Beginning Groovy and Grails: From Novice to Professional

ISBN-13 (pbk): 978-1-4302-1045-0

ISBN-13 (electronic): 978-1-4302-1046-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Steve Anglin, Matthew Moodie
Technical Reviewer: Guilliaume Laforge
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Kylie Johnston
Copy Editors: Nicole Abramowitz, Marilyn Smith
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Kinetic Publishing Services, LLC
Proofreader: Liz Welch
Indexer: Julie Grady
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Introduction to Groovy

In 1995, Java changed the world. The Internet was in its infancy, and most web sites offered only static content. But Java changed that by enabling applications called *applets* to run inside the browser on many different platforms. Java became a popular general-purpose language, but its greatest growth and strength has been on the server side. It is now one of the dominant server-side platforms. But Java is starting to show its age. Many people are even beginning to call it the new COBOL.

With all these years of baggage, Java has become difficult. There are large barriers of entry, such as knowing which of the many competing frameworks and specifications to use. The language itself has remained pretty much unchanged since the early days to help support backward-compatibility. At this point, many organizations are faced with a dilemma. Should they switch to a platform like Ruby, LAMP (an open source platform based on Linux, Apache, MySQL, and PHP, Perl, or Python), or possibly even .NET to try to become more productive and agile at lower costs so they can better compete in the marketplace? Do they stick with Java and try to make the most of the large investments they have made in frameworks, code, education, and infrastructure? Or do they implement a hybrid and work through integration issues?

Fortunately, there is another option. Keep what is great about the Java platform, specifically the Java Virtual Machine (JVM) and the large library of Java Application Programming Interfaces (APIs), and augment the Java language with a more flexible and productive language. In recent years, many languages have competed to become the Java language replacement for the JVM. Implementations of languages like Ruby, Python, and JavaScript run on the JVM. But none of these languages show as much promise as Groovy, a dynamic language made specifically for the JVM.

In this chapter, we will introduce the Groovy language, describe how to install it, and give you an idea of the benefits of Groovy by working through an example.

Groovy Language Features

Groovy is a relatively new dynamic language that can either be interpreted or compiled and is designed specifically for the Java platform. It has been influenced by languages such as Ruby, Python, Perl, and Smalltalk, as well as Java.

Unlike other languages that are ported to the JVM, Groovy was designed with the JVM in mind, so there is little to no impedance mismatch, significantly reducing the learning curve. Java developers will feel right at home with Groovy. For example, Groovy relies on the Java API rather than supplying its own API, so developers do not need to decide between the IO package from Java and the IO methods from the other language libraries. In addition, because Groovy is built for the JVM, there is tight bytecode-level integration that makes it easy for Java to integrate with Groovy and Groovy to integrate with Java.

Groovy does not just have access to the existing Java API; its Groovy Development Kit (GDK) actually extends the Java API by adding new methods to the existing Java classes to make them more Groovy.

Groovy has support for many of the modern programming features that make other languages so productive, such as closures and properties. Groovy has also proven to be a great platform for concepts such as metaprogramming and domain-specific languages.

Groovy is a standard governed by the Java Community Process (JCP)[1] as Java Specification Request (JSR) 241.[2] It is hosted on Codehaus at `http://groovy.codehaus.org`.

Groovy Installation

Groovy comes bundled as a `.zip` file or platform-specific installer for Windows, and Ubuntu, Debian (as well as openSUSE until recent versions). This section will explain how to install the zipped version, since it covers the widest breadth of platforms.

■ **Note** Because Groovy is Java, it requires Java Development Kit (JDK) 1.4 or above to be installed and the `JAVA_HOME` environment variable to be set.

To install Groovy, follow these steps:

1. Download the most recent stable Groovy binary release `.zip` file from `http://groovy.codehaus.org/Download`.

2. Uncompress `groovy-binary-X.X.X.zip` to your desired location.

---

1. `http://www.jcp.org`
2. `http://www.jcp.org/en/jsr/detail?id=241`

3. Set a `GROOVY_HOME` environment variable to the directory in which you uncompressed the `.zip` file.

4. Add the `%GROOVY_HOME%\bin` directory to your system path.

To validate your installation, open a console and type the following:

```
> groovy -version
```

You should see something like this:

```
Groovy Version: 1.5.6 JVM: 1.6.0_02-b06
```

Groovy by Example

The best way to grasp the power and elegance of Groovy is to compare it to Java using an example. In the remainder of this chapter, we will show you how to convert the simple Java class in Listing 1-1 into Groovy. Then we will demonstrate how to adapt the code to use common Groovy idioms.

