms are assigned to a particular compiler phase by the transform writer. Although we don't guarantee it as part of the language specification, transforms within the one phase are currently executed in the order in which the respective annotations appear in the source code.

So, for our little scenario, we will see the tracing information when the `toString()` method is called if the phase for the `@Trace` transform occurs after the phase for the `@ToString` transform, or they are in the same phase and the `@Trace` annotation comes after `@ToString`.

Some frameworks overlay a priority annotation attribute onto Groovy's transform mechanics to allow the order for transforms at the same phase to be explicitly controlled. We have to date resisted adding this feature into the core language but it is something we could consider. Some transforms communicate with one another using node metadata—a capability we added to AST nodes. We could consider making more use of such information. We anticipate making more of the transforms idempotent—this will help in some scenarios.

Given the declarative nature of AST transforms, it is hard to see how the concepts from monad transformers could be applied directly but perhaps they would be an excellent parallel research activity for the language outside AST transforms.

8 THE EVOLUTION OF JAVA COMPATIBILITY

Groovy chose initially to remain close to Java in many areas. With each release of Groovy and each release of Java, Groovy has the opportunity to reassess that choice.

When Java adds a new feature we must ask:

- Will Groovy users likely want to use the feature?
- Is it automatically supported by Groovy?
- If changes are needed to support the new feature, what will be the cost to implement and maintain those changes, and are there any implications or interactions with existing Groovy features?
- Does the new feature resolve a Java pain point that Groovy has addressed previously and which can now be deprecated or possibly removed?
- Are there now new Java pain points?

Maintaining a close syntax with Java remains an often debated topic in the mailing lists. We should support *cut-and-paste compatibility* is a common catchcry. And true enough, Groovy's low learning curve for Java developers has been a key selling point for many adopters, so supporting the syntax they are familiar with is valuable. On the other hand, when Groovy offers different syntax or a different feature to Java, it does so because it believes that difference allows a greatly improved alternative in some particular scenario. Perhaps we should force all developers to use this better way, for their own good of course.

Let's look at how and why Java compatibility evolved with each Groovy release:

Groovy 1.5 Groovy 1.0 was targeted as being a better version of Java 1.4 but Java 5 had since been released and offered many new features. The project team thought that Groovy could be a great glue language for enterprise development but for that it would need to support generics, enums and annotations since frameworks like Hibernate and JPA made use of those features. Of these, generics was the most controversial (see Section 8.1). Cut-and-paste compatibility also led to the support of Java's classic and enhanced for loop, varargs and static imports.

Groovy 1.7 Added support for Anonymous Inner Classes (AIC) and nested classes. The team held off supporting AIC earlier since Closures often offer improved solutions but for cut-and-paste compatibility we finally added support.

Groovy 1.8 Added support for the Diamond operator—easy to do for dynamic Groovy since it was “ignoring” generics information anyway.

Groovy 2.0 Included numerous Java compatibility related features inspired by the Project Coin changes being introduced as part of Java 7 including: binary literals, underscores being allowed within number literals, and multi-catch statements. Also added was initial support for the new “invoke dynamic” bytecode instruction included with JDK 7. It is optionally turned on with a special flag and initially affects bytecode produced for method calls.

Groovy 2.1 Added support for automatic coercion of Closures to single abstract method (SAM) types (see Section 8.2) This release also included completion of invoke dynamic support. Previously, the invoke dynamic support didn't handle constructor calls and was missing numerous special cases like spread calls.

Groovy 2.5 Added support for numerous JDK8 inspired changes:

- Groovy had long supported repeated annotations within the source code but with the importance of annotations for AST transformation usage and the increasing use of annotations in enterprise frameworks, we wanted to support *repeated annotations* [JEP 120 2015] in the bytecode.
- Frameworks like Micronaut and Grails were beginning to use *method parameter names in bytecode* [JEP 118 2015] when available. Our roadmap also had automatic support of named parameters when such information was available.
- Java 8's *java.time* package [JSR 310 2014] was becoming more widely used so the GDK methods for Java's traditional Date and Calendar classes were extended to cover this new package and the @Immutable transform was updated to recognise the immutable classes in that package.
- Groovy has supported annotations in more places for some time, e.g. allowing annotations on import statements and package statements. It made sense to also support Annotations on Java Types [JSR 308 2014].

JDK9 JPMS module system also has had huge implications for Groovy. Groovy's modular jars weren't strictly compliant with newly introduced split packaging naming requirements and

more restrictive security requirements broke many parts of Groovy's duck typing functionality. Most of these issues have been addressed in recent versions of Groovy but the work is on-going.

Groovy 3.0 The Parrot parser has enabled support for additional Java syntax that hasn't been previously supported:

- Java's `do..while` loop.
- Enhanced classic Java-style for loop with commas.
- Java-style array initialization for non-ambiguous cases.
- Improved try with resources (Java 9).
- Nested blocks.
- Lambdas (Java 8: all the variants; some with Groovy enhancements).
- Method references (Java 8)
- Default methods in interfaces.
- Non-static inner class instantiation: `outer.new Inner()`.

Groovy 4.0 Completion of JPMS remediation work is planned for this release.

8.1 Supporting Generics

Generics enriched Java's type system and allowed the Java compiler to perform enhanced static type checking. But why would a dynamic language which ignores much of the type information at compile time want to support richer type information?

To make matters more confusing, due to Java's type erasure process, it appeared at first glance that the generics type information was going to be completely erased in the bytecode, so Groovy wouldn't even be able to do comparable checks at runtime. It was widely acknowledged within the team that supporting generics syntax would increase cut-and-paste compatibility between Java and Groovy but there was quite a debate that supporting the generics syntax might be more confusing than beneficial for a dynamic language.

It turns out that generic type information isn't completely erased. The java compiler emits a *signature* attribute on types and methods which retains some of the generic type information and this extra type information is what is used by enterprise frameworks like JPA and Hibernate. So support was added at the syntax level and indeed Groovy bytecode would include the required signature information.

In hindsight, this gave the project a much better starting point when it added a static nature to Groovy—as discussed in 5.5.

8.2 Implicit SAM Coercion

A timely introduction in the release was support for automatic coercion of abstract classes and interfaces containing a Single Abstract Method (SAM types) [GEP-12 2013]. This was originally to match exactly the same level of functionality that Java was going to introduce but Java later dropped non-interface SAM coercion [OpenJDK 2012]. The significance of this support is that Groovy developers would be able to automatically use Groovy closures with all of the new JDK8 APIs. This includes all of the functional interfaces which were being worked on as part of Java 8 which was introducing lambdas and the streams API among other things. This offered Groovy users a whole new range of libraries to use (with Closures) and would be a stop gap until Groovy 3 arrived with support for Java's lambda syntax.

8.3 Lessons Learned Maintaining Java Compatibility

- It might have been worthwhile thinking about our Java compatibility with a longer term view. Because we introduced new changes iteratively, each change seemed small enough that

it seemed worthwhile. If we look back over the entire history of Groovy though, the amount of engineering resource directed to this compatibility could have potentially been directed to some other killer feature.

- Sometimes what might seem like small Java changes impact us greatly. For instance, we used a less strict class package naming convention within our modular jars compared to the split packaging requirements of Java 9+ modules. This caused us to have to carry out some remediation work across Groovy versions 2.5 through 4.0. For us, this would have been hard to prevent since we adopted our conventions many years before JPMS planning. Perhaps if we more closely followed Java planning activities we could have caught this a little earlier.
- Our syntax compatibility with Java has been both a boon and a bane. It's great to minimize the learning curve but the downside is that we always needed to play catchup with Java, sometimes at the cost of not investing in other improvements or new features in the language. Furthermore, for a feature added in Groovy which eventually has an equivalent in Java, we have to potentially take on board an additional maintenance burden of supporting both the Groovy and Java ways of supporting the feature.

9 A STATIC NATURE FOR A DYNAMIC LANGUAGE

Groovy was originally meant to be a dynamic-only counterpart to Java. Users requiring static typing could continue to use Java when needed. Groovy would remain close to Java to allow switching between the two without a huge cognitive switch. Integration with Java would be good enough so that a developer could easily mix and match, using Groovy when they needed dynamic capabilities or wanted the succinct syntax options offered by Groovy and could use Java if they needed static type checking. Over time though, many Groovy language users found that they liked Groovy's succinct syntax and wanted to stay using Groovy all the time but they wanted better checking for the kind of errors that static type checking could find.

The language designers resisted for a long time to offer static type checking. After all, we had Java as our friendly companion language and what's more, IDE support had vastly improved and many of the errors that type checking would find were already being identified as potential issues with visual feedback in the IDE. The IDE might perhaps use a yellow warning highlight rather than some red error indicator like in Java, since it couldn't tell if the code involved some advanced runtime metaprogramming, but it gave a type-time⁶ visual clue nonetheless. Why debate runtime vs compile-time checking when you are already getting the feedback earlier at type time.

