Why use Groovy in 2024?

Apache Groovy is a multi-faceted programming language for the JVM. It provides a Java-like programming experience but adds several innovations, some of which we'll explore here[1].

Scripting

Groovy has *excellent support for scripting*. Groovy makes it easy to write simple scripts for automation tasks, testing, business rules, or your own domain specific language (DSL). You can easily use Java libraries by adding them to your classpath or build dependencies, or *grab* them within the script itself, e.g. let's use the `capitalize` method from Apache Commons Lang:

```
@Grab('org.apache.commons:commons-lang3:3.14.0')
import org.apache.commons.lang3.StringUtils

assert StringUtils.capitalize('groovy') == 'Groovy'
```

Extension Methods

Extension methods conceptually add methods to existing classes. An existing class doesn't actually change, the Groovy compiler or runtime just remembers the extensions. This reduces cognitive overhead (String-related methods belong to the String class, not sometimes in a utility class) and can reduce the need for external libraries. As an example, Groovy adds a capitalize method to Java's String class (we can remove the `@Grab`, imports and Commons Lang dependency of the previous example):

```
assert 'groovy'.class.name == 'java.lang.String'
assert 'groovy'.capitalize() == 'Groovy'
```

Groovy includes around 2000 such extensions across 150 Java classes, providing a *feature-rich out-of-the-box experience*.

Data scientists will find many useful extension methods including numerous ones for working with primitive arrays. Let's compare finding the maximum absolute value from a primitive integer array using Groovy's extension methods compared to the Groovy and Java streams equivalents:

```
Benchmark                                      Score  Units
ArrayStreamsBenchmark.groovyArrayMaxAbs        2.302  ns/op
ArrayStreamsBenchmark.groovyStreamsMaxAbs     43.100  ns/op
ArrayStreamsBenchmark.javaStreamsMaxAbs       42.346  ns/op
```

As this microbenchmark[2] shows, the array variant performs almost 20 times faster than the Groovy and Java stream variants.

Flexible Typing

Groovy has both dynamic and static natures. It supports dynamic typing like Python, Ruby or Smalltalk or *stricter-than-Java* typing through its type-checking extensions. As just one example of a type checking extension, we can find the errant '%D' (meant to be '%d') in the following example (which Java would find at runtime):

```
assert String.format('%d %02X %D', 15>>1, 15, (int)(Math.PI**2)) == '7 0F 9'
1 compilation error:
[Static type checking] - UnknownFormatConversion: Conversion = 'D'
```

Groovy supports duck typing to *reduce verbosity and allow terser, easier to read and change code*. It combines type inference with flow-sensitive typing to support static type checking of such code.

Improved OO and Functional Programming Support

Traits provide more *powerful code reuse* options in object-oriented designs. Groovy supports stateful traits, stackable traits, sharable behavior not just default behavior, and runtime traits like here:

```
trait Starable {
  String starify() { this.replaceAll('o', '⭐') }
}

var groovy = 'Groovy' as Starable
assert groovy.starify() == 'Gr⭐⭐vy'
```

Groovy Closures support memoization (caching), tail call elimination, and partial application. These can make functional algorithms *more powerful*, feasible and for some algorithms can *increase performance* by several orders of magnitude.

AST Transforms

AST Transforms provide a controlled way to alter the program being compiled. They are often used to inject boilerplate code and best-practice design patterns into your classes and records, meaning that you *write less code*. Consider writing an immutable Book class with built-in comparators for two of its properties and additional methods to provide externalizability and indexed property JavaBean support.
It takes less than 10 lines of Groovy and is the equivalent of about 600 lines of Java code.

```
@Immutable(copyWith = true)
@Sortable(excludes = 'authors')
@AutoExternalize
class Book {
  @IndexedProperty List<String> authors
  String title
  Date publicationDate
}
```

As another example, let's write a Shape record with deep immutability and a cached description:

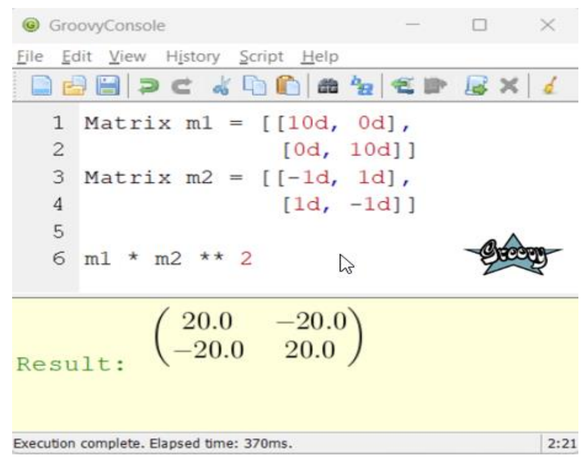
```
@PropertyOptions(propertyHandler = ImmutablePropertyHandler)
record Shape(List<Tuple2<Integer, Integer>> points, String color) {
  @Memoized
  String description() {
    "${color.toUpperCase()} shape with points: ${points.join(', ')}"
  }
}
```

Unrivalled Extensibility

Groovy lets you *write your own* extension methods, your own type checking extensions, and your own AST transforms. It even lets you extend the display functionality of some tooling.

Operator Overloading

Groovy makes processing logic *easier to read and write* by using familiar numerical operators with other classes, e.g. here we use Apache Commons Math matrices (we could similarly have used Java's `BigInteger` and `BigDecimal` classes, imaginary numbers, fractions, or even shopping carts):



As an added bonus, Groovy's console allows us to customize the output; here generating a pretty LaTeX generated visualization of the matrix.

Java Features Earlier

Groovy allows some Java features in earlier JDK versions:

Feature	JDK version with Java	JDK version with Groovy
Sealed types	17	8 emulated, 17 native
Records	16	8 emulated/similar, 16 native
Text blocks/templating	14/21 preview	8 equivalent
Scripting *JSR 223* *JEP 445*	- - 22 preview	8 8 22 preview (Groovy 5)
Sequenced collections	21	8 equivalent
Gatherers	22 preview	8 related non-stream functionality
Switch improvements	14/21	8 similar functionality

Apache Groovy has over 500 contributors, over 240 releases and over 3B downloads. We are always keen to chat to folks using Groovy and warmly welcome feedback and contributions. Why not visit us at https://groovy.apache.org/.

[1] Example code: https://github.com/paulk-asert/groovy-today
[2] Microbenchmark: https://github.com/paulk-asert/stream-performance