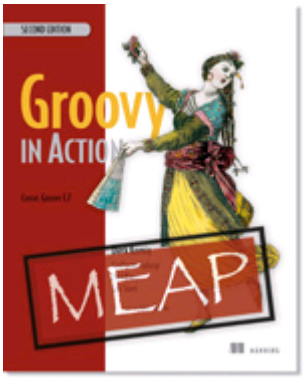
MANNING PUBLICATIONS

What is Groovy?

Green Paper from



Groovy in Action, Second Edition

EARLY ACCESS EDITION

Dierk König, Paul King, Guillaume Laforge, Jon Skeet

MEAP Release: June 2009
Softbound print: Early 2011 | 700 pages
ISBN: 9781935182443

This green paper is taken from the book Groovy in Action, Second Edition from Manning Publications. The authors explain the benefits of Groovy, specifically its seamless integration with Java, richness of features, and areas of applicability. For the table of contents, the author forum, and other resources, go to http://www.manning.com/koenig2/.

Groovy is an optionally typed, dynamic language for the Java platform, with many features that are inspired by languages like Python, Ruby, and Smalltalk, which makes those features available to Java developers through Java-like syntax. Unlike other alternative languages, it is designed as a companion, not a replacement for Java.

Groovy is often referred to as a scripting language—and it works very well for scripting. It's a mistake to label Groovy purely in those terms, though. It can be precompiled into Java bytecode and integrated into Java applications and power web applications. It can add an extra degree of control within build files and form the basis of whole applications on its own—Groovy is too flexible to be pigeon-holed.

What we can say about Groovy is that it is closely tied to the Java platform. This is true in terms of both implementation (many parts of Groovy are written in Java, with the rest being written in Groovy itself) and interaction. When you program in Groovy, in many ways you're writing a special kind of Java. All the power of the Java platform—including the massive set of available libraries—is there to be harnessed.

Does this make Groovy just a layer of syntactic sugar? Not at all. Although everything you do in Groovy could be done in Java, it would be madness to write the Java code required to work Groovy's magic. Groovy performs a lot of work behind the scenes to achieve its agility and dynamic nature. Many of the Groovy features that seem extraordinary at first—encapsulating logic in objects in a natural way, building hierarchies with barely any code other than what is absolutely required to compute the data, expressing database queries in the normal application language before they are translated into SQL, manipulating the runtime behavior of individual objects after they have been created—are tasks that Java wasn't designed for.

Let's take a closer look at what makes Groovy so appealing, starting with how Groovy and Java work hand-in-hand.

Playing nicely with Java: seamless integration

Being Java friendly means two things: seamless integration with the Java Runtime Environment and having a syntax that is aligned with Java.

Seamless integration

Figure 1 shows the integration aspect of Groovy: it runs inside the Java Virtual Machine and makes use of Java's libraries (together called the Java Runtime Environment or JRE). Groovy is only a new way of creating *ordinary* Java classes—from a runtime perspective, Groovy is Java with an additional jar file as a dependency.

Figure 1 Groovy and Java join together in a tongue-and-groove fashion

Consequently, calling Java from Groovy is a nonissue. When developing in Groovy, you end up doing this all the time without noticing. Every Groovy type is a subtype of `java.lang.Object`. Every Groovy object is an instance of a type in the normal way. A Groovy date is a `java.util.Date`. You can call on it all methods that you know are available for a Date and you can pass it as an argument to any method that expects a Date.

Calling into Java is an easy exercise. It is something that all JVM languages offer—at least the ones worth speaking of. They all make it possible, some by staying inside their own non-Java abstractions, some by providing a gateway. Groovy is one of the few that does it its own way and the Java way at the same time, since there is no difference.

