SAMPLE CHAPTER

Making Java Groovy

Kenneth A. Kousen

FOREWORD BY Guillaume Laforge

MANNING

Making Java Groovy

by Kenneth A. Kousen

Chapter 1

brief contents

1 Why add Groovy to Java?

This chapter covers

- Issues with Java
- Groovy features that help Java
- Common use cases for Java and how Groovy makes them simpler

For all of its flaws (and we'll be reviewing them shortly), Java is still the dominant object-oriented programming language in the industry today. It's everywhere, especially on the server side, where it's used to implement everything from web applications to messaging systems to the basic infrastructure of servers. It's therefore not surprising that there are more Java developers and more Java development jobs available than for any other programming language. As a language, Java is an unmitigated success story.

If Java is so ubiquitous and so helpful, why switch to anything else? Why not continue using Java everywhere a Java Virtual Machine (JVM) is available?

In this book, the answer to that question is, go right ahead. Where Java works for you and gets the job done, by all means continue to use it. I expect that you already have a Java background and don't want to lose all that hard-earned experience. Still, there are problems that Java solves easily, and problems that Java makes difficult. For those difficult issues, consider an alternative.

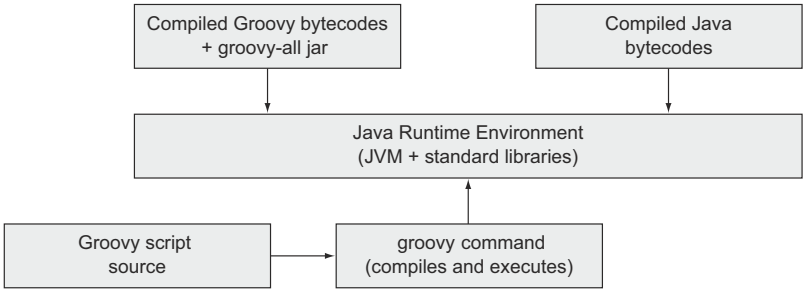

Figure 1.1 Groovy generates bytecodes for the Java Virtual Machine. Either compile them ahead of time or let the `groovy` command generate them from source.

That alternative is Groovy. In this chapter I'll review some of the issues with Java that lead to problems for developers and discuss how Groovy can help alleviate them. I'll also show a range of tools, provided as part of the Groovy ecosystem, that can make pure Java development easier. In the long run, I suggest a blended approach: let Java do what it does well, and let Groovy help where Java has difficulties.

Throughout, this will be the mantra:

GUIDING PRINCIPLE Java is great for tools, libraries, and infrastructure. Groovy is great for everything else.

Use Java where Java works well, and use Groovy where it makes your life easier. Nobody is ever going to rewrite, say, the Spring Framework, in Groovy. There's no need. Groovy works beautifully with Spring, as I'll discuss in detail in chapter 7. Likewise, the JVM is everywhere. That's a good thing, because wherever Java can run, so can Groovy, as shown in figure 1.1.

I'll discuss the practical details in the next chapter, but at its base Groovy *is* Java. Groovy scripts and classes compile to bytecodes that can be freely intermixed with compiled Java classes. From a runtime point of view, running compiled Groovy means just adding a single JAR file to your environment.

One of the goals of this book is to identify opportunities where Groovy can significantly help Java developers. To do that, let me first review where Java might have some issues that need help.

1.1 Issues with Java

A perfect storm swept through the development world in the mid- to late-1990s, which ultimately resulted in moving the primary development language from C++ to Java. Java is effectively the next-generation language in the C++ family. Its syntax shares much in common with C and C++. Language constructs that caused intermediate-level developers problems, like memory management and pointer arithmetic, were handled automatically or removed from programmer control altogether. The language was small (as hard as that might be to imagine now), easy to write, and, above

all, free. Just download a JDK, access the library docs (making available clean, up-to-date, hyperlinked library documentation was quite the innovation at the time), and start coding. The leading browser of the day, Netscape, even had a JVM built right into it. Combined with the whole Write Once, Run Anywhere mantra, Java carried the day.