Improved type checking wasn't an ignored topic within the core team and informal discussions about it occurred regularly in the mailing lists and at conferences. More serious discussion about how it could be implemented began at the Groovy developer conference in 2011 [Devcon6 2011] which brainstormed how to give the compiler a "Grumpy" mode. Feedback from the community was also mixed with some wanting Groovy to remain focused on its dynamic capabilities [Winder 2011] while others wanted as much type checking as they could get even if that meant losing some of the dynamic features [GROOVY-8329 2009].

While opinions varied, the general consensus from the community was that Groovy needed a better static typing story. Other languages like Kotlin were gaining mindshare, Java was progressing more rapidly, and frameworks like Grails and Gradle were being used in more places where some extra type checking would be greatly valued. It became clear that it was now time for Groovy to make a change. Looking back in hindsight, we still feel delaying the introduction of static typing was the right thing. For such a big change, you don't want to be half sure you need it and it gave us plenty of time to think about how we'd introduce such a change.

⁶Here we refer to *type* as in *typing at the keyboard* not type systems.

Determining how best to introduce the static nature was certainly not obvious for some time. There were several not necessarily aligned goals:

- Groovy's dynamic nature was central to the language and should not be watered down to make provision for the static nature. In fact, if possible, existing code relying on Groovy's dynamic nature should not be affected at all.
- While the community wanted static typing, there was only a fuzzy idea about what that meant. They preferred not to change how they wrote their code and even in some cases where limited dynamic behavior was in play, they wanted extra checking if possible. In any case, some flexibility and extensibility was highly desirable.
- Others weren't interested in the type checking benefits per se but more in the possibility of performance improvements. If no dynamic behavior is in play, the bytecode could contain direct method calls rather than being routed through the Groovy runtime with its dynamic method dispatch.

9.1 Exploring Options

One widely discussed idea at the time for languages combining static and dynamic natures was to use *gradual typing* [Siek and Taha 2007, see in particular Section 8, Related Work], where variables declared with types would be statically type checked and others (like those declared with Groovy's `def` placeholder) would remain dynamic. While we liked some of the ideas behind gradual typing, it would break existing code. Groovy already had the concept of optional typing which allowed using either existing types or the `def` placeholder for dynamic code. The existence of the declared type indicated that the type would be checked at runtime rather than relying solely on the runtime type (duck typing). So the presence of types to indicate which code should be statically type checked was rejected.

Another option was that the compiler could have two modes. A special switch could put the compiler into a static mode where all code compiled would be statically typed. This was felt to be too coarse grained since in our experience many code bases would have a mixture of dynamic flavored code and other code where no dynamic capabilities were being used and which could benefit from static type checking. We also felt that intuition would suffer if it wasn't obvious when looking at some source code whether it was static or dynamic. We briefly looked at using a different extension for the statically typed code but we wanted to support having a mixture of dynamic and static code within the one class. So, compiler switches and different extensions were rejected too, but in fact both those mechanisms can be used in combination with the solution we eventually adopted.

We were already making frequent use of Java-style annotations. 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 dynamic⁷. A class could be annotated as a shorthand for annotating all methods. A class-level annotation could be overridden if needed at the method level. A static class could have one or more dynamic methods and a dynamic class one or more static methods for instance.

One outstanding question remained. When the compiler was in static type checking mode, should it just do type checking or should it do further optimizations that some folks were keen on that this extra type information might now make possible? Bytecode generation was done in a separate part of the compiler, so the question really was should bytecode generation be revamped too? It might seem that an obvious yes would be the answer to this last question. But it turns out that dynamic nature vs static nature is more a spectrum than a binary toggle. Groovy code falls right

⁷The idea had already been explored in the Groovy++ static nature spike which had the `@Typed` annotation.

across this spectrum but roughly falls into three categories: very dynamic code—all bets are off for trying to type check at compile time; simple static code—think Java-like code; and, what we might call hybrid code—where certain dynamic behavior is in play yet type signatures can be statically checked.

We now faced a dilemma, should we forgo optimizations so that such hybrid code could be type checked? Or, disallow type checking of such hybrid code so that all type checked code could be further optimized? We decided we didn't want to make either compromise and instead provided language users with two annotations:

- The `@TypeChecked` annotation would indicate that static type checking should be enabled [GEP-8 2011]. This is a great option for hybrid code or when performance is not a concern.
- The `@CompileStatic` annotation implied static type checking would be enabled but additionally indicated optimized bytecode should be generated that would call methods directly and avoid using the Groovy runtime [GEP-10 2011]. Essentially it would produce bytecode similar to what the Java compiler would produce and many forms of dynamic behavior are disabled.

9.2 Supporting Existing Coding Idioms

The other feedback we received was that while our users wanted static typing they didn't want to give up the succinct coding style they had become accustomed to while using Groovy. This posed a number of challenges and guided a number of our choices.

Groovy developers were used to optional typing and are happy to write code such as in Listing 17.

```
1 def animal = 'Bee'
2 assert animal.toLowerCase() == 'bee'
3 def animals = ['cat', 'dog']
4 assert animals*.toUpperCase() == ['CAT', 'DOG']
```

Listing 17. Type inference

Note, no declaration type is provided for the local variables `animal` and `animals`. We'd need to support *type inferencing*, including generic type information, to infer the type of `String` for `animal` and `List<String>` for `animals`. Hence the normal call to `toLowerCase` and the spread call to `toUpperCase` should both pass type checking.

Groovy developers are also used to writing code that makes use of duck typing such as that shown in Listing 18.

```
1 def col
2 if (someCondition) {
3     col = new ArrayDeque()
4 } else {
5     col = new Stack()
6 }
7 col.clear()
```

Listing 18. Duck Typing

In Java, we could use `AbstractCollection` as the declared type for `col` but that would not necessarily be natural for a Groovy developer. Groovy's static nature defines the *least upper bound* (LUB) of a set of classes as the union type of the common superclass of all classes and the common interfaces of all classes. In our example, the `col` variable has LUB of:

`AbstractCollection | Cloneable | Serializable` so we can safely call the `clear` method from `AbstractCollection` and we don't need `AbstractCollection` to be the declared type for `col`.

We should note that the LUB doesn't necessarily capture all possible allowed methods but only those that appear in some shared type artifact. For example, both `ArrayDeque` and `Stack` have a `peek` method but it doesn't appear in any superclass or interface. The use of `instanceof` can help us in such cases as illustrated in Listing 19.

```

1 def component = someCondition ? new ArrayDeque() : new Stack()
2 component.clear() // LUB: AbstractCollection or Serializable or Cloneable
3 if (component instanceof ArrayDeque) {
4     component.addFirst(1) // only in ArrayDeque
5 } else if (component instanceof Stack) {
6     component.addElement(2) // only in Stack
7 }
8 if (component instanceof ArrayDeque || component instanceof Stack) {
9     println component.peek() // checked duck typing
10 }

```

Listing 19. Duck Typing in combination with `instanceof`

Here we can call `clear()` directly since it is within the LUB. The `addFirst` and `addElement` methods are only available in the `ArrayDeque` and `Stack` classes respectively. The `peek` method is in both classes but not in any shared superclass or interface. The double `instanceof` check is smart enough to handle this extra case. Note, to match dynamic Groovy, there are no (explicit) casts in this code like there would be for Java. Dynamic Groovy would also allow the call to `peek` with normal duck typing but the Groovy type inference engine isn't smart enough to do this (yet).

Another important concept in the Groovy type checker is flow sensitive typing [Wikipedia 2018a], or *flow typing* for short. We developed this approach to support type checking for styles of programming commonly in use within existing dynamic Groovy code. We later found prior art in the Whyley language. The idea has been copied by other languages since then. Flow typing provides a smarter approach when type checking dynamic flavored code.

While it isn't necessarily preferred style, consider the code in Listing 20 which alters the value (and type) of a variable on subsequent lines.

```

1 def o = 'foobarbaz'
2 o = o.toUpperCase() // String
3 o = o.size()         // int
4 o = Math.sqrt(o)     // double
5 assert o == 3d

```

Listing 20. Flow Sensitive Typing

Here the inferred type flows from `String` to `int` to `double`. With flow typing, each line is correctly type checked and no casts are needed.

9.3 Type Checking Extensions

There are often cases where we are only using a small amount of dynamic behavior. Do we have to forgo all static checking in such cases. Consider for example, Listing 21.

```

1 class Bar {
2     String name() { "Bar is here" }
3     def methodMissing(String name, args) {
4         metaClass.invokeMethod(this, name.toLowerCase(), args)
5     }
6 }

```

Listing 21. Dynamic class

This class has a `name` method but also uses the `methodMissing` dynamic lifecycle hook to alter unknown method calls to the class to be lowercase. Dynamic code can call a `Bar` instance using method names like `name`, `Name` or `NAME`. This might not be preferred style for general programming but could be a relaxation allowed for business rules, tests or some other kind of DSL. Normally, static typing checks will ensure all method calls are validly defined and won't find the mixed case variants. We could document this as dynamic code where static typing isn't possible but instead Groovy offers an extension point on the static compiler. We write our type checking modification as shown in Listing 22.

```

1 methodNotFound { receiver, name, argumentList, argTypes, call ->
2     def result = null
3     withTypeChecker {
4         def candidates = findMethod(receiver, name.toLowerCase(), argTypes)
5         if (candidates && candidates.size() == 1) {
6             result = candidates[0]
7         }
8     }
9     result
10 }
```

Listing 22. Type checking extension

The details aren't important but there are a number of hooks we can use like `methodNotFound`. Each such method has predefined parameters we can use within the extension. In our case we search for lowercase variants of the method in question. We can use our extension as shown in Listing 23.

```

1 @TypeChecked(extensions = 'LowerChecker.groovy')
2 def method() {
3     def bar = new Bar()
4     println bar.name()
5     println bar.NAME()
6 }
```

Listing 23. Using a type checking extension

In our example, mixed case variants of `name` with no arguments will pass checking but if the incorrect parameter types are passed or a method call involving address or some other undefined method will fail. This extension mechanism allows numerous forms of hybrid checking to occur. The type checker can be made more lenient as shown here or stricter depending on our extension.