Integration in the opposite direction is just as easy. Suppose a Groovy class `MyGroovyClass` is compiled into `MyGroovyClass.class` and put on the classpath. You can use this Groovy class from within a Java class by typing

```
new MyGroovyClass(); // create from Java
```

You can then call methods on the instance, pass the reference as an argument to methods, and so forth. The JVM is blissfully unaware that the code was written in Groovy. This becomes particularly important when integrating with Java frameworks that call your class where you have no control over how that call is affected.

The "interoperability" in this direction is a bit more involved for alternative JVM languages. Yes, they may "compile to bytecode" but that does not mean much for itself, since one can produce valid bytecode that is totally incomprehensible for a Java caller. A language may not even be object oriented and provide classes and methods. And, even if it does, it may assign totally different semantics to those abstractions. Groovy, in contrast, stays fully inside the Java object model. Actually, compiling to class files is only one of many ways to integrate Groovy into your Java project. The integration ladder in Figure 2 arranges the integration criteria by their significance.

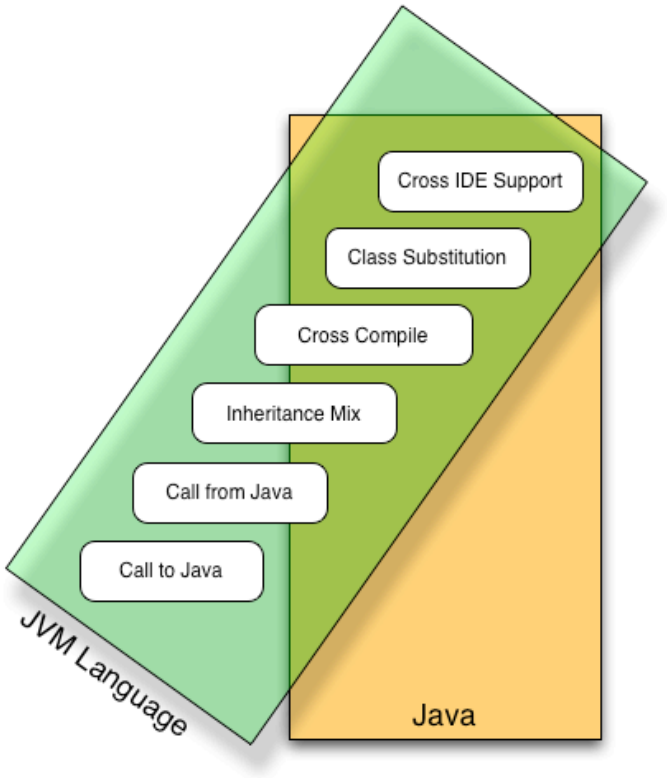


Figure 2 The integration ladder shows increasing cross-language support, from simple calls for interoperability up to seamless tool integration.

One step up on the integration ladder, and we meet the issue of references. A Groovy class may reference a Java class (that goes without saying) and a Java class may reference a Groovy class, as we have seen above. We can even have circular references, and `groovyc` compiles them all transparently. Even better, the leading IDEs provide cross-language compile, navigation, and refactoring such that you hardly ever need to care about the project build setup. You are free to choose Java or Groovy when implementing any class for that matter. Such tight build-time integration is a challenge for every other language.

Overloaded methods are the next rung where candidates slip off. Imagine you set out to implement the Java interface `java.io.Writer` in any non-Java language. It comes with three versions of "write" that take one parameter: `write(int c)`, `write(String str)`, and `write(char[] buf)`. Implementing this in Groovy is trivial; it's exactly like in Java. The formal parameter types distinguish which methods you override. That's one of many merits of optional typing. Languages that are solely dynamically typed have no way of doing this.

But the buck doesn't stop here. The Java/Groovy mix allows annotations and interfaces to be defined in either language and implemented and used in the other. You can subclass in any combination even with abstract classes and "sandwich" inheritance like Java-Groovy-Java or Groovy-Java-Groovy in arbitrary depth. It may look exotic at first sight, but we actually needed this feature in customer projects. Of course, this integration presupposes that your language knows about annotations and interfaces like Groovy does.

