

# Making Java Groovy

Kenneth A. Kousen



 MANNING



**MEAP Edition**  
**Manning Early Access Program**  
**Making Java Groovy version 5**

Copyright 2011 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# *Table of Contents*

## **Part 1: Up to Speed with Groovy**

1. Why Add Groovy to Java
2. Groovy by Example
3. Code-Level Integration

## **Part 2: Groovy Tools**

4. Enhance your Build
5. Testing

## **Part 3: Groovy in the Real World**

6. The Spring Framework
7. Working with Databases
8. SOAP-based Web Services
9. RESTful Web Services
10. Web Applications
11. Desktop Applications and Swing

## **Part 4: The Future**

12. Concurrency
13. Groovy in the Cloud

## **Appendixes**

- A. Groovy by Feature

# 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 is 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 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. We 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, we're asking you to consider an alternative.

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

Throughout, our mantra will be:

**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, which we'll discuss in detail in a later chapter. Likewise, the Java Virtual Machine is everywhere. That's a good thing, because wherever Java can run, so can Groovy, as shown in Figure x.

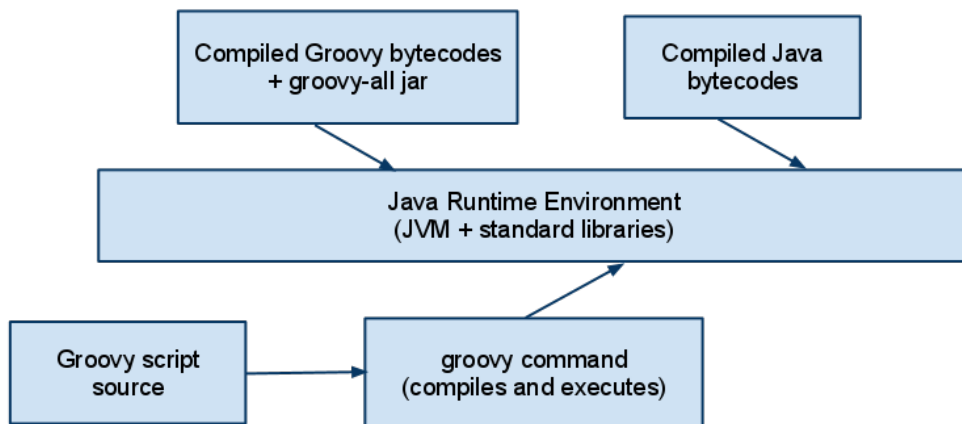


Figure 1.1 Groovy generates bytecodes for the Java Virtual Machine. Either compile them ahead of time, or let the groovy command generate them from source.

We'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's first review where Java might have some issues that need help.

## 1.1 Issues with Java

As we see over and over again in the IT industry, the best technology doesn't always win. Just because something is popular doesn't necessarily mean that it's good. A perfect storm swept through the development world in the mid to late 1990's, 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 (and 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 is overly verbose.
- Java forbids operator overloading.
- The Java libraries have accumulated inconsistencies over the years.
- The default access modifier is still, strangely enough, "package private".
- Java treats primitives differently from classes.
- It's hard to implement Domain Specific Languages in Java.
- Java testing tools are a good start, but could use improvement.
- All attempts to standardize Java build processes have drawbacks.

There are more, 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 viewed as a benefit. The combination of static typing and dynamic binding meant that we 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, since they can use the types to prompt developers for the correct fields and methods. IDE's like Eclipse or 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 unnecessarily restrictive. Dynamically typed languages have much more freedom to let one object stand in for another.

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 had a language with dynamic typing, and all we are invoking is the `append` method on it using character arguments, we could supply a string wherever we had an array and everything would 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 class 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 x.