Type checking extensions can offer similar functionality to F# type providers [Microsoft 2020] as shown in Listing 24.

```

1 @CompileStatic(extensions='NatureServeAnimalTypeProvider.groovy')
2 def predatorObservations() {
3     species SandhillCrane eats ThistleGrasshopper
4     species SandhillHawk eats DesertPocketMouse
5 }
```

Listing 24. Two lines of code the second with a typo

The type checker, because of the `NatureServeAnimalTypeProvider` extension, is aware of animals that appear in the online NatureServe database [NatureServe 2020]. Such animals can appear within our code and because of the code within the type provider (not shown), become automatically defined properties in this example. If we supply an incorrect animal that doesn't appear in the

database, our code won't compile. Line 3 in Listing 24 is fine because `SandhillCrane` is a valid animal but `SandhillHawk` is a typo—there is no such animal, hence line 4 will fail compilation as shown in Listing 25.

```

1 NatureServe.groovy: 16: [Static type checking] - Unknown species: SandhillHawk.
   Did you mean: GrayHawk, FerruginousHawk, FiveSpottedHawkMoth, RedTailedHawk?
2 @ line 16, column 13.
3     species SandhillHawk eats DesertPocketMouse
4 1 error          ^

```

Listing 25. Error indicating the typo

The type checker extension can contain quite complex logic if you so desire, perhaps even calling into a theorem prover for instance. Depending on your tooling, you may not get assistance. In IntelliJ IDEA for instance, you need to register the type checker extension to obtain additional assistance.

9.4 Type Checking Groovy's Closure

There was another divergence between the direction Groovy had taken with its support of closures and where Java headed with lambdas which would need to be addressed in the type checker.

All Groovy's closures map onto a single `Closure` class. Execution of closures is funnelled through the class' `call` methods. There are variants with no, one, or more `Object` parameters. This design is simple and flexible, well suited to a dynamic language.

Java uses functional interfaces when determining target types for lambda expressions and method references. There are many common such functional interfaces, e.g. `Callable<V>`, `Supplier<T>`, `BiFunction<T, U, R>`. This design has many benefits for a statically typed language as the necessary typing information can be determined from the classes being used and the supplied type parameters.

The Groovy compiler should deal with lambdas⁸ in a similar way to the Java compiler but if Groovy users want to use Closures instead of lambdas, how is the typing information going to be determined, recalling that we only have `Object` as our type? Groovy supplies the `@ClosureParams` and `@DelegatesTo` annotations to convey the extra typing information. These annotations are quite powerful but we will give just a few examples of usage for some of Groovy's GDK methods as shown in Listing 26.

```

1 static void times(Number self,
2                   @ClosureParams(value= SimpleType.class, options="int")
3                   Closure closure) { ... }
4
5 static void upto(Number self, Number to,
6                  @ClosureParams(FirstParam.class)
7                  Closure closure) { ... }
8
9 static <T> int findIndexOf(Iterator<T> self,
10                           @ClosureParams(FirstParam.FirstGenericType.class)
11                           Closure condition) { ... }
12
13 static <T> int findIndexOf(T[] self,
14                           @ClosureParams(FirstParam.Component.class)
15                           Closure condition) { ... }

```

Listing 26. Closure Parameters for several GDK methods

⁸Available in Groovy 3 and above

Here `int` will be the inferred parameter type for the closure for the `times` method, `Number` for the `upto` method, `Iterator` and array variants of the `findIndexOf` methods. We can make use of such methods as shown in Listing 27.

```
1 @CompileStatic
2 def method() {
3     assert 1 == ['ant', 'bear', 'cat'].findIndexOf{ it.toLowerCase().size() > 3 }
4 }
```

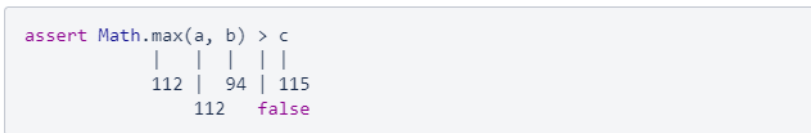
Listing 27. `@CompileStatic` usage with `@ClosureParams`

10 THE IMPORTANCE OF TESTING

Testing played an integral role during Groovy’s initial development. On the same day that James announced the project, he created a developer mailing list. In his first message [Strachan 2003c] to the list, he indicated he would use a test-driven development process for Groovy—writing the tests for Groovy in Groovy. Bootstrapping the compiler became a task of doing just enough so that the Groovy tests within the codebase could compile and run. The `assert` statement was one of the first statements the compiler supported [Strachan 2003d]. Testing was also seen as one of the areas where Java’s out-of-the-box experience could be improved. So, both by necessity and design, considerable testing capabilities were incorporated in Groovy from the start.

Similarly, when Groovy embarked on the next big stepping stone in its evolution, the JSR activity (see 2.1), testing was paramount. A large effort was put into creating the language specification test suite as part of the initial JSR activity. The evolution of this test suite lives on within the codebase today. In addition, most of the examples in the Groovy documentation are tested as part of the projects test suite.

Useful libraries (and frameworks) designed to work with a language are extremely important. Important for language users so they can more easily build their solutions and great for language designers since it often means that they can keep such code out of the main codebase. User requests for such functionality can be directed to the library. Having said that, every now and then some functionality in such a library is seen as being vital enough that it should come included always with the language. Such was the case when Peter Niederwieser developed the Spock testing framework [Niederwieser et al. 2020]. Spock’s power assert feature (see Figure 6) was deemed to be one of those vital features.



```
assert Math.max(a, b) > c
           |  |  |  |
          112 | 94 | 115
             112  false
```

Fig. 6. Power Assert error reporting

The core team discussed with Peter about this possibility and he agreed to incorporate it into Groovy itself [GROOVY-3425 2009; Niederwieser 2009].

Testing remains an important part of Groovy in recent versions:

- Assertions are built in and always enabled
- Power asserts provide friendlier assertion error messages
- JUnit 3–5 functionality is included in the standard distribution; Groovy and its tools like the Groovy Console can automatically run such tests directly just like a script
- Groovy provides various enhancements to JUnit

- Groovy has inbuilt support for mocks and stubs using a unique metaprogramming-based mocking approach
- Groovy provides many ways to create test collaborators and dummies

10.1 Testing as Part of Community Building

Testing has also played a central role in supporting Groovy's open source community approach to development. Groovy has a large test suite which embodies much knowledge about how the language and libraries are supposed to work. This not only gives us confidence when embarking on significant refactoring activities but is key to having a broad range of contributors to the language.

The pool of people in the world who know the whole Groovy codebase and understand exactly how their contribution to one part of the codebase may impact other parts of the codebase is small. Luckily, we don't have the expectation that all our contributors come from such a pool. We do however expect our contributors to execute the test suite before submitting proposed changes. It helps us vet incoming contributions but also helps contributors create a pull request that they can have some confidence about. Of course, we certainly review all contributions as well. Manual reviews can cover aspects like the need of the feature and overall consistency of its implementation.

Groovy has numerous quality checks over and above the test suite. Static code analysis, code coverage and binary compatibility checks are also performed. Continuous integration (CI) servers kick-off whenever code is checked in or pull requests received. Various branches of Groovy across various Java versions are checked as are test suites for a handful of important projects within the Groovy ecosystem.

Testing has also played an important role in increasing Groovy's adoption. Conservative organizations may not be willing to try using a new language for their production code but are often more open to using a language like Groovy for testing. Once they become familiar with Groovy for testing they often use it in other places.

10.2 Testing Lessons Learned

- The value of an extensive test suite should never be underestimated. It provides us with the confidence to do bold refactoring steps and helps us vet contributions even from newcomers.
- Most Groovy documentation embeds tested code examples. This has proved invaluable in ensuring that the documentation is kept up to date.
- CI servers play an important role to ensure we don't break backwards compatibility or expected behavior of the language.