Listing 1-1. *Simple Java Class*

```
01 package com.apress.bgg;
02
03 import java.util.List;
04 import java.util.ArrayList;
05 import java.util.Iterator;
06
07 public class Todo {
08   private String name;
09   private String note;
10
11   public Todo() {}
12
13   public Todo(String name, String note) {
14     this.name = name;
15     this.note = note;
16   }
17
18   public String getName() {
19     return name;
20   }
```

```
21
22   public void setName(String name) {
23     this.name = name;
24   }
25
26   public String getNote() {
27     return note;
28   }
29
30   public void setNote(String note) {
31     this.note = note;
32   }
33
34   public static void main(String[] args) {
35     List todos = new ArrayList();
36     todos.add(new Todo("1", "one"));
37     todos.add(new Todo("2", "two"));
38     todos.add(new Todo("3","three"));
39
40     for(Iterator iter = todos.iterator();iter.hasNext();) {
41       Todo todo = (Todo)iter.next();
42       System.out.println(todo.getName() + " " + todo.getNote());
43     }
44   }
45 }
```

If you have any Java experience, you will recognize Listing 1-1 as a basic `Todo` JavaBean. It has getters and setters for `name` and `note` attributes, as well as a convenience constructor that takes a `name` and `note` for initializing new instances. As you would expect, this class can be found in a file named `Todo.java` in the `com.apress.bgg` package.

The class includes a `main()` method, which is required for Java classes to be executable and is the entry point into the application. On line 35, the `main()` method begins by creating an instance of a `java.util.ArrayList` to hold a collection of `Todo`s. On lines 36–38, three `Todo` instances are created and added to the `todos` list. Finally, on lines 40–43, a `for` statement is used to iterate over the collection and print the `Todo`'s name and note to `System.out`. Notice that on line 41, the object returned from the iterator must be cast back to a `Todo` so the `getName()` and `getNote()` methods can be accessed. This is required because Java is type-safe and because prior to Java 1.5 and the introduction of generics, the Java collections API interface used `java.lang.Object` so it could handle any and all Java objects.

Converting Java to Groovy

To convert the Java `Todo` class in Listing 1-1 to Groovy, just rename the file to `Todo.groovy`. That's right, Groovy derives its syntax from Java. This is often referred to as *copy/paste compatibility*. So congratulations, you are a Groovy developer (even if you didn't know it)!

This level of compatibility, along with a familiar API, really helps to reduce the Groovy learning curve for Java developers. It also makes it easier to incorporate Java examples found on the Internet into a Groovy application and then refactor them to make them more Groovy-like, which is what we will do with Listing 1-1.

To run this Groovy application, from the command line, type the following:

```
> groovy com\apress\bgg\Todo.groovy
```

If you are coming from a Java background, you may be a little surprised that you did not need to first compile the code. Here's the Java equivalent:

```
> javac com\apress\bgg\Todo.java
> java com.apress.bgg.Todo
```

Running the Java application is a two-step process: compile the class using `javac`, and then use `java` to run the executable class in the JVM. But Groovy will compile to bytecode at runtime, saving a step in the development process and thereby increasing Groovy's productivity.

Groovy provides a lot of syntactic sugar and is able to imply more than Java. You'll see this in action as we make our Groovy application more Groovy by applying some of the Groovy idioms.

Converting a JavaBean to a GroovyBean

Let's begin by simplifying the JavaBean, which could also be referred to as a Plain Old Java Object (POJO). Groovy has the *GroovyBean*, which is a JavaBean with a simpler Groovy syntax, sometimes referred to as a Plain Old Groovy Object (POGO). GroovyBeans are publicly scoped by default. Listing 1-2 shows our example using a GroovyBean.

Listing 1-2. *Simple Example Using a GroovyBean*

```
01 package com.apress.bgg;
02
03 import java.util.List;
04 import java.util.ArrayList;
05 import java.util.Iterator;
06
07 public class Todo {
```

```
08
09   String name;
10   String note;
11
12   public static void main(String[] args) {
13     List todos = new ArrayList();
14     todos.add(new Todo(name:"1", note:"one"));
15     todos.add(new Todo(name:"2", note:"two"));
16     todos.add(new Todo(name:"3", note:"three"));
17
18     for(Iterator iter = todos.iterator();iter.hasNext();) {
19       Todo todo = (Todo)iter.next();
20       System.out.println(todo.name + " " + todo.note);
21     }
22   }
23 }
```

Listing 1-2 is significantly shorter than Listing 1-1, primarily because Groovy has a concept of native properties, which means getters and setters do not need to be declared. By default, all class attributes—such as the `name` and `note` attributes on lines 9 and 10—are public properties and automatically generate corresponding getters and setters in the bytecode. So if the class is used from Java code, or reflection is used to interrogate the class, you will see the getters and setters.

These properties also have a more intuitive usage model. They can be assigned or used directly, as on line 20, where the `name` and `note` properties, rather than the getters, are used to generate the output. Also, rather than needing to explicitly create a convenience constructor for initializing a GroovyBean, you can pass named parameters in the constructor to initialize any properties you want, as in lines 14–16.