A lot of time has passed since then. Java has grown considerably, and decisions made early in its development now complicate development rather than simplify it. What sorts of decisions were those? Here's a short, though hardly exhaustive, list:

- Java is statically typed.
- All methods in Java must be contained within a class.
- Java forbids operator overloading.
- The default access for attributes and methods is "package private."
- Java treats primitives differently from classes.

Over time Java also accumulated inconsistencies. For example, arrays have a `length` property, strings have a `length` method, collections have a `size` method, and node lists (in XML) have a `getLength` method. Groovy provides a `size` method for all of them.

Java also lacks metaprogramming capabilities.[1] That's not a flaw, but it limits Java's ability to create domain-specific languages (DSLs).

There are other issues as well, but this list will give us a good start. Let's look at a few of these items individually.

1.1.1 Is static typing a bug or a feature?

When Java was created, the thinking in the industry was that static typing—the fact that you must declare the type of every variable—was a benefit. The combination of static typing and dynamic binding meant that developers had enough structure to let the compiler catch problems right away, but still had enough freedom to implement and use polymorphism. Polymorphism lets developers override methods from superclasses and change their behavior in subclasses, making reuse by inheritance practical. Even better, Java is dynamically bound by default, so you can override anything you want unless the keyword `final` is applied to a method.

Static typing makes Integrated Development Environments useful too, because they can use the types to prompt developers for the correct fields and methods. IDEs like Eclipse and NetBeans, both powerful and free, became pervasive in the industry partly as a result of this convenience.

So what's wrong with static typing? If you want an earful ask any Smalltalk developer. More practically, under Java's dynamic binding restrictions (that you can't override anything unless two classes are related by inheritance), static typing is overly restrictive. Dynamically typed languages have much more freedom to let one object stand in for another.

---

[1] That's for a variety of good reasons, many of which relate to performance. Metaprogramming depends on dynamic capabilities like reflection, which was very slow when Java was first released. Groovy in 1998 on Java 1.2 would have been a daunting prospect at best.

As a simple example, consider arrays and strings. Both are data structures that collect information: arrays collect objects, and strings collect characters. Both have the concept of appending a new element to the existing structure. Say we have a class that includes an array and we want to test the class's methods. We're not interested in testing the behavior of arrays. We know they work. But our class has a dependency on the array.

What we need is some kind of mock object to represent the array during testing. If we have a language with dynamic typing, and all we are invoking is the `append` method on it using character arguments, we can supply a string wherever we have an array and everything will still work.

In Java one object can only stand in for another if the two classes are related by inheritance or if both implement the same interface. A static reference can only be assigned to an object of that type or one of its subclasses, or a class that implements that interface if the reference is of interface type. In a dynamically typed language, however, we can have any classes stand in for another, as long as they implement the methods we need. In the dynamic world this is known as *duck typing*: if it walks like a duck and it quacks like a duck, it's a duck. See figure 1.2.

We don't care that a string is not an array as long as it has the `append` method we need. This example also shows another feature of Groovy that was left out of Java: operator overloading. In Groovy all operators are represented by methods that can be overridden. For example, the + operator uses a `plus()` method and * uses `multiply()`. In the previous figure the << operator represents the `leftShift()` method, which is implemented as `append` for both arrays and strings.

GROOVY FEATURE Groovy features like optional typing and operator overloading give developers greater flexibility in far less code.

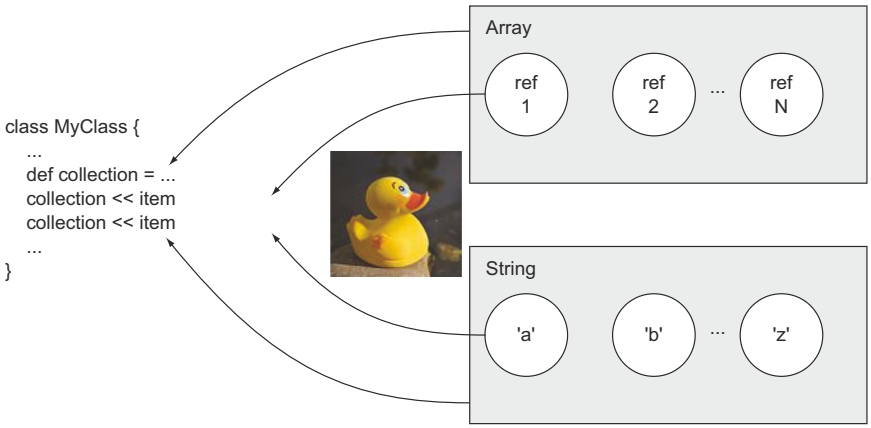

Figure 1.2 Arrays and strings from a duck-typing point of view. Each is a collection with an `append` method. If that's all we care about, they're the same.