11 HEALTH STATUS

There are numerous language rankings which compare the popularity of programming languages. According to the TIOBE index [TIOBE 2019] for October 2019, Groovy is the 11th most popular programming language putting it ahead of languages like Go (14), Ruby (15), Swift (16), R (19), Perl (20), Scala (30), Kotlin (37), and Clojure (within 51-100 ranking cluster). The RedMonk index [RedMonk 2020] places the languages at Ruby (7), Swift (11), R and Scala (tied 13), Go (15), Perl (18), Kotlin (19), and Groovy (23). Such rankings factor in difference source statistics and so have a high degree of variability but they are also flawed in other ways. As just one example, most rankings include some statistics based on number of searches including the language or number of questions answered in forums—so a simple language, or a well documented one, will be penalised for that statistic. Never-the-less, these rankings indicate that overall interest in Groovy remains high.

A legitimate question to ask is whether the project is in a healthy state so that it can maintain its relatively high ranking over the long term? We can't really answer that question but what we can do is look at metrics we know about.

The project is often asked how many Groovy users are there? This is another question for which we don't know the answer. We do know that Groovy artifacts have been downloaded at least a quarter of a billion times and the trend is for increasing downloads each year. We have download statistics for approximately the last 7 years (grouped into every half year).

We also have some other project activity metrics including the number of commits, the number of issues resolved in the issue tracker, and the number of releases. We have these metrics since the project inception (grouped here every half year) and while all of these metrics have certain flaws, there are some things we can learn.

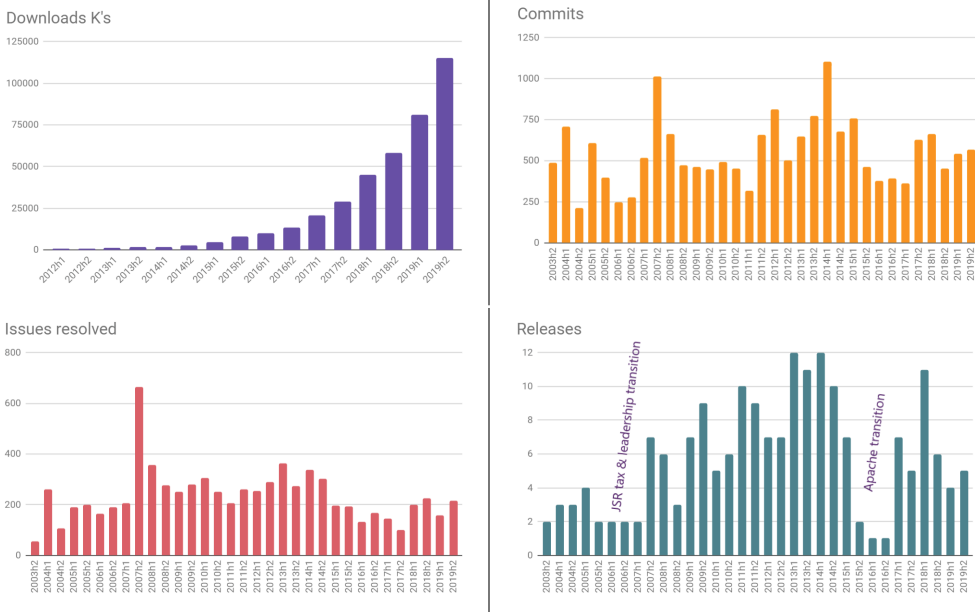


Fig. 7. Groovy project statistics

Issues resolved is not a great metric. It could indicate an active language or just one with lots of bugs—but in either case, enough folks in the community are involved enough to suggest enhancements and log bugs, and some progress is being made on those requests.

The number of commits is also a poor metric. Some commits might involve adding a large new language feature or a major refactoring, others might fix a typo. But still over multi-month time-frames we expect a mix of all sizes of commits and an overall picture can be seen that activity has happened at a relatively constant pace over the entire lifetime of the language regardless of the changing circumstances in which the project has found itself.

From the number of releases we see that 2006 and 2016 were not the best years for the project. The former involved an intense focus on the JSR activity and the project leadership transition from James Strachan to Guillaume Laforge. The latter was a transition year where we were learning “The Apache Way” and reworking our previously almost fully automated release process to allow the formal voting steps required by the ASF. After taking on board that learning, release cadence is back to similar levels to what they were before joining the ASF.

Lessons learned about releases:

- Our community was understanding when we delivered imperfect functionality so long as we communicated what we were doing and didn't take too long between releases. They preferred that we release early and correct if needed rather than delay excessively trying to craft the perfect release.
- Despite how well we try to advertise alpha and beta releases, it's hard to get feedback on them. We'll sometimes have a feature that has been in a beta for many months, and of course it will be the day after the final release that a rather important and easy to spot bug surfaces. We must be ready to re-release without delay in such circumstances.
- For languages, at least mature ones, backwards compatibility is an important balancing force to "release early release often". Making bug fix releases we should do often. Making breaking changes if needed needs to be more controlled.
- Maintaining binary compatibility without automated checks is hard. Developers on open source projects often do work in sporadic bursts, and may not remember all the quality checks and analysis that would be ideal. Baking them into the quality checking parts of a CI build helps dramatically.

In summary, these metrics seem to indicate that the language is progressing in a steady fashion.

12 ON-GOING CHALLENGES AND MISSED OPPORTUNITIES

Groovy is by no means a perfect language and we know its flaws as well as anyone. There are a few items we are keen to address in future releases.

12.1 On-again Off-again Focus on Performance

In the early releases, Groovy could be quite slow in various scenarios. A method call in Java can be exactly determined at compile time and may end up being 1 or a small number of bytecode instructions. In Groovy, this might involve 10s or 100s of bytecode instructions since the runtime must check for metaclass enhancements, handle the late dispatching required for multimethods, and account for various lifecycle hooks that might be involved in the call.

On the one hand, Groovy has espoused object model purity and business logic consistency as trumping performance when selecting for instance `BigDecimal`s as the default type for certain division calculations. Then, when the performance is slow, we have cached, optimized, re-worked various scenarios so as to be at least competitive with Java and more recently the same as Java when using `@CompileStatic`.

This means that under different circumstances, we might produce different bytecode variations for the same line of source code. There might be a normal path, a path where certain calls can be optimized, one where primitives are optimized, one with invoke dynamic support and another for compile static. In Groovy 4.0, we are planning to consolidate these paths.

12.2 Traits and AST Transforms

Traits are not officially compatible with most AST transformations. Some of them, like `@CompileStatic` will be applied on the trait itself (not on implementing classes), for others it is not clear whether the intent is to apply the transform on the trait, the class implementing the trait, or both. Perhaps it is clearer with an example. Consider a trait annotated with the `@ToString` transform. A language user seeing this could wonder which of two possible intents apply:

- The trait will gain a `toString` method printing out the properties defined just within the trait. Implementing classes will weave in this method—they are inheriting behavior after application of all transforms.

- The `@ToString` annotation is meant to be part of the trait definition in its current form and will be applied after initial weaving. In this scenario, all implementing classes will end up with a `toString` method but it will print out all properties of the implementing class, not just those coming from the trait.

As a language, we can easily pick one or the other but we see valid cases for both scenarios. Our initial “*compatibility*” disclaimer buys us some time to see how usage unfolds and lets us explore whether we can allow both behaviors to somehow be available.

12.3 Joint Compiler

The current implementation of the Joint compiler makes some compromises. It produces stubs during an early stage of the Groovy compiler before calling the Java compiler with those stubs on the classpath. Once the Java compiler completes, normal Groovy compilation continues with all of the classes from the Java compilation step on the Groovy classpath. The issue is that at the early phase at which the Java compiler is called, not all transforms have made their changes yet to the AST, so the generated skeleton stubs are incomplete. The phase can’t be moved later because some of those transforms will want access to the Java class files. This is not a problem in some scenarios but if both the Groovy and Java parts of the codebase want access to the missing stub methods, the process will fail, and the earlier workarounds need to be reinstated.

In Groovy 4.0, further work will be done to allow transforms to mark themselves as being idempotent or as being able to be safely called early. This will alleviate some of the issues. We have discussed approaches more similar to how the Eclipse compiler works effectively calling into the Groovy and Java compilers in a more fine-grained fashion. If we have sufficient people interested in helping, we’ll explore this too.

12.4 Android Development—a Missed Opportunity

As we saw in Section 4.3, despite sponsorship changes, Groovy has maintained a fairly steady number of contributors over its lifetime. What the figures don’t reveal though is that large scale language changes have tended to occur when the project has larger numbers of sponsored committers. For Groovy after Pivotal ceased sponsorship, some areas of progress slowed but it was Android support which fell into a hole. While the basics were done, it needed to be marketed at conferences, and it needed improvements and evolution as the Android world was constantly evolving. At the time there was much interest in the improved developer experience that Swift provided for iOS development compared to Objective-C. Groovy had the opportunity to be the Swift of the Android world but for now at least it appears Kotlin is now the preferred language for such development.

12.5 Learning to Say “No”

The hardest thing at times for an open source project is to say no. Fresh ideas and enthusiasm can’t be reason enough to justify new additions to the language. In many cases, the team suggested that certain ideas be progressed as an external library which could be incorporated into Groovy core at a later date if it proved worthwhile. There are numerous cases where this approach worked well:

- Extensions for Java’s `java.time` package [JSR 310 2014] started life as a separate Goodtimes project [Goodtimes 2017] and were later merged.
- The Spock testing framework was kept as a separate project but basic power assert functionality was merged into the Groovy project (see 10).
- Groovy++ was an initial spike for how a static nature to Groovy might look written by Alex Tkachman [Tkachman 2011]. It was never merged but some of its ideas were inspiration when Groovy added its static nature.