True seamless integration means that you can take any Java class from a given Java codebase and replace it with a Groovy class. Likewise, you can take any Groovy class and rewrite it in Java, both without touching any other class in the codebase. That's what we call a drop-in replacement, which imposes further consideration about annotations, static members, and accessibility of the used libraries from Java.

Finally, generated bytecode can be more or less Java tool-friendly. There are more and more tools on the market that directly augment your bytecode, be it for gathering test coverage information or "weaving aspects" in.

These tools expect not only bytecode to be valid but also to find well-known patterns in it such as the Java and Groovy compiler provide. Bytecode generated by other languages is often not digestible for such tools.

Alternative JVM languages are often attributed as working "seamlessly" with Java. With the integration ladder, you can check to what degree this applies: calls into Java, calls from Java, bidirectional compilation, inheritance intermix, mutual class substitutability, and tool support. We didn't even consider security, profiling, debugging, and other Java "architectures". So much for the platform integration; now onto the syntax.

Syntax alignment

The second dimension of Groovy's friendliness is its syntax alignment. Let's compare the different mechanisms to obtain today's date in various languages and to demonstrate what alignment should mean:

```
import java.util.*;       // Java
Date today = new Date();  // Java

today = new Date()        // Groovy

require 'date'             # Ruby
today = Date.new           # Ruby

import java.util._        // Scala
var today = new Date      // Scala

(import '(java.util Date)) ; Clojure
(def today (new Date))     ; Clojure
(def today (Date.))        ; Clojure alternative
```

The Groovy solution is short, precise, and more compact than regular Java. Groovy does not need to import the `java.util` package or specify the Date type. This is very handy when using Groovy to evaluate user input. In those cases, one cannot assume that the user is proficient in Java package structures or willing to write more code than necessary. Additionally, Groovy doesn't require semicolons when it can understand the code without them. Despite being more compact, Groovy is fully comprehensible to a Java programmer.

The Ruby solution is listed to illustrate what Groovy avoids: a different packaging concept (require), a different comment syntax, and a different object-creation syntax. Scala introduces a new wildcard syntax with underscores and has its own way of declaring whether a reference is supposed to be (in Java terms) "final" or not (var vs. val). The user has to provide one or the other. Clojure doesn't support wildcard imports as of now and shows two alternative ways of instantiating a Java class, both of which differ syntactically from Java.

Although all the alternative notations make sense in themselves and may even be more consistent than Java, they do not align as nicely with the Java syntax and architecture as Groovy does. Throw into the mix that Groovy is the only language besides Java that fully supports the Java notation of generics and annotations and you easily retrace why we position the Groovy syntax as being perfectly aligned with Java.

Now you have an idea what Java friendliness means in terms of integration and syntax alignment. But how about feature richness?

Power in your code: a feature-rich language

Giving a list of Groovy features is a bit like giving a list of moves a dancer can perform. Although each feature is important in itself, it's how well they work together that makes Groovy shine. Groovy has three main types of features over and above those of Java: language features, libraries specific to Groovy, and additions to the existing Java standard classes (GDK). Figure 3 shows some of these features and how they fit together. The shaded circles indicate the way that the features use each other. For instance, many of the library features rely heavily on language features. Idiomatic Groovy code rarely uses one feature in isolation—instead, it usually uses several of them together, like notes in a chord.