Regarding optional typing, Groovy gives you the best of both worlds. If you know the type of a variable, feel free to specify it. If you don't know or you don't care, feel free to use the `def` keyword.

1.1.2 Methods must be in a class, even if you don't need or want one

Some time ago, Steve Yegge wrote a very influential blog post called "Execution in the Kingdom of the Nouns."[2] In it he described a world where nouns rule and verbs are second-class citizens. It's an entertaining post and I recommend reading it.

Java is firmly rooted in that world. In Java all methods (verbs) must reside inside classes (nouns). You can't have a method by itself. It has to be in a class somewhere. Most of the time that's not a big issue, but consider, for example, sorting strings.

Unlike Groovy, Java does not have native support for collections. Although collections have been a part of Java from the beginning in the form of arrays and the original `java.util.Vector` and `java.util.Hashtable` classes, a formal collections framework was added to the Java 2 Standard Edition, version 1.2. In addition to giving Java a small but useful set of fundamental data structures, such as lists, sets, and maps, the framework also introduced iterators that separated the way you moved through a collection from its underlying implementation. Finally, the framework introduced a set of polymorphic algorithms that work on the collections.

With all that in place we can assemble a collection of strings and sort them as shown in the following listing. First a collection of strings must be instantiated, then populated, and finally sorted.

Listing 1.1 Sorting strings using the `Collections.sort` method

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SortStrings {
    public static void main(String[] args) {
        List<String> strings = new ArrayList<String>();
        strings.add("this"); strings.add("is");
        strings.add("a");    strings.add("list");
        strings.add("of");   strings.add("strings");

        Collections.sort(strings);
        System.out.println(strings);
    }
}
```

Instantiating a list

Populate the list

A destructive sort

The collections framework supplies interfaces, like `List`, and implementation classes, like `ArrayList`. The `add` method is used to populate the list. Then the `java.util.Collections` utility class includes static methods for, among other things, sorting and searching lists. Here I'm using the single-argument `sort` method, which sorts its

[2] Read the post from March 30, 2006 at Steve Yegge's blog: http://mng.bz/E4MB.

argument according to its natural sort. The assumption is that the elements of the list are from a class that implements the `java.util.Comparable` interface. That interface includes the `compareTo` method, which returns a negative number if its argument is greater than the current object, a positive number if the argument is less than the current object, and zero otherwise. The `String` class implements `Comparable` as a lexicographical sort, which is alphabetical, but sorts capital letters ahead of lowercase letters.

We'll look at a Groovy equivalent to this in a moment, but let's consider another issue first. What if you want to sort the strings by length rather than alphabetically? The `String` class is a library class, so I can't edit it to change the implementation of the `compareTo` method. It's also marked final, so I can't just extend it and override the `compareTo` implementation. For cases like this, however, the `Collections.sort` method is overloaded to take a second argument, of type `java.util.Comparator`.

The next listing shows a second sort of our list of strings, this time using the comparator, implemented as an anonymous inner class. Instead of using a `main` method as in the previous example, here's a `StringSorter` class that sorts strings either using the default sort or by length.

Listing 1.2 A Java class to sort strings

```
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class StringSorter {
    public List<String> sortLexicographically(List<String> strings) {
        Collections.sort(strings);                                    <-- Default sort
        return strings;
    }

    public List<String> sortByDecreasingLength(List<String> strings) {
        Collections.sort(strings, new Comparator<String>() {          <-- Anonymous inner class
            public int compare(String s1, String s2) {
                return s2.length() - s1.length();
            }
        });
        return strings;
    }
}
```

Here we see a consequence of the triumph of the nouns over the verbs. The `Comparator` interface has a `compare` method, and all we want to do is to supply our own implementation of that method to `Collections.sort`. We can't implement a method, however, without including it in a class. In this case, we supply our own implementation (sort by length in decreasing order) via the awkward Java construct known as an anonymous inner class. To do so, we type the word `new` followed by the name of the interface we're implementing (in this case, `Comparator`), open a brace, and stuff in our implementation, all as the second argument to the `sort` method. It's an ugly, awkward syntax, whose only redeeming feature is that you do eventually get used to it.

Here's the Groovy equivalent in script form:

