Groovy and Grails Recipes

ISBN-13 (pbk): 978-1-4302-1600-1

ISBN-13 (electronic): 978-1-4302-1601-8

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

Lead Editors: Steve Anglin, Tom Welsh
Technical Reviewer: Dave Klein
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Sharon Wilkey
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Lynn L'Heureux
Proofreaders: Linda Seifert and Patrick Vincent
Indexer: Carol Burbo
Artist: April Milne
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

Getting Started with Groovy

Let me start by congratulating you for making the decision to learn more about Groovy. Rest assured that the time you spend reading it will repay you well. Groovy is a wonderfully crafted language with great capabilities. When you see how much time and effort Groovy can save you, you will wish you had discovered it earlier. Groovy—some of the best news in the Java community in a long time—can greatly enhance the productivity and efficiency of Java developers and non-Java developers alike.

Note It is important to make the distinction between Java *the language* and Java *the platform*. When using the word *Java* by itself in this book, I am referring to the language. I use the term *Java platform* to refer to the Java virtual machine (JVM).

1-1. What Is Groovy?

Groovy is a programming language with a Java-like syntax that compiles to Java bytecode and runs on the JVM. Groovy integrates seamlessly with Java and enables you to mix and match Groovy and Java code with minimal effort.

Groovy has a Java-like syntax to make it easier for Java programmers to learn. However, Groovy's syntax is much more flexible and powerful than Java's. Think of Groovy as Java on steroids; dozens of lines of code in Java can be shortened to a few lines of code in Groovy with little to no sacrifice in readability, maintainability, or efficiency.

Some people refer to Groovy as a scripting language, a term I don't like to use because Groovy is much more than a language for writing scripts. It is a full-fledged, fully object-oriented language with many advanced features. Groovy has many applications—from writing quick and dirty shell scripts to building complex, large-scale projects with thousands of lines of code.

1-2. What Is Wrong with Java?

Java, the most popular programming language on earth today, has a huge user base and a plethora of libraries and add-ons. Although it is very well designed for the most part, the language is beginning to show its age. It doesn't shine well in a few areas, which can cause major frustrations for developers.

To start with, Java is unnecessarily verbose. Anyone who has ever tried to read from or write to a disk file in Java (two very common tasks) knows that such a simple job takes at least ten lines of code. Some people might argue that verbosity increases the readability and maintainability of a language. Although this might be true to a certain extent, Java is so verbose that it could be made a lot terser with no sacrifice in clarity.

Second, despite what some people might believe, Java is not a purely object-oriented language. It has primitive types (such as `int`, `long`, and `double`) that are not objects and have no reference semantics. Operators in Java (such as +, *, and -) can operate on primitive types only and not on objects (with the exception of `String` concatenation using the + operator). This can cause confusion to newcomers to the language and makes working with collections (which are essential in any language) unnecessarily painful.

Third, Java has no language-level support for collections (that is, it has no literal declaration for collections such as lists or maps, as it has for arrays). If you have ever worked with languages such as Python, Ruby, or Lisp, you know that supporting collections at the language level makes them much more usable and adds a lot of flexibility and power to the language.

Fourth, Java lacks many advanced features that exist in other languages. Closures, builders, ranges, and metaprogramming are concepts that might not be familiar to Java programmers, but these features could greatly enhance the productivity and efficiency of Java developers if they were available. Every new version of Java seems to add new features to the language (for example, generics were introduced in Java 5.0). However, to ensure backward and migration compatibility, a lot of these features are not correctly implemented and can adversely affect the language. Generics, for example, are very limited in Java because of the unnecessary use of erasures. The new proposed syntax for closures is complicated and clunky. Adding new features to the Java language at this point is not an easy task, and I believe that it's better to focus efforts on new languages that run on the Java platform.

Finally, there is no quick way to write scripts in Java or to perform sanity checks on your Java code. Because everything in Java has to be enclosed by a class, and must have an executable `main` method for the class to run, there is no quick way to execute just the code you wish to test. For example, suppose you forgot whether the `substring(int beginIndex, int endIndex)` method in Java's `String` class includes or excludes the `endIndex` from the resulting substring. Let's also assume that for some reason you can't access the API docs for that class, and the only way for you to find out what `substring` does is to write a small program to test it. The shortest possible program to test such a method will contain at least three lines of code, as shown in Listing 1-1.