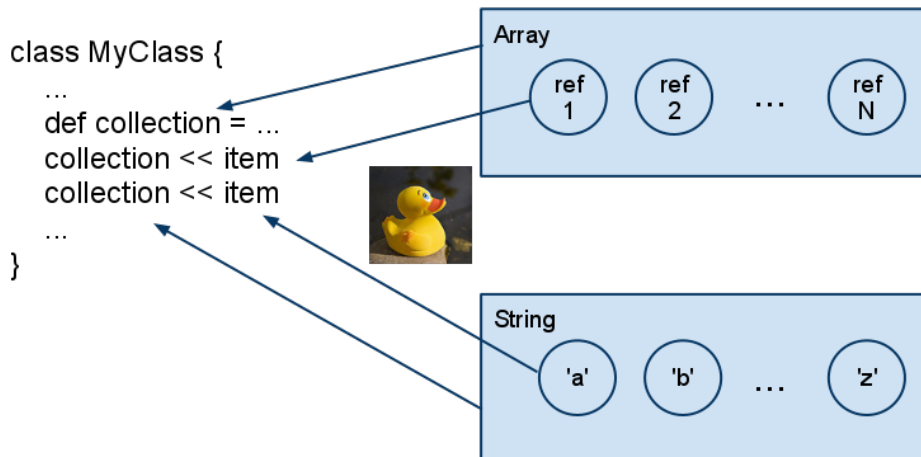


Figure 1.2 Arrays and strings from a duck typing point of view. Each is a collection with an `append` method. If that's all we care about, they're the same.

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, `*` uses `times()`, and so on. 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.

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 entitled “Execution in the Kingdom of the Nouns”<sup>1</sup>. 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, like lists, sets, and maps, the framework also introduced iterators that separated the way you move 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.

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

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

public class SortStrings {
```

<sup>1</sup> <http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html>

```

public static void main(String[] args) {
    List<String> strings = new ArrayList<String>(); #1
    strings.add("this"); strings.add("is");      #2
    strings.add("a");   strings.add("list");     #2
    strings.add("of");  strings.add("strings");  #2

    Collections.sort(strings); #3
    System.out.println(strings);
}

```

**#1 Instantiating a list**

**#2 Populate the list**

**#3 A destructive sort**

The collections framework supplies interfaces, like `List` (#1), and implementation classes, like `ArrayList` (#1). The `add` method (#2) 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 (#3) we use the single-argument `sort` method, which sorts its 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 we can't edit it to change the implementation of the `compareTo` method. It's also marked `final`, so we 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`.

Here's a second sort of our list of strings, this time using the comparator.

```

Collections.sort(strings, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s2.length() - s1.length();
    }
});
System.out.println(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 say the word `new` followed by the name of the interface we're

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=724>

implementing (in this case, `Comparable`), 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.

Now let's look at the Groovy equivalent.

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

Next, we sort our strings using the two-argument version of `Collections.sort`. We're taking advantage of the fact that the parentheses are optional if there is no ambiguity, which removes some clutter. 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.

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

Finally, we use the spread-dot operator to invoke the `size` method on each element of the sorted collection, which returns a list of results. In this case, we're 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 assumes that the `java.lang` package is always automatically imported. Groovy also automatically brings in `java.util`, `java.net`, `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 reduce 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. We 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, our entire sort can be reduced to a single line.

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

We tell the `sort` method to use the `size()` method of each element for sorting, with the minus sign implying that we're 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.

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

### 1.1.3 Java is overly verbose

In the following listing is a simple POJO. In this case we have a class called `Task`, presumably part of a project management system. It has attributes to represent the name, priority, and start and end dates of the task. We'll show the Groovy version later.

#### Listing 1.2 A Java class representing a Task.

```
import java.util.Date;

public class Task {
    private String name;           #1
    private int priority;         #1
    private Date startDate;       #1
    private Date endDate;        #1

    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; }           #2
    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() {                       #3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=724>

```

        return "Task [name=" + name + ", priority=" + priority + ",
            startDate=" + startDate + ", endDate=" + endDate + "];
    }
}
#1 Data we care about
#2 public getters and setters for the data
#3 typical override of toString

```

We have private fields (#1) and public getter and setter methods (#2), along with whatever constructors we need. We also add a typical override of the `toString` method. We could probably use an override of `equals` and `hashCode` as well, but we left those out for simplicity.

First, we need to note that although most of this code can be autogenerated by an Integrated Development Environment (IDE), the result is still extensive. The listing is straightforward, but that's a lot of code for what is essentially a dumb data structure.

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

```

class Task {
    String name
    int priority
    Date startDate
    Date endDate