```
def strings = ['this','is','a','list','of','strings']
Collections.sort(strings, {s1,s2 -> s2.size() - s1.size()} as Comparator)
assert strings*.size() == [7, 4, 4, 2, 2, 1]
```

First of all, I'm taking advantage of Groovy's native support for collections by simply defining and populating a list as though it's an array. The `strings` variable is in fact a reference to an instance of `java.util.ArrayList`.

Next, I sort the strings using the two-argument version of `Collections.sort`. The interesting part is that the second argument to the `sort` method is a closure (between the braces), which is then "coerced" to implement `Comparable` using the `as` operator.[3]

The closure is intended to be the implementation of the `compare(String,String)` method analogous to that shown in the previous Java listing. Here I show the two dummy arguments, `s1` and `s2`, to the left of the arrow, and then use them on the right side. I provide the closure as the implementation of the `Comparator` interface. If the interface had several methods and I wanted to supply different implementations for each method, I would provide a map with the names of the methods as the keys and the corresponding closures as the values.

Finally, I use the so-called spread-dot operator to invoke the `size` method on each element of the sorted collection, which returns a list of results. In this case I'm asking for the length of each string in the collection and comparing the results to the expected values.

By the way, the Groovy script didn't require any imports, either. Java automatically imports the `java.lang` package. Groovy also automatically brings in `java.util`, `java.net`, `java.io`, `groovy.lang`, `groovy.util`, `java.math.BigInteger`, and `java.math.BigDecimal`. It's a small thing, but convenient.

> **GROOVY FEATURE** Native syntax for collections and additional automatic imports reduces both the amount of required code and its complexity.

If you've used Groovy before you probably know that there's actually an even simpler way to do the sort. I don't need to use the `Collections` class at all. Instead, Groovy has added a `sort` method to `java.util.Collection` itself. The default version does a natural sort, and a one-argument version takes a closure to do the sorting. In other words, the entire sort can be reduced to a single line:

```
strings.sort { -it?.size() }
```

The closure tells the `sort` method to use the result of the `size()` method on each element to do the sorting, with the minus sign implying that here I'm asking for descending order.

> **GROOVY FEATURE** Groovy's additions to the JDK simplify its use, and Groovy closures eliminate artificial wrappers like anonymous inner classes.

---

[3] Closure coercion like this is discussed further in chapter 4.

There were two major productivity improvements in this section. First, there are all the methods Groovy added to the Java libraries, known as the Groovy JDK. I'll return to those methods frequently. Second, I take advantage of Groovy's ability to treat methods as objects themselves, called closures. I'll have a lot to say about closures in the upcoming chapters, but the last example illustrated one advantage of them: you almost never need anonymous inner classes.

Incidentally, in the closure I used an additional Groovy feature to protect myself. The question mark after the word `it` is the safe de-reference operator. If the reference is null it invokes the `size` method here. If not it returns `null` and avoids the `NullPointerException`. That tiny bit of syntax wins over more Java developers to Groovy than I ever would have believed.[4]

1.1.3 Java is overly verbose

The following listing shows a simple POJO. In this case I have a class called `Task`, perhaps part of a project management system. It has attributes to represent the name, priority, and start and end dates of the task.

Listing 1.3 A Java class representing a task

```
import java.util.Date;

public class Task {
    private String name;
    private int priority;
    private Date startDate;
    private Date endDate;

    public Task() {}

    public Task(String name, int priority, Date startDate, Date endDate) {
        this.name = name;
        this.priority = priority;
        this.startDate = startDate;
        this.endDate = endDate;
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getPriority() { return priority; }
    public void setPriority(int priority) { this.priority = priority; }
    public Date getStartDate() { return startDate; }
    public void setStartDate(Date startDate) { this.startDate = startDate;}
    public Date getEndDate() { return endDate; }
    public void setEndDate(Date endDate) { this.endDate = endDate; }

    @Override
    public String toString() {
        return "Task [name=" + name + ", priority=" + priority +
```

Data we care about

Public getters and setters for the data

Typical override of toString

[4] Sometimes they get tears in their eyes. "Really?" they say. "I don't have to put in all those null checks?" It's touching how happy they are.