- @Sortable was inspired by a similar AST transform in the Griffon desktop application framework [Griffon 2020].
- The ExpandoMetaClass started its life in the Grails project but was later moved to Groovy.
- The GContracts design-by-contract library [Steingress 2020] supports class invariants, and pre and post conditions. It was kept as a separate project but Groovy was adapted to support supplying code (a closure) as an annotation attribute value to support this library. The feature is now used in numerous other places including within the Groovy project itself.

12.6 Coping with Big Tasks in Open Source

Many of the statistics from Section 11 seemed to indicate that Groovy development was fairly stable even in light of fluctuating sponsorship arrangements. While that is mostly true, it fails to recognize one important factor. Throughout its lifespan, Groovy has had a mixture of small, medium and large changes. While there have certainly been far fewer of the large kind than smaller ones, it is the large ones which are most impacted when the project is mostly being progressed by individuals and volunteers. We saw in Section 5.11 that the Parrot parser took over 8 years from inception of the idea to actual final release. During the times when the Groovy team did have multi-person sponsorship, that task had not been a priority of those team members.

Adding a static nature to Groovy was also a large change but was completed in a much shorter time-frame. While Cédric Champeau did a substantial amount of the work on that change, the fact that there were multiple sponsored team members at the time meant that other important activities within the project could be handled as well as progressing the large change.

Another large change which has been on Groovy's roadmap for a long time is a re-write of the Meta-Object Protocol (MOP). This is the set of rules used for property and method selection. It is more complex than it might sound. Are we dealing with a Java or Groovy object? Are we inside a category block? Are metaclass enhancements in play? Are optimizations in play? Does the property or method exist and if not, does the object in question have lifecycle hooks? And the list of questions goes on. The logic in the existing MOP grew organically. It has many branches in the logic and numerous special cases. There has long been a desire to re-write it. It has been discussed at numerous Groovy developer conferences [Devcon5 2009; Devcon6 2011; Devcon7 2012; Devcon8 2013; Devcon9 2013; Devcon10 2014] and many times on the mailing lists over the years but has failed to progress. It is a difficult task to break into smaller pieces but remains on our roadmap for Groovy 4.

13 INFLUENCING OTHER LANGUAGES

As to be expected, there is a healthy flow of good ideas between languages. Guillaume's conference presentation [Laforge 2017] gives a nice overview of languages which introduced a similar Groovy feature at a later time. In some cases we know there was a strong direct influence by Groovy. In other cases, we don't know the details. There might have been inspiration from Groovy or not. In any case, Groovy helped popularise some ideas so that even if inspiration was elsewhere, language users had become familiar with the concept. Languages introducing features similar to Groovy at a later date include:

- The dangling closure⁹ syntax is used by Scala, Kotlin, and Swift.
- The safe navigation operator [Wikipedia 2018b] is used by Swift, Ruby, C#, Kotlin, CoffeeScript, and others.
- Range notation is used by Swift.

⁹Also known as trailing closure

- The Spaceship operator, somewhat inspired from Perl, is now also used by Php, Ceylon, and Ruby.
- The Elvis operator `'? : '` is also used by Fantom, Kotlin, Gosu, Xtend, and Ruby. Swift and C# use `'??'`.
- Flow sensitive typing is also used by Ceylon, Kotlin, Facebook Flow, and TypeScript.

Kotlin borrows from Groovy in numerous ways [NA Breslav 2020]:

- Groovy's dangling closures inspired Kotlin's dangling lambdas (see also [Breslav 2018, Slide 29, Video 32:15])
- Groovy's builder concept inspired Kotlin's type-safe builders (see also [Breslav 2018, Slides 32/33, Video 36:30])
- Kotlin's default `'it'` parameter for lambdas was borrowed from Groovy (see also [Breslav 2018, Slide 29, Video 32:25])¹⁰
- Kotlin's string template syntax followed Groovy syntax used in its interpolated strings and templates
- Kotlin's Elvis operator was inspired by Groovy's similar operator (see also [Breslav 2018, Slide 39, Video 47:50])
- Kotlin's null-safe call was inspired by Groovy's null-safe navigation operator (see also [Breslav 2018, Slide 39, Video 47:50]) (but with a different approach for non/nullable types)

Kotlin has numerous other similarities with Groovy:

- Kotlin's data classes and lazy mechanisms, while Scala and C# inspired have many similarities with Groovy's `@Immutable` and `@Lazy` transformations. Kotlin bakes these into the language whereas Groovy makes use of AST transforms.
- Kotlin uses a similar method name convention for operator overloading
- Kotlin followed Scala's stance on optional semicolons
- Kotlin has a feature similar to Groovy's import aliasing (but mostly Python and Scala inspired)
- Kotlin syntax for smart casts (inspired from Nice or Gosu) piggy-backs on the idea of flow typing. Groovy's flow typing (inspired from Whiley) leads to similar code simplifications.

Groovy has also borrowed ideas from Kotlin.

- The type safety aspect of Kotlin's type-safe builders is incorporated back into Groovy as a static feature with some slight differences
- Groovy has added extensions methods like `shuffle` and `average` inspired by Kotlin's similar methods

14 WHAT'S NEXT FOR GROOVY?

There are many Groovy developers who find value in using Groovy—we must keep offering them useful functionality. Java, and other JVM languages, will continue to evolve but Java and Groovy developers will continue to have pain points. Groovy's extensibility makes it an ideal language to find innovative solutions to their issues. Making the language extensible has been core to Groovy's design and evolution, and will remain so. We do need to improve selling certain features of Groovy to its user base such as when to use dynamic vs static aspects of the language.

Most importantly, we'll keep listening to our community. They so far haven't failed to keep coming up with new ideas for us to tackle.

¹⁰Groovy had made use of the `'it'` parameter since before version 1.0 [Strachan 2003e].

ACKNOWLEDGMENTS

Groovy has had three project leads, many members at different times who have been part of the Groovy core team, more than 350 total contributors to the project's codebase, and hundreds of other folks who have contributed in various ways within the broader Groovy community. Without all of these contributors, Groovy would not be the language it is today. The author would like to thank Andrey Breslav, Hans Docktor, and Graeme Rocher for their feedback on Groovy influence on Kotlin, Gradle and Grails/Micronaut respectively. Thanks also go to Andres Almiray, Cédric Champeau, Ralph Johnson, Dierk König, Guillaume Laforge, and Daniel Sun for their valuable comments on drafts of this paper, and Guy Steele and Yannis Smaragdakis for assistance during the publication process.

A NON-OVERRIDABLE OPERATORS

Groovy has a number of operators which can't be overridden. The Elvis and Elvis assignment operators are also just shorthands and likewise Regex operators are not overridable.

Operator	Name	Usage/shorthand for
a.b a.c()	Dot operator	member navigation
a?.b a?.c()	Null-safe operator	expression is null if a is null
a.@b	Field/attribute access	access field directly bypassing getter/setter; attribute access in XML ¹¹
4..5 'a'..'z'	Range notation (inclusive)	new IntRange(true, 0, 1) new ObjectRange('a', 'z')
4.<6	Range notation (exclusive)	new IntRange(false, 0, 1)
target.&method	Method pointer	A closure backed by the specified method
target::method	Method reference	Similar to Java's method reference
collection*.p	Spread-dot (property)	collection.collect{ it.p }
collection*.q()	Spread-dot (method)	collection.collect{ it.q() }
[item1, item2, *list]	Spread operator	spreads items from collection into outer collection
[k1:v1, k2:v2, *:map]	Spread-map operator	spreads items from map into outer map
a ?: b	Elvis operator	a ? a : b
a ?= b	Elvis assignment	a = a ?: b
string =~ regex	Regex find operator	Pattern.compile(regex).matcher(string)
string ==~ regex	Regex match operator	Pattern.compile(regex).matcher(string).matches()
~string	Regex pattern operator	Pattern.compile(string) (re-purpose of bitwiseNegate)

B SUPPLEMENTARY LISTINGS

The GroovyConsole can be configured in numerous ways including output customization scripts. A simple output customization script is shown in Listing 28.

```

1 import javax.swing.JLabel
2 transforms << { result ->
3     if (result.class.name.endsWith('RealMatrix')) {
4         return new JLabel(matrixAsHTML(result))
5     }
6 }
```

Listing 28. Optional console output configuration script

Where `matrixAsHTML` is a pretty printing method that prints out the matrix data in HTML table format (which Java's `JLabel` class supports).

¹¹This is really just a naming convention as first character of property name

A slightly more sophisticated customization script using a library which renders a \TeX representation of the matrix as an image is shown in Listing 29.

```

1 import org.scilab.forge.jlatexmath.TeXFormula
2 import static org.scilab.forge.jlatexmath.TeXConstants.*
3
4 transforms << { result ->
5     if (result.class.name.endsWith('RealMatrix')) {
6         new TeXFormula(pmatrix(result)).createTeXIcon(STYLE_DISPLAY, 20)
7     }
8 }
9
10 def pmatrix(m) {
11     /\begin{pmatrix}/ +
12     (0..