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

```

Seriously, that's the whole class. 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 provides a map-based constructor automatically that eliminates the need for lots of overloaded constructors. Finally, we 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 correct. 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.

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 descendents, 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, like `testArrayEquals`, `testToString`, or even `shouldFail`. Next, Groovy's metaprogramming capabilities have given rise to simple Domain Specific Languages (DSL) for testing.

One particularly nice example is the Spock framework, which we'll discuss in the testing chapter. 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 above. In our previous listing, we executed everything in a main method, which worked but was difficult to test. Here we refactor our sorting into a `StringSorter` class, as shown in the following listing.

### Listing 1.3 A Java class for sorting strings by length or lexicographically

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

public class StringSorter {
    public List<String> sortLexicographically(List<String> strings) { #1
        Collections.sort(strings);
        return strings;
    }

    public List<String> sortByDecreasingLength(List<String> strings) {
        Collections.sort(strings, new Comparator<String>() { #2
            @Override
            public int compare(String s1, String s2) {
                return s2.length() - s1.length();
            }
        });
        return strings;
    }
}
#1 Natural sort defined in the Task class
#2 Sorting using a Comparator
```

The listing wraps both our lexicographical sort (#1) and our decreasing length sort that uses a `Comparator` (#2) into methods that can be invoked from outside. The sort methods from the `Collections` class sort in place, rather than creating a new collection of sorted results, so I return the provided arguments after they've been modified.

A Spock test, written in Groovy, which checks both sorting methods is shown in the following listing.

### Listing 1.4A Spock test that checks each Java sorting method

```
import spock.lang.Specification;

class StringSorterTest extends Specification {
    StringSorter sorter #1
    def strings

    def setup() { #2
        sorter = new StringSorter()
        strings = ['this', 'is', 'a', 'list', 'of', 'strings']
    }

    def "lexicographical sort returns alphabetical"() { #3
        when:
            sorter.sortLexicographically strings

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

    def "reverse sort by length should be decreasing size"() { #4
        when:
            sorter.sortByDecreasingLength strings

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

#1 Testing a Java class
#2 Prepare test data
#3 Test the lexicographical sort
#4 Test the reverse length sort
```

In our Spock test, we're testing a Java class (#1). We 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. We'll have more to say about this kind of code-level integration in Chapter 2. We have two tests (#3 and #4), and in each case even without knowing anything about Spock it should be clear what the tests are doing. We take advantage of Groovy capabilities like optional parentheses and the spread-dot operator, which applies to a list and returns a list with the specified properties only.

#### 1.1.5 Groovy tools simplify your build

Another area where Groovy helps Java is in the build process. We'll have lots to say about Groovy builders in general, but here we'll just mention a couple of ways they help Java. If you're accustomed to using Apache Ant for building systems, Groovy adds two options. The first, called AntBuilder, makes it easy to write Ant tasks using Groovy syntax. The other, Gant, takes that to another level, allowing you to script entire Ant-based builds

using Groovy. As explained in the documentation, Gant is “a lightweight façade on Groovy’s AntBuilder.”

That’s actually a common theme in Groovy, which we should emphasize.

**Groovy Feature** Groovy augments and enhances existing Java tools, rather than replaces them.

Many companies have moved from Ant to Maven, a tool that works at a higher level of abstraction than Ant and also manages dependencies for you. Whether Maven’s transitive dependency management is a bug or a feature is a debatable point, but it’s certainly a fact of life. Groovy offers GMaven as a way to add Groovy to Maven, but recently a better way to build has appeared.

In the build chapter, we’ll 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 is easy to set up and run. Maven is very powerful, but has a lot of trouble with projects that weren’t designed from the beginning with it in mind. It’s particularly hard to customize. Gradle is all about customization, acknowledging that virtually every build is a custom build.