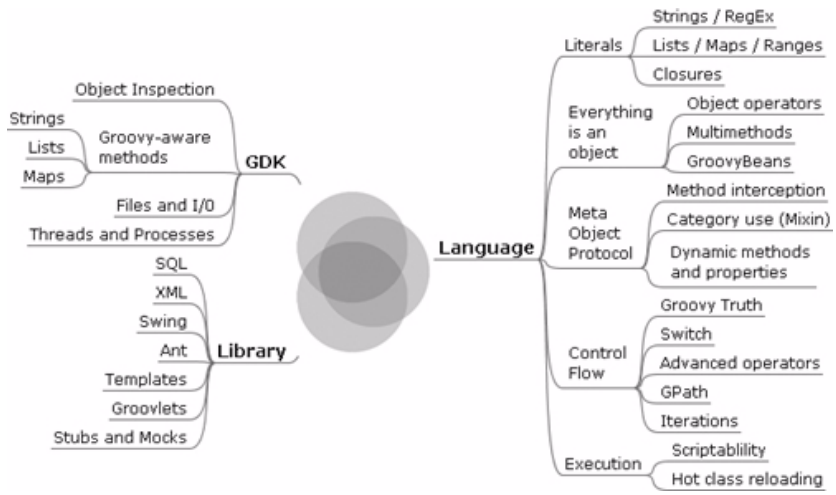


Figure 3 Many additional libraries and JDK enhancements in Groovy build on the new language features. The combination of the three forms a “sweet spot” for clear and powerful code.

Unfortunately, many of the features can't be understood in just a few words. Closures, for example, are an invaluable language concept in Groovy, but the word on its own doesn't tell you anything. We won't go into all the details now, but here are a few examples to whet your appetite.

Listing a file: closures and I/O additions

Closures are blocks of code that can be treated as first-class objects: passed around as references, stored, executed at arbitrary times, and so on. Java's anonymous inner classes are often used this way, particularly with adapter classes, but the syntax of inner classes is ugly, and they're limited in terms of the data they can access and change.

File handling in Groovy is made significantly easier with the addition of various methods to classes in the `java.io` package. A great example is the `File.eachLine` method. How often have you needed to read a file, a line at a time, and perform the same action on each line, closing the file at the end? This is such a common task, it shouldn't be difficult—so in Groovy, it isn't.

Let's put the two features together and create a complete program that lists a file with line numbers:

```
def number = 0
new File('data.txt').eachLine { line ->
    number++
    println "$number: $line"
}
```

which prints

```
1: first line
2: second line
```

The curly braces enclose the closure. It is passed as an argument to File's new `eachLine` method, which, in turn, calls back the closure for each line that it reads, passing the current line as an argument.

Printing a list: collection literals and simplified property access

`java.util.List` and `java.util.Map` are probably the most widely used interfaces in Java, but there is little language support for them. Groovy adds the ability to declare list and map literals just as easily as you would a string or numeric literal, and it adds many methods to the collection classes.

Similarly, the JavaBean conventions for properties are almost ubiquitous in Java, but the language makes no use of them. Groovy simplifies property access, allowing for far more readable code.

Here's an example using these two features to print the package for each of a list of classes. Note that the word *clazz* is not class because that would be a Groovy keyword—exactly like in Java. Although Java would allow a similar first line to declare an array, we're using a real list here—elements could be added or removed with no extra work:

```
def classes = [String, List, File]
for (clazz in classes) {
    println clazz.package.name
}
```

which prints

```
java.lang
java.util
java.io
```

In Groovy, you can even avoid such commonplace for loops by applying property access to a list—the result is a list of the properties. Using this feature, an equivalent solution to the previous code is

```
println( [String, List, File]*.package*.name )
```

to produce the output

```
[java.lang, java.util, java.io]
```

Pretty cool, eh? The star character is optional in the above code. We add it to emphasize that the access to package and name is spread over the list and thus applied to every item in it.

XML handling the Groovy way: GPath with dynamic properties

Whether you're reading it or writing it, working with XML in Java requires a considerable amount of work. Alternatives to the W3C DOM make life easier, but Java itself doesn't help you in language terms—it's unable to adapt to your needs. Groovy allows classes to act as if they had properties at runtime even if the names of those properties aren't known when the class is compiled. `GPath` was built on this feature, and it allows seamless XPath-like navigation of XML documents.