```

Listing 29. A fancier output customization script

The Groovy compiler can be configured in numerous ways. As an example, we could add some compiler customization to indicate that we want:

- we want to use the `MatrixBase` base script for scripts
- we want an automatic import of our `RealMatrix` class but with a more friendly `Matrix` alias
- as one option, we can apply these changes only for scripts having the extension “`.mgroovy`” so our changes for matrix-flavored Groovy scripts won’t impact other scripts

These configuration options are shown in Listing 30.

```

1 withConfig(configuration) {
2     source(extension: 'mgroovy') {
3         configuration.scriptBaseClass = 'MatrixBase'
4         imports {
5             alias 'Matrix', 'org.apache.commons.math3.linear.RealMatrix'
6         }
7     }
8 }
```

Listing 30. A Compiler Customization Script

The details of this script aren’t important. Groovy has numerous classes for customizing the compilation process and this configuration script itself is a mini DSL using those classes. The script is amenable to parsing by the compiler but also IDEs and other tooling. This is further discussed in Section 3.4.

C RELEASE INFORMATION

Version	Released	Major contributors
1.0	January 2007	James Strachan, Jochen Theodorou, Guillaume Laforge, John Wilson, Jeremy Rayner, Dierk Koenig, Pilho Kim, Sam Pullara, Boc McWhirter, Paul King, Christian Stein, Russel Winder
1.5	December 2007	Paul King, Jason Dillon (including major reworking of the groovysh repl), Jochen Theodorou (including some major compiler enhancements)
1.6	February 2009	Paul King, Danno Ferrin (including SwingBuilder, Grapes, and much of the initial plumbing for AST transforms), Jochen Theodorou, Alex Tkachman (including numerous AST transformations and callsite caching logic)
1.7	December 2009	Paul King, Roshan Dawrani, Jochen Theodorou
1.8	April 2011	Paul King, Roshan Dawrani, Guillaume Laforge, Jochen Theodorou. Václav Pech, the key author of GPars, was also a key contributor to several of the functional enhancements.
2.0	June 2012	Cédric Champeau (a key driver behind the design of Groovy's static nature), Paul King, Jochen Theodorou, Guillaume Laforge
2.1	January 2013	Cédric Champeau (including guiding the work on type checker enhancements), Paul King, Jochen Theodorou, Guillaume Laforge
2.2	November 2013	Pascal Schumacher, Paul King, Cédric Champeau (including much of the SAM coercion implementation), Jochen Theodorou, Thibault Kruse
2.3	May 2014	Cédric Champeau (a key driver of the design for traits), Paul King, Guillaume Laforge, Jochen Theodorou
2.4	January 2015	Paul King, Pascal Schumacher, Cédric Champeau, John Wagenleitner, Jochen Theodorou, Thibault Kruse
2.5	May 2018	Paul King, Sun Lan (Daniel), Pascal Schumacher, Cédric Champeau, John Wagenleitner
3.0	February 2020	Paul King, Sun Lan (Daniel) (including the majority of the Parrot parser), Eric Milles

REFERENCES

- Tiago Antao. 2008. Chloroquine malaria treatment and Groovy (DSL tactics in Groovy 2). Archived at <http://web.archive.org/web/20080302023038/https://tiago.org/ps/2008/02/27/chloroquine-malaria-treatment-and-groovy-dsl-tactics-in-groovy-2/>
- Tiago Antao, Ian Hastings, and Peter McBurney. 2008. Ronald: A Domain-Specific Language to Study the Interactions Between Malaria Infections and Drug Treatments. Las Vegas, USA (Jan.), 747–752. *Proceedings of the 2008 International Conference on Bioinformatics and Computational Biology, BIOCOMP 2008* (Jan.).
- ASF 2020a. The Apache Software Foundation (website). NON-ARCHIVAL <https://www.apache.org/> (also at [Internet Archive 10 March 2020 11:41:12](https://www.archive.org/details/apache-software-foundation-20200310)).
- ASF 2020b. The Apache Software Foundation Briefing: The Apache Way. NON-ARCHIVAL <https://www.apache.org/theapacheway/> (also at [Internet Archive 18 Feb. 2020 08:55:52](https://www.archive.org/details/apache-software-foundation-briefing-the-apache-way-20200218)).
- ASF 2020c. The Apache Software Foundation: Corporate Governance Overview. NON-ARCHIVAL <https://www.apache.org/foundation/governance/> (also at [Internet Archive 18 Feb. 2020 12:12:40](https://www.archive.org/details/apache-software-foundation-corporate-governance-overview-20200218)).
- Beanshell 2020. The Beanshell scripting language (website). NON-ARCHIVAL <https://beanshell.github.io/> (also at [Internet Archive 8 March 2020 11:21:39](https://www.archive.org/details/beanshell-scripting-language-20200308)).
- Joshua Bloch. 2017. *Effective Java* (3 ed.). Addison-Wesley, Boston, MA, USA.
- Andrey Breslav. 2018. Shoulders of Giants. Geekout 2018 conference presentation. Archived at <https://web.archive.org/web/20200227103321/https://www.slideshare.net/abreslav/shoulders-of-giants-languages-kotlin-learned-from>. Video at <https://2018.geekout.ee/andrey-breslav/>. Slides at <https://www.slideshare.net/abreslav/shoulders-of-giants-languages-kotlin-learned-from>.
- A presentation about how Kotlin design was inspired by other languages and encouraging language designers to collaborate rather than fight.
- Thanks a lot to the creators of Groovy, it's been a pleasure borrowing features from you.*
- BSF 2020. Apache Commons Bean Scripting Framework (website). NON-ARCHIVAL <https://commons.apache.org/proper/commons-bsf/> (also at [Internet Archive 18 Feb. 2020 06:52:52](https://www.archive.org/details/apache-commons-bean-scripting-framework-20200218)).
- Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks (Amsterdam, Netherlands: 1999)* 41, 1 (15 Jan.), 115–141.
- Codehaus. 2015. The Codehaus governance manifesto. Archived at <https://web.archive.org/web/20151201153729/http://www.codehaus.org/history/manifesto/>