Simplifying the Code

Some of the syntax sugar included in the Groovy language is making semicolons, parentheses, and data typing optional. Other interesting features to simplify code include implicit imports like the `java.util.*` package, common methods like `println()` applying to all objects including Java objects, and more flexible strings. Listing 1-3 applies these features to our example.

Listing 1-3. *Simple Example Applying Syntactic Sugar, Implicit Imports, Common Methods, and String Features*

```
01 package com.apress.bgg;
02
```

```
03 public class Todo {
04
05   String name
06   String note
07
08   public static void main(String[] args) {
09     def todos = new ArrayList()
10     todos.add(new Todo(name:"1", note:"one"))
11     todos.add(new Todo(name:"2", note:"two"))
12     todos.add(new Todo(name:"3", note:"three"))
13
14     for(Iterator iter = todos.iterator();iter.hasNext();) {
15       def todo = iter.next()
16       println "${todo.name} ${todo.note}"
17     }
18   }
19 }
```

In Listing 1-3, under the package declaration we no longer need to import `java.util.List`, `java.util.ArrayList`, and `java.util.Iterator`. These are implicitly imported since they are in the `java.util.*` package. Other implicitly included packages are `java.lang.*`, `java.net.*`, `java.io.*`, `groovy.lang.*`, and `groovy.util.*`.

Also notice that, other than in the `for` statement (which we will clean up in the next round of refactoring), all the semicolons have been removed.

On line 16, we have used optional parentheses with the implicit `println()` method. But that is not the only change to line 16. The `println()` method has been modified to use Groovy's GString format, which is similar to the Apache Ant[3] property format, rather than concatenating two strings. We'll cover Groovy strings in Chapter 2. At this point, just notice how much simpler this is to read.

Lines 9 and 15 have been changed to use optional typing. The variables `todos` and `todo` are no longer typed to `List` or `Todo`, respectively. Groovy uses "duck typing," which means if it sounds like a duck and walks like a duck, it must be a duck. Do you really care what the type of an object is, as long as you can pass it a message and it will handle the request if it can? If the object cannot handle the request, you will receive a `groovy.lang.MissingMethodException` or `groovy.lang.MissingPropertyException`. Of course, where you think typing is necessary, you always have the option of explicitly typing variables.

3. `http://ant.apache.org`

Using Groovy Collection Notation and Closure

The next step in refactoring the example is to take advantage of Groovy's collection and map notation, as well as replace the ugly `for` statement with a more elegant closure. Listing 1-4 shows this version.

Listing 1-4. *Example with the Groovy Collection Notation and Closure*

```
01 package com.apress.bgg;
02
03 public class Todo {
04
05   String name
06   String note
07
08   public static void main(String[] args) {
09     def todos = [
10       new Todo(name:"1", note:"one"),
11       new Todo(name:"2", note:"two"),
12       new Todo(name:"3", note:"three")
13     ]
14
15     todos.each {
16       println "${it.name} ${it.note}"
17     }
18   }
19 }
```

Notice how the `ArrayList` was replaced with `[]`. Again, this is just syntactic sugar; Groovy really is instantiating an `ArrayList`. Similarly, we can create maps with the `[:]` syntax.

Also to make the code more clean, we can initialize the list without needing to call the `add()` method for each entry. Then to simplify the iteration, we call the `each()` method, passing a closure that prints out the string. Notice that, by default, the iteration variable is `it`. Chapter 2 will provide more explanations and examples of Groovy lists, maps, and closures.

Getting Rid of Main()

One bit of Java ugliness left in our example is the `main()` method. After all these improvements, the `main()` method now just sticks out. Fortunately, Groovy has a concept of scripts as well as classes, and we can turn this into a script, removing the need for the `main()` method.

To begin, the file must be renamed to something like `Todos.groovy`. This is because a script will also be compiled to a class, and if we didn't change the name, there would be a name clash between the `Todo` class and the `Todo` script.

Then we simply move the code that currently exists in the `main()` method outside the `Todo` class. When the script is run, it will behave the same as before. Listing 1-5 shows the script version.

Listing 1-5. *Example As a Script*

```
package com.apress.bgg;

public class Todo {

  String name
  String note

}

def todos = [
  new Todo(name:"1", note:"one"),
  new Todo(name:"2", note:"two"),
  new Todo(name:"3", note:"three")
]

todos.each {
  println "${it.name} ${it.note}"
}
```

Finally, we have elegant, easy-to-read code at a fraction of what we started with in Java. It should be obvious that if we had started with the Groovy idioms to begin with, the Groovy approach would have been much more productive.

Summary

This chapter provided a brief introduction to Groovy. After describing how to install it, we demonstrated how you can dramatically reduce the code it takes to write the equivalent Java class in Groovy, while increasing the readability and expressiveness. In the next chapter, we will continue exploring Groovy by looking at its basic language features.