Suppose you have a file called `customers.xml` such as this:

```
<?xml version="1.0" ?>
<customers>
  <corporate>
    <customer name="Bill Gates"        company="Microsoft" />
    <customer name="Steve Jobs"        company="Apple" />
    <customer name="Jonathan Schwartz" company="Sun" />
  </corporate>
  <consumer>
    <customer name="John Doe" />
    <customer name="Jane Doe" />
  </consumer>
</customers>
```

You can print out all the corporate customers with their names and companies using just the following code.

```
def customers = new XmlSlurper().parse(new File('customers.xml'))
for (customer in customers.corporate.customer) {
    println "${customer.@name} works for ${customer.@company}"
}
```

which prints

```
Bill Gates works for Microsoft
Steve Jobs works for Apple
Jonathan Schwartz works for Sun
```

Note that Groovy cannot possibly know anything in advance about the elements and attributes that are available in the XML file. It happily compiles anyway. That's one capability that distinguishes a *dynamic* language.

Scripting the web

For closing up, we show a little trick that Scott Davis presented at JavaOne 2009: fetching a rhyme from a REST web service and evaluating the result as if it were Groovy code. This code will print all rhymes to movie. Expect your favorite programming language to be included!

```
def text = "http://azarask.in/services/rhyme/?q=movie".toURL().text
for (rhyme in evaluate(text)) println rhyme
```

The term *scripting* refers to the ability to take a string of program code and evaluate it at runtime. That string may be given as user input, read from a database, or fetched from the web like above. The text we fetch happens to be so simple[1] that we can treat it is as valid Groovy code that denotes a list of Strings. We don't need to write a parser. The Groovy parser does all the work.

Even trying to demonstrate just a few features of Groovy, you've seen other features in the preceding examples—string interpolation with `GString`, simpler for loops, optional typing, and optional statement terminators and parentheses, just for starters. The features work so well with each other and become second nature so quickly, you hardly notice you're using them.

Although being Java friendly and feature rich are the main driving forces for Groovy, there are more aspects worth considering. So far, we have focused on the hard technical facts about Groovy, but a language needs more than that to be successful. It needs to attract people. In the world of computer languages, building a better mousetrap doesn't guarantee that the world will beat a path to your door. It has to appeal to both developers and their managers, in different ways.

Community-driven but corporate-backed

For some people, it's comforting to know that their investment in a language is protected by its adoption as a standard. This is one of the distinctive promises of Groovy. Since the passage of JSR-241, Groovy is the second language under standardization for the Java platform (the first being the Java language).

The size of the user base is a second criterion. The larger the user base, the greater the chance of obtaining good support and sustainable development. Groovy's user base has grown beyond all expectations. Recent polls suggest that Groovy is used in the majority of all organizations that develop professionally with Java, much higher than any alternative language. Groovy is regularly covered in Java conferences and publications, and virtually any Java open-source project that allows scripting extensions supports Groovy. Groovy and Grails mailinglists are the busiest ones at codehaus. Groovy has become an important item in many developers' CVs and job descriptions.

Many corporations support Groovy in various ways. Sun Microsystems, Inc. integrates Groovy support in their NetBeans IDE tool suite, presents Groovy at JavaOne, and pushes forward the idea of multiple languages on the JVM like in the JSRs 241 (Groovy), 223 (Scripting Integration), and 292 (InvokeDynamic). Oracle Corporation has a long-standing tradition of using Groovy in a number of products just like other big players, including IBM and SAP. While the development of Groovy has always been driven by its community, it also profited from financial backing. Sustainability of Groovy development was first sponsored by Big Sky Technology, then by G2One, and recently taken over by SpringSource. Big thanks to all that made this development possible!

[1] It is actually JavaScript Object Notation (JSON) format.

Commercial support is also available if needed. Many companies offer training, consulting and engineering for Groovy, including the ones that we authors work for (alphabetically): ASERT, Canoo, and SpringSource.