- CommonsMath 2020a. Apache Commons Math: Java mathematics library (website). NON-ARCHIVAL <http://commons.apache.org/proper/commons-math/> (also at Internet Archive 18 Feb. 2020 11:03:36).
- CommonsMath 2020b. Real Matrix class usage (Apache Commons Math documentation). NON-ARCHIVAL http://commons.apache.org/proper/commons-math/userguide/linear.html#a3.2_Real_matrices (retrieved 16 Feb. 2020; also at Internet Archive 21 Aug. 2019 14:04:05).
- Conservancy. 2020. Software Freedom Conservancy (website). NON-ARCHIVAL <https://sfconservancy.org/> (also at Internet Archive 5 March 2020 20:05:19).
- Devcon3 2007. Groovy Developer Conference 3, 29th and 30th Jan 2007, Paris, France (meeting minutes). Archived at <https://web.archive.org/web/20150521060610/https://docs.codehaus.org/display/GroovyJSR/GDC+3+report>
- Devcon4 2007. Groovy Developer Conference 4, 15th and 16th Oct 2007, London, UK (meeting minutes). Archived at <https://web.archive.org/web/20150521060611/https://docs.codehaus.org/display/GroovyJSR/GDC4+Discussions>
- Devcon5 2009. Groovy Developer Conference 5, 11-15th May 2009, Paris, France (meeting minutes). Archived at <https://web.archive.org/web/20150520235115/https://docs.codehaus.org/display/GroovyJSR/Groovy+DevCon+5>
- Devcon6 2011. Groovy Developer Conference 6, 16th and 20th May 2011, Copenhagen, Denmark (meeting minutes). Archived at <https://web.archive.org/web/20150521060612/http://docs.codehaus.org/display/GroovyJSR/Groovy+DevCon+6>
- Devcon7 2012. Groovy Developer Conference 7, 6th and 7th Jun 2012, Copenhagen, Denmark (meeting minutes). Archived at <https://web.archive.org/web/20150521060612/https://docs.codehaus.org/display/GroovyJSR/Groovy+DevCon+7>
- Devcon8 2013. Groovy Developer Conference 8, 21st and 22nd May 2013, Paris, France (meeting minutes). NON-ARCHIVAL <https://docs.google.com/document/d/1mGxflR8gNm5v-HTGztzIDBmDiR70yZYdHZnCA1yiE8/edit?usp=sharing> Archived at <https://web.archive.org/web/20200301020012/https://docs.google.com/document/d/1mGxflR8gNm5v-HTGztzIDBmDiR70yZYdHZnCA1yiE8/edit>
- Devcon9 2013. Groovy Developer Conference 9, 10th and 11th Dec 2013, (meeting minutes). NON-ARCHIVAL https://docs.google.com/document/u/1/d/16tNNNvW4lhb-6Cw_JD4llcukqccAPui_wC-nf4UAXA/pub (also at Internet Archive 29 Feb. 2020 14:22:11).
- Devcon10 2014. Groovy Developer Conference 10, 5th Jun 2014, Copenhagen, Denmark. Meeting minutes. NON-ARCHIVAL <https://docs.google.com/document/d/18axejowbqSZuWF-06bEzx63FqhygmNHUuKx65i14EWU/edit?pli=1#heading=h.pd7xybm87okp> Archived at <https://web.archive.org/web/20200229143311/https://docs.google.com/document/d/18axejowbqSZuWF-06bEzx63FqhygmNHUuKx65i14EWU/edit?pli=1#heading=h.pd7xybm87okp>
- Eclipse. 2020. Eclipse Foundation (website). NON-ARCHIVAL <https://eclipse.org> Archived at <https://web.archive.org/web/20200307215618/https://www.eclipse.org/>
- Frege. 2013. Frege: Haskell-like language for the JVM (website). NON-ARCHIVAL <https://github.com/Frege/frege> (also at Internet Archive 7 Jan. 2020 09:31:58).
- Richard P. Gabriel. 1994. Lisp: Good News Bad News How to Win Big. *AI Expert* 6, 31–39.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- GEP-6 2010. GEP: NIO.2 Support for Groovy. Apache Groovy website. NON-ARCHIVAL <https://groovy.apache.org/wiki/GEP-6.html> (also at Internet Archive 13 March 2020 13:20:14).
- GEP-8 2011. GEP: Static type checking. Apache Groovy website. NON-ARCHIVAL <https://groovy.apache.org/wiki/GEP-8.html> (also at Internet Archive 10 March 2020 11:26:13).
- GEP-10 2011. GEP: Static compilation. Apache Groovy website. NON-ARCHIVAL <https://groovy.apache.org/wiki/GEP-10.html> (also at Internet Archive 13 March 2020 12:30:59).
- GEP-12 2013. GEP: SAM coercion. Apache Groovy website. NON-ARCHIVAL <https://groovy.apache.org/wiki/GEP-12.html> (also at Internet Archive 13 March 2020 13:39:28).
- GEPS. 2020. List of Groovy Enhancement Proposals (website). NON-ARCHIVAL <http://groovy.apache.org/wiki/geps.html> (also at Internet Archive 12 March 2020 14:57:50).
- Jeremy Gibbons. 2010. Design Patterns as Higher-Order Datatype-Generic Programs. 36. NON-ARCHIVAL <http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/hodgp-journal.pdf> Archived at <https://web.archive.org/web/20170813032800/http://www.cs.ox.ac.uk/jeremy.gibbons/publications/hodgp-journal.pdf> Submitted for publication; revised version of the WGP2006 paper.
- Goodtimes 2017. Goodtimes: java.time extensions library for Groovy (website). NON-ARCHIVAL <https://github.com/bdkosher/goodtimes> (also at Internet Archive 13 June 2018 00:40:49).
- Gradle 2020. Gradle build tool (website). NON-ARCHIVAL <https://gradle.org/> (also at Internet Archive 7 March 2020 09:59:16).
- Grengine 2020. Grengine embedded Groovy engine (website). NON-ARCHIVAL <http://grengine.ch> Archived at <https://web.archive.org/web/20190111170918/https://grengine.ch/>
- Griffon 2020. Griffon: desktop application development platform for the JVM (website). NON-ARCHIVAL <http://griffon-framework.org/> (also at Internet Archive 27 Aug. 2019 19:39:46).

- Groovy 2020. Groovy documentation. Apache Groovy project website. NON-ARCHIVAL <https://groovy-lang.org/documentation.html> (also at Internet Archive 14 Feb. 2020 01:59:18).
- GROOVY-1884 2008. GEP: Adding Bound Properties to GroovyBeans. Apache Groovy issue tracker. NON-ARCHIVAL <https://issues.apache.org/jira/browse/GROOVY-1884> (also at Internet Archive 4 March 2020 15:16:33).
- GROOVY-3425 2009. New feature: Power Assertions. Apache Groovy issue tracker. NON-ARCHIVAL <https://issues.apache.org/jira/browse/GROOVY-3425> (also at Internet Archive 28 Feb. 2020 06:09:48).
- GROOVY-8329 2009. New feature: Consider statically typed/compiled as default for Groovy 3.0. Apache Groovy issue tracker. NON-ARCHIVAL <https://issues.apache.org/jira/browse/GROOVY-8329> (also at Internet Archive 4 March 2020 15:20:08).
- InfoQ. 2015. The Demise of Open Source Hosting Providers Codehaus and Google Code. NON-ARCHIVAL <https://www.infoq.com/news/2015/03/codehaus-google-code> (also at Internet Archive 6 April 2017 09:25:54).
- Jenkins 2020a. Grails web application framework (website). NON-ARCHIVAL <https://grails.org/> (also at Internet Archive 27 Feb. 2020 19:20:13).
- Jenkins 2020b. Jenkins build automation server (website). NON-ARCHIVAL <https://jenkins.io/> (also at Internet Archive 8 March 2020 20:58:10).
- JEP 118 2015. JEP 118: Access to Parameter Names at Runtime. NON-ARCHIVAL <http://openjdk.java.net/jeps/118> (also at Internet Archive 25 July 2019 12:03:12).
- JEP 120 2015. JEP 120: Repeating Annotations. NON-ARCHIVAL <http://openjdk.java.net/jeps/120> (also at Internet Archive 14 Aug. 2019 03:50:59).
- JSR 223 2006. JSR 223: Scripting for the Java™Platform. NON-ARCHIVAL <https://www.jcp.org/en/jsr/detail?id=223> Archived at <https://web.archive.org/web/20200103154145/https://www.jcp.org/en/jsr/detail?id=223>
- JSR 241 2004. JSR 241: The Groovy Programming Language. NON-ARCHIVAL <https://www.jcp.org/en/jsr/detail?id=241> (also at Internet Archive 5 Dec. 2019 17:11:17).
- JSR 308 2014. JSR 308: Annotations on Java Types. NON-ARCHIVAL <https://jcp.org/en/jsr/detail?id=308> (also at Internet Archive 9 Aug. 2019 08:15:51).
- JSR 310 2014. JSR 310: Date and Time API. NON-ARCHIVAL <https://www.jcp.org/en/jsr/detail?id=310> (also at Internet Archive 19 Oct. 2019 04:25:09).
- Michael Kimberlin. 2010. Reducing Boilerplate Code with Project Lombok. NON-ARCHIVAL <https://objectcomputing.com/resources/publications/sett/january-2010-reducing-boilerplate-code-with-project-lombok> (also at Internet Archive 27 Feb. 2020 12:16:16).
- Dierk König, Paul King, Guillaume Laforge, Hamlet D'Arcy, Cédric Champeau, Erik Pragt, and Jon Skeet. 2015. *Groovy in Action* (2nd ed.). Manning Publications Co., Greenwich, CT, USA.
- Guillaume Laforge. 2007a. Groovy 1.0 is there! Groovy org.codehaus.groovy.jsr mailing list. 2 Jan. 2007. NON-ARCHIVAL <http://markmail.org/message/qspd5ufq35v7yk3a> (also at Internet Archive 11 March 2020 22:38:24).
- Guillaume Laforge. 2007b. Re: RFC: New operator for default assignment. Groovy org.codehaus.groovy.user mailing list. 26 June 2007. NON-ARCHIVAL <https://markmail.org/message/o4cmsyq5g3w4clcf> (also at Internet Archive 27 Feb. 2020 07:30:55).
- I'm sure we'll call it the Elvis operator!*
- Guillaume Laforge. 2007c. Re: Stupid Groovy 2 suggestion? Groovy org.codehaus.groovy.user mailing list. 22 Jan. 2007. NON-ARCHIVAL <https://markmail.org/message/4jx2qxdqfs3ue4ep> (also at Internet Archive 27 Feb. 2020 02:41:09).
- I wouldn't want Groovy to become too perlish with many cryptic weird operators.*
- Guillaume Laforge. 2008. A Domain-Specific Language for unit manipulations. NON-ARCHIVAL <https://dzone.com/articles/domain-specific-language-unit-> (also at Internet Archive 18 July 2019 01:39:38).
- Guillaume Laforge. 2017. How Languages Influence Each Other: Reflections on 14 Years of Apache Groovy. JavaOne 2017 conference presentation. Oct. 2017. Blog posting at: NON-ARCHIVAL <https://glaforge.appspot.com/article/javaone-how-languages-influence-each-other-reflections-on-14-years-of-apache-groovy>. Archived at https://web.archive.org/web/20200227114031/https://speakerdeck.s3.amazonaws.com/presentations/2159d9f06d0b4b388a849823f43e82fc/Apache_Groovy_language_influences_-_Guillaume_Laforge_-_JavaOne_2017.pdf (as PDF). Slides at <https://speakerdeck.com/glaforge/how-languages-influence-each-other-reflections-on-14-years-of-apache-groovy>.
- Dustin Marx. 2015. Codehaus: The Once Great House of Code Has Fallen. NON-ARCHIVAL <http://marxsoftware.blogspot.com/2015/03/codehaus-has-fallen.html> (also at Internet Archive 21 July 2019 05:37:04).
- Bob McWhirter. 2003. Re: My thoughts. Groovy org.codehaus.groovy.dev mailing list. 29 Aug. 2003. NON-ARCHIVAL <http://markmail.org/message/ymbllbcomgutzwf3e> (also at Internet Archive 1 June 2015 03:15:54).
- Right now, pretty much Strachan n'you guys are the idea rats working to design the language to be the most useful that it can, while I have fun being the mindless implementation monkey. :)*
- Bertrand Meyer. 1997. *Object-oriented Software Construction* (2nd ed.). Prentice Hall International (May).