```
                ", startDate=" + startDate + ", endDate=" + endDate + "]";
    }
}
```

We have private fields and public getter and setter methods, along with whatever constructors we need. We also add a typical override of the `toString` method. I could probably use an override of `equals` and `hashCode` as well, but I left those out for simplicity.

Most of this code can be generated by an IDE, but it still makes for a long listing, and I haven't added the necessary `equals` and `hashCode` overrides yet. That's a lot of code for what's essentially a dumb data structure.

The analogous Plain Old Groovy Object (POGO) is shown here:

```
@EqualsAndHashCode
class Task {
    String name
    int priority
    Date startDate
    Date endDate

    String toString() { "($name,$priority,$startDate,$endDate)" }
}
```

Seriously, that's the whole class, and it does include overrides of the `equals` and `hashCode` methods. Groovy classes are public by default, as are Groovy methods. Attributes are private by default. Access to an attribute is done through dynamically generated getter and setter methods, so even though it looks like we're dealing with individual fields we're actually going through getter and setter methods. Also, Groovy automatically provides a map-based constructor that eliminates the need for lots of overloaded constructors. The `@EqualsAndHashCode` annotation represents an Abstract Syntax Tree (AST) transformation that generates the associated methods. Finally, I use a Groovy string with its parameter substitution capabilities to convert a task into a string.

> **GROOVY FEATURE** Groovy's dynamic generation capabilities drastically reduce the amount of code required in a class, letting you focus on the essence rather than the ceremony.

Java also includes checked exceptions, which are a mixed blessing at best. The philosophy is to catch (no pun intended) problems early in the development cycle, which is also supposed to be an advantage to static typing.

1.1.4 Groovy makes testing Java much easier

Just because a class compiles doesn't mean it's implemented correctly. Just because you've prepared for various exceptions doesn't mean the code works properly. You've still got to test it, or you don't really know.[5]

[5] My favorite example of this comes from a friend who used to teach C++ back when that language was shiny and new. He looked at a student's code, and it was a mess. Then he noticed the first line was `/*` and the last line was `*/`. He said, "You commented out your entire program." The student shrugged and said, "That's the only way I could get it to compile!"

One of the most important productivity improvements of the past decade or so has been the rise of automated testing tools. Java has tools like JUnit and its descendants, which make both writing and running tests automated and easy.

Testing is another area where Groovy shines. First, the base Groovy libraries include `GroovyTestCase`, which extends JUnit's `TestCase` class and adds a range of helpful methods, such as `testArrayEquals`, `testToString`, and even `shouldFail`. Next, Groovy's metaprogramming capabilities have given rise to simple DSLs for testing.

One particularly nice example is the Spock framework, which I'll discuss in chapter 6 on testing. Spock is lean and expressive, with blocks like `given`, `expect`, and `when/then`.

As an example, consider sorting strings, as implemented in Java and discussed earlier. In listing 1.3 I presented a Java class that sorted strings both lexicographically and by decreasing length. Now I'd like to test that, and to do so I'm going to use the Spock testing framework from Groovy.

A Spock test that checks both sorting methods is shown in the following listing.

Listing 1.4 A Spock test that checks each Java sorting method