Attraction is more than strategic considerations, however. Beyond what you can measure is a gut feeling that causes you to enjoy programming or *not*.

The developers of Groovy are aware of this feeling, and it is carefully considered when deciding upon language features. After all, there is a reason for the name of the language.

GROOVY

"A situation or an activity that one enjoys or to which one is especially well suited (found his groove playing bass in a trio). A very pleasurable experience; enjoy oneself (just sitting around, grooving on the music). To be affected with pleasurable excitement. To react or interact harmoniously." (http://dict.leo.org)

Someone recently stated that Groovy was, "Java-stylish with a Rubyesque feeling". We cannot think of a better description. Working with Groovy feels like a partnership between you and the language, rather than a battle to express what is clear in your mind in a way the computer can understand.

Of course, while it's nice to "feel the groove," you still need to pay your bills. In the next topic, we'll look at some of the practical advantages Groovy will bring to your professional life.

What Groovy can do for you

Depending on your background and experience, you are probably interested in different features of Groovy. It is unlikely that anyone will require every aspect of Groovy in their day-to-day work, just as no one uses the whole of the mammoth framework provided by the Java standard libraries.

This topic presents interesting Groovy features and areas of applicability for Java professionals, script programmers, and pragmatic, extreme, and agile programmers. We recognize that developers rarely have just one role within their jobs and may well have to take on each of these identities in turn. However, it is helpful to focus on how Groovy helps in the kinds of situations typically associated with each role.

Groovy for Java professionals

If you consider yourself a Java professional, you probably have years of experience in Java programming. You know all the important parts of the Java Runtime API and most likely the APIs of a lot of additional Java packages.

But—be honest—there are times when you cannot leverage this knowledge, such as when faced with an everyday task like recursively searching through all files below the current directory. If you're like us, programming such an ad-hoc task in Java is just too much effort.

With Groovy you can quickly open the console and type

```
groovy -e "new File('.').eachFileRecurse { println it }"
```

to print all filenames recursively.

Even if Java had an `eachFileRecurse` method and a matching `FileListener` interface, you would still need to explicitly create a class, declare a main method, save the code as a file, and compile it, and only then could you run it. For the sake of comparison, let's see what the Java code would look like, assuming the existence of an appropriate `eachFileRecurse` method:

```
import java.io.*;                          // JAVA !!
public class ListFiles {
    public static void main(String[] args) {
        new File(".").eachFileRecurse(     // imagine Java had this
            new FileListener() {
                public void onFile (File file) {
                    System.out.println(file.toString());
                }
            }
        );
    }
}
```

Notice how the intent of the code (printing each file) is obscured by the scaffolding code Java requires you to write in order to end up with a complete program.

Besides command-line availability and code beauty, Groovy allows you to bring dynamic behavior to Java applications, such as through expressing business rules that can be maintained while the application is running, allowing smart configurations, or even implementing domain specific languages.

You have the options of using static or dynamic types and working with precompiled code or plain Groovy source code with on-demand compiling. As a developer, you can decide where and when you want to put your solution "in stone" and where it needs to be flexible. With Groovy, you have the choice.

This should give you enough safeguards to feel comfortable incorporating Groovy into your projects so you can benefit from its features.

Groovy for script programmers

As a script programmer, you may have worked in Perl, Ruby, Python, or other dynamic (nonscripting) languages such as Smalltalk, Lisp, or Dylan.

But the Java platform has an undeniable market share, and it's fairly common that folks like you work with the Java language to make a living. Corporate clients often run a Java standard platform (e.g., J2EE), allowing nothing but Java to be developed and deployed in production. You have no chance of getting your ultraslick scripting solution in there, so you bite the bullet, roll up your sleeves, and dig through endless piles of Java code, thinking all day, "If I only had [your language here], I could replace this whole method with a single line!" We confess to having experienced this kind of frustration.