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. Let’s be a bit more realistic, however, and at least assume we’re at least going to do some testing. That means we’ll have a dependency on JUnit, and, as we saw, Spock. Our build file is shown here.

```
apply plugin: 'java'

repositories {
    mavenCentral() #1
}

dependencies {
    #2
    testCompile 'junit:junit:4.8.1'
    testCompile "org.spockframework:spock-core:0.4-groovy-1.7"
}

#1 Standard Maven repository
#2 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`. We'll talk about them in the build chapter.

In this section, we reviewed several of the features built into Java and how they can lead to more verbose, complicated code than necessary. We demonstrated how Groovy can streamline the implementations and even augment existing Java tools to make them easier to use and more powerful. We'll see more details throughout the book, but let's list some of the new capabilities in the next section.

## 1.2 Groovy features that help Java

We've actually been discussing these all along, but let's make a few specific points here. First, as we've seen repeatedly, 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 here. We're not advocating rewriting all your Java code in Groovy. Quite the contrary, actually – if your existing Java code works, that's great, though you might want to consider adding test cases in Groovy if you don't already have them. In this book, we're much more interested in helping Java than replacing it.

What does Groovy offer Java? Here's another short list of topics that we'll discuss 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 we used for strings was a simple example. We can also use Java classes with Groovy, and add features like operator overloading to Java. These and related topics will be discussed in Chapter 2.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=724>

2. **Groovy leverages 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 easy.**

Here is an area where Groovy shines. Groovy includes a class called `MarkupBuilder`, which makes it easy to generate XML. It also has classes called `XmlParser` and `XmlSlurper`, which convert XML data structures into DOM structures in memory. Both will be discussed in the next chapter, as well as in the chapters on web services.

4. **Groovy includes simplified data source manipulation.**

The `groovy.sql.Sql` class provides a very simple way to work with databases. We'll talk about this in Chapter x.

5. **Groovy's metaprogramming streamlines development.**

The builder classes are an example of Groovy metaprogramming. We'll see examples of DSL's in several chapters. We'll also discuss writing them and applying them to Groovy and Java projects.

6. **Groovy tests work for Java code.**

The Spock testing tool, demonstrated in this chapter and extensively discussed in the testing chapter, is a great way to test Java systems.

7. **Groovy build tools work on Java (and mixed) projects.**

In the build chapter, we'll talk about `AntBuilder` and `Gant`, which bring Groovy to Ant builds. We'll especially feature `Gradle` as a way to make the build process much easier.

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.

When looking at the sort 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 we've discussed so far are all code-level simplifications. They're very helpful, but we 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 we're 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, we'll examine how adding Groovy can make your life easier as a developer.

Here is a list many of the areas we'll discuss as we go along, and give you a brief idea where Groovy will help. This will also provide a light-weight survey of the upcoming chapters.

### **SPRING FRAMEWORK SUPPORT FOR GROOVY**

One of the most successful open source projects in 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 Groovy is concerned, it's just another set of byte codes, so who cares where they came from? Groovy can use Spring as though it was 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 we'll see.

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 Gant or Gradle enhances XML build files.

We'll discuss all these capabilities in the chapter on Groovy with Spring.

### **SIMPLIFIED DATABASE ACCESS**

Virtually all Java developers work with databases. Groovy has a special set of classes to make database integration easy, and we'll review them in our database chapter. We'll also borrow from the Grails world and discuss GORM, the Grails Object-Relational Mapping tool, a DSL for configuring Hibernate.

### **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 leveraging HTTP as much as possible. We have a chapter on each, below. In both cases, we'll see that if a little care is applied, the existing Java tools work just fine with Groovy implementations.

### **WEB APPLICATION ENHANCEMENTS**

Groovy includes a "groovlet" class, which acts like a Groovy-based servlet. It receives HTTP requests and returns HTTP responses, and includes pre-built objects for requests, responses, sessions, and more. We can use it in Java web applications easily. Of course,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=724>

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. We'll discuss both in the chapter on web development.

### **DESKTOP GROOVY APPLICATIONS**

Though Java has been most successful on the server side, there is a small but enthusiastic group of developers who focus on client side GUI development. In our chapter on Desktop Groovy we'll talk about how Groovy annotations help implement good Model-View-Controller architectures, eventually resulting in Griffon, the project that promises to bring Grails-like productivity to desktop 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, we 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. We saw that simply using Groovy native collections and the methods Groovy adds to the standard Java libraries can reduce huge swathes of code down to a few lines. We 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), we still don't recommend replacing your existing Java with Groovy. In this book, we advocate a blended approach. Our philosophy is to use Java wherever we can, which mostly means using its tools and libraries and deploying to its infrastructure. We add Groovy to Java wherever it helps the most. In the next chapter, we'll begin that journey by examining class-level integration of Java and Groovy.