Listing 1-1. *Testing the substring Method in Java*

```
public class SubStringTest {
  public static void main(String[] args) {
    System.out.println("Test_String".substring(0,4));
  }
}
```

You will also need to compile the class first with the `javac` command and then run it with the `java` command to see the result:

```
Test
```

It is definitely better to write a unit test to test the method instead of visually inspecting the generated output, but that's still a lot of coding. It is true that with a good IDE, such a process can be completed more quickly, but don't you wish you were able to write something like the following and run it on the fly?

```
assert "Test_String".substring(0,4) == "Test"
```

1-3. How Does Groovy Address the Shortcomings of Java?

While Java the language is beginning to show its age, Java the platform has a lot of life left in it and will continue to be ubiquitous for many years to come. Groovy's strongest feature is that it compiles to native Java bytecode, which enables Groovy to run natively on the Java platform. This feature also enables Groovy to integrate seamlessly with Java. This is great news for Java developers: you can reuse all of your Java code and use any Java library or framework when working with Groovy. You also don't need to write your entire project in Groovy; you can have some parts written in Java and other parts written in Groovy. As a matter of fact, large parts of Groovy are written in Java (the rest is written in Groovy itself).

Groovy is a great add-on for any Java developer's toolbox because it solves most of the problems with Java that I enumerated in the previous section. For a start, Groovy is succinct. Unlike Java, it's brief, concise, and to the point. Groovy is made concise by leaving out most of the always-required Java syntax elements. Semicolons, type declarations, parentheses, checked exceptions handling, and return statements are all optional in Groovy. In addition, Groovy introduces a helper library called the Groovy Development Kit (GDK) that makes common programming tasks a whole lot easier and less verbose. To

illustrate this, consider the very common task of reading a file. If you want to program it in Java, your code will look like Listing 1-2.

Listing 1-2. *Reading and Printing the Contents of a File in Java*

```
package com.apress.groovygrailsrecipes.chap01;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class SampleFileReader {
  static public String readFile(File file) {
    StringBuffer contents = new StringBuffer();
    try {
      BufferedReader reader = new BufferedReader(new FileReader(file));
      try {
        String line = null;
        while ((line = reader.readLine()) != null) {
          contents.append(line).append
          (System.getProperty("line.separator"));
        }
      } finally {
        reader.close();
      }
    } catch (IOException ex) {
      contents.append(ex.getMessage());
      ex.printStackTrace();
    }
    return contents.toString();
  }

  public static void main(String[] args) {
    File file = new File("C:\\temp\\test.txt");
    System.out.println(SampleFileReader.readFile(file));
  }
}
```

That's about 25 lines of code just to read a file and display its contents to the console output! Now let's see how you can achieve the same task in Groovy with two lines of code. Take a look at Listing 1-3.

Listing 1-3. *Reading and Printing the Contents of a File in Groovy*

```
f = new File("C:\\temp\\test.txt")
f.eachLine{println it}
```

That's it! No unnecessary boilerplate code for catching exceptions, releasing resources, and wrapping readers. Groovy's GDK does all this for you without having to worry about the internals of Java's I/O. This leads to faster development—and easier-to-read, more stable, less error-prone code.

On top of that, the code makes no sacrifices in clarity or readability. Even for someone who has never seen Groovy code before, reading the code in Listing 1-3 makes perfect sense. First you are creating a `File` object, passing the full name of the file you want to read in the constructor, and then you are iterating over each line printing it.

Unlike Java, *everything* in Groovy is an object. There are no primitive types or operators. Numbers, characters, and Booleans in Groovy are Java objects that use Java's wrapper classes. For example, an `int` in Groovy is actually an instance of `java.lang.Integer`. Similarly, operators in Groovy are Java method calls. For example, the operation `3 + 3` in Groovy is executed as `3.plus(3)`, where the first operand is converted to an instance of `Integer` and the second operand is passed as an argument of type `Integer` to the `plus` operation, returning a new `Integer` object of value `6`.

You will appreciate Groovy's model of treating everything as an object when dealing with collections. Collections in Java can work on objects only and not on primitive types. Java 5.0 added support for *autoboxing*—automatic wrapping and unwrapping of objects with their primitive types. In Groovy, no autoboxing is needed because *everything* is an object.

As an example, suppose you want to create three lists: the first list contains the integers from 0 to 9, the second list contains the integers from 1 to 10, and the third list contains the average of the two elements with the same index from the two lists. That is, the third list will contain the floats 0.5, 1.5, 2.5, and so on. The Groovy code to do so is shown in Listing 1-4.