```
import spock.lang.Specification;

class StringSorterTest extends Specification {
    StringSorter sorter = new StringSorter()
    def strings = ['this','is','a','list','of','strings']

    def "lexicographical sort returns alphabetical"() {
        when:
        sorter.sortLexicographically strings

        then:
        strings == ['a','is','list','of','strings','this']
    }

    def "reverse sort by length should be decreasing size"() {
        when:
        sorter.sortByDecreasingLength strings

        then:
        strings*.size() == [7, 4, 4, 2, 2, 1]
    }
}
```

Testing a Java class

Test the lexicographical sort

Test the reverse length sort

In the Spock test the Java class under test is instantiated as an attribute. I populate the data using the native collection in Groovy, even though the class under test is written in Java and the methods take Java lists as arguments.[6] I have two tests, and in each case, even without knowing anything about Spock, it should be clear what the tests are doing. I'm taking advantage of Groovy capabilities like optional parentheses and the

[6] Applying Groovy tests to Java code is discussed in chapter 6.

spread-dot operator, which applies to a list and returns a list with the specified properties only.

The test passes, and I can use the same test with the Groovy implementation. The point, though, is that I can add a Groovy test to a Java system without any problems.

1.1.5 Groovy tools simplify your build

Another area where Groovy helps Java is in the build process. I'll have a lot to say about Groovy build mechanisms in chapter 5, but here I'll just mention a couple of ways they help Java. If you're accustomed to using Apache Ant for building systems, Groovy adds execution and compilation tasks to Ant. Another option is to use `AntBuilder`, which allows you to write Ant tasks using Groovy syntax.

That's actually a common theme in Groovy, which I should emphasize:

> **GROOVY FEATURE** Groovy augments and enhances existing Java tools, rather than replacing them.

If your company has moved from Ant to Maven you're using a tool that works at a higher level of abstraction and manages dependencies for you. In chapter 5 two ways are provided to add Groovy to a Maven build. The Groovy ecosystem, however, provides another alternative.

In chapter 5 I discuss the latest Groovy killer app, Gradle. Gradle does dependency management based on Maven repositories (though it uses Ivy under the hood) and defines build tasks in a manner similar to Ant, but it's easy to set up and run. Maven is very powerful, but it has a lot of trouble with projects that weren't designed from the beginning with it in mind. Maven is a very opinionated framework, and customization is done through plugins. Ultimately, in Maven the build file is written in XML. Gradle is all about customization, and because the build file is written in Groovy you have the entire power of the Groovy language available to you.

That fact that Gradle build files are written in Groovy doesn't limit it to Groovy projects, though. If your Java project is in fact written in Maven form and has no external dependencies, here's your entire Gradle build file:

```
apply plugin:'java'
```

Applying the Java plugin defines a whole set of tasks, from compile to test to JAR. If that one line of code is in a file called build.gradle, then just type `gradle build` at the command line and a whole host of activities ensue. If you're (hopefully) going to do some testing, you'll need to add a dependency on JUnit, or even Spock. The resulting build file is shown here:

```
apply plugin:'java'

repositories {
    mavenCentral()
}
```

Standard Maven repository

```
dependencies {
    testCompile 'junit:junit:4.10'
    testCompile "org.spockframework:spock-core:0.7-groovy-2.0"
}
```

Dependencies in Maven form

Now running `gradle build` results in a series of stages:

```
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build
```

The result is a nice, hyperlinked set of documentation of all the test cases, plus a JAR file for deployment.

Of course, if there's a plugin called `java`, there's a plugin called `groovy`. Better yet, the Groovy plugin includes the Java plugin and, as usual, augments and improves it. If your project is similar to the ones discussed in this book, in that it combines Groovy and Java classes and uses each where most helpful, then all you need is the Groovy plugin and you're ready to go. There are many other plugins available, including `eclipse` and `web`. I'll talk about them in chapter 5 on build processes.

In this section I reviewed several of the features built into Java and how they can lead to code that's more verbose and complicated than necessary. I demonstrated how Groovy can streamline implementations and even augment existing Java tools to make them easier to use and more powerful. I'll show more details throughout the book. First I want to list some of the additional capabilities Groovy brings to Java in the next section.

1.2 Groovy features that help Java

I've actually been discussing these all along, but let me make a few specific points here. First, the Groovy version of a Java class is almost always simpler and cleaner. Groovy is far less verbose and generally easier to read.

As true as that statement is, though, it's a bit misleading. I'm not advocating rewriting all your Java code in Groovy. Quite the contrary; if your existing Java code works, that's great, although you might want to consider adding test cases in Groovy if you don't already have them. In this book, I'm more interested in helping Java than replacing it.

What does Groovy offer Java? Here's a short list of topics that are discussed in much more detail in the rest of the book:

1. ***Groovy adds new capabilities to existing Java classes.***
   Groovy includes a Groovy JDK, which documents the methods added by Groovy to the Java libraries. The various `sort` methods added to the `Collection` interface

that I used for strings was a simple example. You can also use Java classes with Groovy and add features like operator overloading to Java. These and related topics will be discussed in chapter 4.

2 ***Groovy uses Java libraries.***
Practically every Groovy class relies on the Java libraries, with or without Groovy additions. That means virtually every Groovy class is already an integration story, mixing Groovy and Java together. One nice use case for Groovy is to experiment with Java libraries you haven't used before.

3 ***Groovy makes working with XML and JSON easy.***
Here's an area where Groovy shines. Groovy includes classes called `MarkupBuilder`, which makes it easy to generate XML, and `JsonBuilder`, which produces JSON objects. It also has classes called `XmlParser` and `XmlSlurper`, which convert XML data structures into DOM structures in memory, and `JsonSlurper`, to parse JSON data. These will be used throughout the book, especially in chapter 9 on RESTful web services.

4 ***Groovy includes simplified data source manipulation.***
The `groovy.sql.Sql` class provides a very simple way to work with relational databases. I'll talk about this in chapter 8 on databases, chapter 7 on working with the Spring framework, and chapter 9 on RESTful web services.

5 ***Groovy's metaprogramming streamlines development.***
The builder classes are an example of Groovy metaprogramming. I'll show examples of DSLs in several chapters.

6 ***Groovy tests work for Java code.***
The Spock testing tool, demonstrated in this chapter and extensively discussed in chapter 6 on testing, is a great way to test Java systems.

7 ***Groovy build tools work on Java (and mixed) projects.***
In chapter 5 on enhancing build processes, I'll talk about `AntBuilder`, how to add Groovy to Maven builds, and Gradle.

8 ***Groovy projects like Grails and Griffon make developing web and desktop applications easier.***
The Grails project is a complete-stack, end-to-end framework for building web applications, based on Spring and Hibernate. Griffon brings the same convention-over-configuration ideas to desktop development. Grails is discussed in chapter 8 on databases and chapter 10 on web applications.

When looking at the sorts of problems Java developers typically encounter, this list will be a source of ideas for making implementations simpler, easier to read and understand, and faster to implement.

1.3 Java use cases and how Groovy helps

The examples I've discussed so far are all code-level simplifications. They're very helpful, but I can do more than that. Groovy developers work on the same sorts of problems

that Java developers do, so many higher-level abstractions have been created to make addressing those problems easier.

In this book I'm also going to survey the various types of problems that Java developers face on a regular basis, from accessing and implementing web services to using object-relational mapping tools to improving your build process. In each case I'll examine how adding Groovy can make your life easier as a developer.

Here's a list of some of the areas I'll discuss as we go along, and I'll give you a brief idea of how Groovy will help. This will also provide a lightweight survey of the upcoming chapters.

1.3.1 Spring framework support for Groovy

One of the most successful open source projects in the Java industry today is the Spring framework. It's the Swiss Army chainsaw of projects; it's pervasive throughout the Java world and has tools for practically every purpose.

No one is ever going to suggest rewriting Spring in Groovy. It works fine in Java as it is. Nor is there any need to "port" it to Groovy. As far as Spring is concerned, compiled Groovy classes are just another set of bytecodes. Groovy can use Spring as though it's just another library.