Groovy can give you relief and bring back the fun of programming by providing advanced language features where you need them: in your daily work. By allowing you to call methods on anything, pass blocks of code around for immediate or later execution, augment existing library code with your own specialized semantics, and use a host of other powerful features, Groovy lets you express yourself clearly and achieve miracles with little code.

Just sneak the groovy-all-*.jar file into your project's classpath, and you're there.

Today, software development is seldom a solitary activity, and your teammates (and your boss) need to know what you are doing with Groovy and what Groovy is about. This book aims to be a device you can pass along to others so they can learn, too. (Of course, if you can't bear the thought of parting with it, you can tell them to buy their own copies. We won't mind.)

Groovy for pragmatic programmers, extremos, and agilists

If you fall into this category, you probably already have an overloaded bookshelf, a board full of index cards with tasks, and an automated test suite that threatens to turn red at a moment's notice. The next iteration release is close, and there is anything but time to think about Groovy. Even uttering the word makes your pair-programming mate start questioning your state of mind.

One thing that we've learned about being pragmatic, extreme, or agile is that every now and then you have to step back, relax, and assess whether your tools are still sharp enough to cut smoothly. Despite the ever-pressing project schedules, you need to sharpen the saw regularly. In software terms, that means having the knowledge and resources needed and using the right methodology, tools, technologies, and languages for the task at hand.

Groovy will be your house elf for all automation tasks that you are likely to have in your projects. These range from simple build automation, continuous integration, and reporting, up to automated documentation, shipment, and installation. The Groovy automation support leverages the power of existing solutions such as Ant and Maven, while providing a simple and concise language means to control them. Groovy even helps with testing, both at the unit and functional levels, helping us test-driven folks feel right at home.

Hardly any school of programmers applies as much rigor and pays as much attention as we do when it comes to self-describing, intention-revealing code. We feel an almost physical need to remove duplication while striving for simpler solutions. This is where Groovy can help tremendously.

Before Groovy, I (Dierk) used other scripting languages (preferably Ruby) to sketch some design ideas, do a spike—a programming experiment to assess the feasibility of a task—and run a functional prototype. The downside was that I was never sure if what I was writing would also work in Java. Worse, in the end, I had the work of

porting it over or redoing it from scratch. With Groovy, I can do all the exploration work directly on my target platform.

EXAMPLE

Recently, Guillaume and I did a spike on prime number disassembly. We started with a small Groovy solution that did the job cleanly but not efficiently. Using Groovy's interception capabilities, we unit-tested the solution and counted the number of operations. Because the code was clean, it was a breeze to optimize the solution and decrease the operation count. It would have been much more difficult to recognize the optimization potential in Java code. The final result can be used from Java as it stands, and although we certainly still have the option of porting the optimized solution to plain Java, which would give us another performance gain, we can defer the decision until the need arises.

The seamless interplay of Groovy and Java opens two dimensions of optimizing code: using Java for code that needs to be optimized for runtime performance and using Groovy for code that needs to be optimized for flexibility and readability.

Along with all these tangible benefits, there is value in learning Groovy for its own sake. It will open your mind to new solutions, helping you to perceive new concepts when developing software, whichever language you use.

No matter what kind of programmer you are, we hope you are now eager to get some Groovy code under your fingers.

Summary

We hope that by now we've convinced you that you really want Groovy in your life. As a modern language built on the solid foundation of Java and with community support and corporate backing, Groovy has something to offer for everyone, in whatever way they interact with the Java platform.

With a clear idea of why Groovy was developed and what drives its design, you should be able to see where features fit into the bigger picture. Keep in mind the principles of Java integration and feature richness, making common tasks simpler and your code more expressive.

Once you have Groovy installed, you can run it both directly as a script and after compilation into classes. If you have been feeling energetic, you may even have installed a Groovy plug-in for your favorite IDE. With this preparatory work complete, you are ready to see (and try!) more of the language itself.