- Micronaut 2020. Micronaut microservice framework (website). NON-ARCHIVAL <https://micronaut.io/> (also at Internet Archive 13 Jan. 2020 17:49:01).
- Microsoft. 2020. Type Providers. F# documentation. NON-ARCHIVAL <https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers/> (also at Internet Archive 22 Oct. 2019 20:18:30).
- Jesper Steen Møller. 2015. Antlr4 Groovy parser. Groovy org.apache.groovy.dev mailing list. 5 June 2015. NON-ARCHIVAL <https://groovy.markmail.org/thread/vsba74vytnv7lpsi> (also at Internet Archive 10 March 2020 07:44:36).
- NatureServe. 2020. NatureServe explorer. An online encyclopedia of life. NON-ARCHIVAL <http://explorer.natureserve.org/> (also at Internet Archive 14 Feb. 2020 12:29:17).
- Nice 2020. Nice programming language (website). NON-ARCHIVAL <http://nice.sourceforge.net/> (also at Internet Archive 17 Feb. 2020 22:28:34).
- Peter Niederwieser. 2009. Power assertions have landed in trunk! Groovy org.codehaus.groovy.user mailing list. 25 May 2009. NON-ARCHIVAL <http://markmail.org/message/7ykgv3zwedqmqjvdh> (also at Internet Archive 28 Feb. 2020 06:18:50).
- Peter Niederwieser et al. 2020. Spock Framework (website). NON-ARCHIVAL <http://spockframework.org/> (also at Internet Archive 13 Feb. 2020 06:49:55).
- Peter Norvig. 1998. Design Patterns in Dynamic Languages. NON-ARCHIVAL <http://norvig.com/design-patterns/> (also at Internet Archive 1 Feb. 2020 10:33:40).
- Martin Odersky. 2006. Pimp my Library (weblog). NON-ARCHIVAL <https://www.artima.com/weblogs/viewpost.jsp?thread=179766> (also at Internet Archive 29 June 2019 16:56:39).
- OpenJDK. 2012. Will JDK 8 support SAM that is Abstract Class? Lambda Development mailing list. NON-ARCHIVAL <http://openjdk.5641.n7.nabble.com/Will-JDK-8-support-SAM-that-is-Abstract-Class-tid103357.html> (also at Internet Archive 27 Feb. 2020 12:10:35).
- Christian Prehofer. 1997. From Inheritance to Feature Interaction or Composing Monads. In *Informatik '97 Informatik als Innovationsmotor*, Matthias Jarke, Klaus Pasedach, and Klaus Pohl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 562–571.
- Jeremy Rayner. 2007. Groovy 1.0 released! (blog post). NON-ARCHIVAL <http://javanicus.com/blog2/items/204-index.html> (also at Internet Archive 1 Aug. 2016 14:22:55).
- RedMonk. 2020. RedMonk Index for January 2020 (website). NON-ARCHIVAL <https://redmonk.com/sograzy/2020/02/28/language-rankings-1-20/> (also at Internet Archive 29 Feb. 2020 18:27:48).
- Graeme Rocher. 2006. Making DGM extensible / Extensible MetaClass. Groovy org.codehaus.groovy.dev mailing list. 26 Sept. 2006. NON-ARCHIVAL <https://markmail.org/message/hqnkatalanqerxot> (also at Internet Archive 28 Feb. 2020 02:24:33).
- Graeme Rocher. 2007. Grails is a breath of fresh air for Java developers. IndicThreads interview. NON-ARCHIVAL <http://www.indicthreads.com/1433/grails-is-a-breath-of-fresh-air-for-java-developers/> (also at Internet Archive 18 Dec. 2017 02:33:04).
- Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27.
- SoapUI 2020. SoapUI: test automation framework for SOAP, REST and more (website). NON-ARCHIVAL <https://www.soapui.org/> (also at Internet Archive 29 Feb. 2020 07:40:05).
- Spring 2020. Spring Integration: Dynamic language support. Spring Framework documentation. NON-ARCHIVAL <https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/dynamic-language.html> (also at Internet Archive 16 April 2019 14:53:27).
- Guy L. Steele, Jr. 1998. Growing a Language. In *Addendum to the 1998 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Addendum)* (Vancouver, British Columbia, Canada) (OOPSLA '98 Addendum). ACM, New York, NY, USA, 0.01–A1. <https://doi.org/10.1145/346852.346922>
- Andre Steingress. 2020. GContracts: design by contract library for Groovy (website). NON-ARCHIVAL <https://github.com/andresteingress/gcontracts> (also at Internet Archive 10 June 2018 23:29:08). <https://web.archive.org/web/20160308052827/http://gcontracts.org/>
- James Strachan. 2003a. current closest languages are.... Groovy org.codehaus.groovy.dev mailing list. 1 Sept. 2003. NON-ARCHIVAL <https://markmail.org/message/hne7dyg3j6sfrfp> (also at Internet Archive 8 March 2020 06:28:35).
- James Strachan. 2003b. Groovy - the birth of a new dynamic language for the Java platform. Archived at Internet Archive: <https://web.archive.org/web/20031205215854/http://radio.weblogs.com/80/0112098/2003/08/29.html>
- James Strachan. 2003c. I've added some example Groovy code... Groovy org.codehaus.groovy.dev mailing list. 29 Aug. 2003. NON-ARCHIVAL <https://markmail.org/message/kjr7awwpoxo7umqf> (also at Internet Archive 3 June 2015 13:43:55).
I thought we might as well be TDD with Groovy—writing tests for Groovy itself in Groovy to get started.
- James Strachan. 2003d. Re: class syntax. Groovy org.codehaus.groovy.dev mailing list. 12 Sept. 2003. NON-ARCHIVAL <https://markmail.org/message/zita5yauvqgsirgp> (also at Internet Archive 27 Feb. 2020 14:50:55).
- James Strachan. 2003e. thought for the day - use of implicit 'it' variable in closures. Groovy org.codehaus.groovy.dev mailing list. 28 Oct. 2003. NON-ARCHIVAL <http://markmail.org/message/annmqimy5daxylqkc> (also at Internet Archive 27

Feb. 2020 14:28:05).

Sun 1997. JavaBeans™ API specification. NON-ARCHIVAL <https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/> Archived at <https://web.archive.org/web/20181018105850/http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec>

TIOBE. 2019. TIOBE Index for October 2019 (website). <https://web.archive.org/web/20191025230404/https://www.tiobe.com/tiobe-index/> (retrieved 25 Oct. 2019) Archived at Internet Archive. Groovy index history: NON-ARCHIVAL <https://www.tiobe.com/tiobe-index/groovy/>

Alex Tkachman. 2011. Groovypp prototyping static compilation extension for Groovy (website). NON-ARCHIVAL <https://code.google.com/archive/p/groovyppptest/> (also at Internet Archive 6 March 2020 11:11:37).

uehaj. 2010. Japanese Groovy DSL (blog post). NON-ARCHIVAL <http://uehaj.hatenablog.com/entry/20100919/1284906117> (also at Internet Archive 14 May 2015 00:11:57).

Wikipedia. 2018a. Flow sensitive typing. NON-ARCHIVAL https://en.wikipedia.org/wiki/Flow-sensitive_typing (also at Internet Archive 18 March 2019 17:14:45).

Wikipedia 2018b. Safe navigation operator. NON-ARCHIVAL https://en.wikipedia.org/wiki/Safe_navigation_operator (also at Internet Archive 20 Feb. 2020 23:27:57).

Wikipedia. 2020. Feature interaction problem. NON-ARCHIVAL https://en.wikipedia.org/wiki/Feature_interaction_problem (also at Internet Archive 11 Oct. 2016 08:18:08).

Russel Winder. 2011. Static compilation for Groovy! Groovy org.codehaus.groovy.user mailing list. 11 Nov. 2011. NON-ARCHIVAL <https://markmail.org/message/g627wfm5au3emher> (also at Internet Archive 10 March 2020 10:54:05).

I don't want Groovy to be a failed Java 8 wannabee. I want a dynamic language to use as a complement to Java.

NON-ARCHIVAL REFERENCES

Andrey Breslav. 2020. Private Communication. Feb. 2020.

The author discussed Groovy's influence on Kotlin with Andrey. Reviewed and confirmed.

Hans Dockter. 2019. Private Communication.

The author discussed Groovy's impact on Gradle with Hans. Reviewed and confirmed.

Graeme Rocher. 2019. Private Communication.

The author discussed Groovy's impact on Grails and Micronaut with Graeme. Reviewed and confirmed.