The developers of Spring, however, are well aware of Groovy and built in special capabilities for working with it. Spring bean files can contain *inline scripted* Groovy beans. Spring also allows you to deploy Groovy source code, rather than compiled versions, as so-called *refreshable* beans. Spring periodically checks the source code of refreshable beans for changes and, if it finds any, rebuilds them and uses the updated versions. This is a very powerful capability, as chapter 7 on working with Spring will show.

Finally, the developers of the Grails project also created a class called `BeanBuilder`, which is used to script Spring beans in Groovy. That brings Groovy capabilities to Spring bean files much the way Gradle enhances XML build files.

1.3.2 Simplified database access

Virtually all Java developers work with databases. Groovy has a special set of classes to make database integration easy, and I'll review them in chapter 8 on databases. I also show an example of working with a MongoDB database through a Groovy library that wraps the corresponding Java API.

I'll also borrow from the Grails world and discuss GORM, the Grails Object-Relational Mapping tool, a DSL for configuring Hibernate. In fact, GORM has been refactored to work with a variety of persistence mechanisms, including NoSQL databases like MongoDB, Neo4J, Redis, and more.

1.3.3 Building and accessing web services

Another area of active development today is in web services. Java developers work with both SOAP-based and RESTful services, the former involving auto-generated proxies and the latter using HTTP as much as possible. REST is covered in chapter 9, and

SOAP-based web services are discussed in appendix C, available as a free download. In both cases, if a little care is applied, the existing Java tools work just fine with Groovy implementations.

1.3.4 Web application enhancements

Groovy includes a "groovlet" class, which acts like a Groovy-based servlet. It receives HTTP requests and returns HTTP responses, and it includes pre-built objects for requests, responses, sessions, and more. One of the most successful instances of Groovy and Java integration, and arguably the killer app for Groovy, is the Grails framework, which brings extraordinary productivity to web applications. Both are covered in chapter 10 on web development.

In each of these use cases, Groovy can work with existing Java tools, libraries, and infrastructure. In some situations, Groovy will simplify the required code. In other cases, the integration is more deeply embedded and will provide capabilities far beyond what Java alone includes. In all of them, the productivity gains will hopefully be both obvious and dramatic.

1.4 Summary

Java is a large, powerful language, but it's showing its age. Decisions made early in its development are not necessarily appropriate now, and over time it has accumulated problems and inconsistencies. Still, Java is everywhere, and its tools, libraries, and infrastructure are both useful and convenient.

In this chapter I reviewed some of the issues that are part of the Java development world, from its verbosity to anonymous inner classes to static typing. Most Java developers are so accustomed to these "problems" that they see them as features as much as bugs. Add a little bit of Groovy, however, and the productivity gains can be considerable. I demonstrated that simply using Groovy native collections and the methods Groovy adds to the standard Java libraries reduced huge sections of code down to a few lines. I also listed the Groovy capabilities that will be a rich source of ideas for simplifying Java development.

As powerful as Groovy is (and as fun as it is to use), I still don't recommend replacing your existing Java with Groovy. In this book I advocate a blended approach. The philosophy is to use Java wherever it is appropriate, which mostly means using its tools and libraries and deploying to its infrastructure. I add Groovy to Java wherever it helps the most. In the next chapter I'll begin that journey by examining class-level integration of Java and Groovy.

JAVA

Making Java Groovy

Kenneth A. Kousen

Free eBook
SEE INSERT

You don't need the full force of Java when you're writing a build script, a simple system utility, or a lightweight web app—but that's where Groovy shines brightest. This elegant JVM-based dynamic language extends and simplifies Java so you can concentrate on the task at hand instead of managing minute details and unnecessary complexity.

Making Java Groovy is a practical guide for developers who want to benefit from Groovy in their work with Java. It starts by introducing the key differences between Java and Groovy and how to use them to your advantage. Then, you'll focus on the situations you face every day, like consuming and creating RESTful web services, working with databases, and using the Spring framework. You'll also explore the great Groovy tools for build processes, testing, and deployment and learn how to write Groovy-based domain-specific languages that simplify Java development.

What's Inside

- Easier Java
- Closures, builders, and metaprogramming
- Gradle for builds, Spock for testing
- Groovy frameworks like Grails and Griffon

Written for developers familiar with Java. No Groovy experience required.

Ken Kousen is an independent consultant and trainer specializing in Spring, Hibernate, Groovy, and Grails.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/MakingJavaGroovy

"Focuses on the tasks that Java developers must tackle every day."
—From the Foreword by Guillaume Laforge Groovy Project Manager

"Thoroughly researched, highly informative, and mightily entertaining."
—Michael Smolyak Next Century Corporation

"A comprehensive tour through the Groovy development ecosystem."
—Sean Reilly Equal Experts in the UK

"I measured this book's ROI in Revelations per Minute."
—Tim Vold, Minnesota State Colleges and Universities

ISBN 13: 978-1-935182-94-8
ISBN 10: 1-935182-94-3
54499
9 781935 182948

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]