Listing 1-4. *Creating a List That Contains the Averages of Two Other Lists in Groovy*

```
list1 = []; list2 = [];list3 = []
for (element in 0..9){
  list1 += element
  list2 += element + 1
  list3 += (list1[element] + list2[element]) / 2
}
list3.each{
  println it
}
```

There are a few points of interest here. First, because everything in Groovy is an object, no boxing and unboxing is necessary. Second, unlike in Java, division in Groovy produces a `BigDecimal` result if both operands are integers. To perform integer division, you need to cast the result of the division to an `Integer`. Third, the preceding example illustrates Groovy's language-level support for lists; by using syntax close to Java's arrays, Java programmers are made to feel at home when working with lists in Groovy. In Chapters 2 and 3, you will see two more collective data types that Groovy supports at the language level: ranges and maps.

Groovy has many powerful and advanced features that are lacking from the Java language. One of the most important features that Java lacks is *closures*: code blocks that can be treated as objects and passed around as method arguments. The closest thing that Java has to closures is anonymous inner classes, but they have severe limitations: they can be used only once, where they are defined; they can access only static and instance variables of the enclosing outer classes and final method variables; and their syntax is confusing. This might explain why anonymous inner classes are not widely used by Java programmers outside of Swing development. You will learn more about closures in Groovy in Chapter 5.

There are other advanced features in Groovy that have no counterparts in Java. You will learn more about these new features throughout the rest of this book.

Groovy code (like Java) can be organized in classes. Groovy can also be written as scripts. Groovy scripts can be compiled and executed in one step to produce immediate output. This means that you no longer need to write boilerplate code when learning Groovy. For example, the mandatory `Hello World` application can be written as a Groovy script in exactly one line:

```
println "Hello World"
```

1-4. How Do I Download and Install Groovy?

The first step toward learning and using Groovy is to install it. The only prerequisite for using Groovy is having JDK version 1.5 or higher installed on your system (starting with version 1.1-rc-1, Groovy requires JDK version 1.5 or higher and won't run on earlier versions).You also need to have the `JAVA_HOME` environment variable set correctly to point to your JDK installation.

Use the following steps to install Groovy on your computer:

1. Download the latest stable version of Groovy from `http://groovy.codehaus.org/Download`. The latest stable version at the time of this writing is 1.5.4.

2. Groovy comes in different package types tailored to your operating system of choice. You can download a binary release in ZIP format, which is platform independent. You can also download a Windows EXE installer if you are using Windows. If you are using a Debian-based Linux distribution (for example, Ubuntu), you can download and install Groovy in one step with the following command:

   ```
   apt-get install groovy
   ```

   If you do download a platform-specific package, you can skip step 3 because the installer will take care of any postinstallation configuration.

3. If you download the binary release in ZIP format, you need to unzip it first to some location on your file system. You then need to create an environment variable called `GROOVY_HOME` and set it to the location where you unpacked your Groovy distribution. The last step is to add `$GROOVY_HOME/bin` to your `PATH` environment variable.

To test whether Groovy has installed correctly, open a command shell (a command prompt in Windows) and type `groovy -v`. If your installation was successful, you should see a message similar to the following (your Groovy and JDK versions might be different):

```
Groovy Version:1.5.4JVM:1.6.0_03-b05
```

1-5. What Tools Come with Groovy?

Groovy comes with two tools that enable you to write and execute scripts: an interactive shell that enables you to type and run Groovy statements from the command line, and a graphical Swing console. Groovy scripts can also be compiled and executed from the command line by using the commands `groovy` and `groovyc`.

1-6. How Do I Use the Groovy Shell?

To start using the Groovy shell, type `groovysh` at the command line. You should see the following output:

```
Groovy Shell (1.5.4, JVM: 1.6.0_03-b05)
Type 'help' or '\h' for help.
-------------------------------------------
groovy:000>
```

The shell should look familiar to users of `bash` or `tcsh` on Linux. You no longer need to type the `go` command to execute the shell's buffer; a simple return will do it. You can still write multiline expressions, however, because the console is smart enough not to evaluate an expression before it's complete. Here is an example of creating a class that sums all the numbers in a list:

```
groovy:000> class ListSum{
groovy:001>     public static int sum(list){
groovy:002>         def result = 0
groovy:003>         list.each{
groovy:004>             result += it
groovy:005>         }
groovy:006>     return result
groovy:007>    }
groovy:008> }
===> true
groovy:000>
groovy:000> a = [1,2,3,4,5]
===> [1, 2, 3, 4, 5]
groovy:000> println ListSum.sum(a)
15
===> null
```

Typing `\h` at the command line will display the list of commands the shell supports. If you need more help on a particular command, type `help` *`command`*. For example, to get more information on the `inspect` command, type `help inspect`:

```
groovy:000> help inspect
usage: inspect [<variable>]
Opens the GUI object browser to inspect a variable or the result of the evaluation.
```

1-7. How Do I Use the Groovy Console?

As an alternative to the shell, Groovy offers a graphical console that enables you to edit and execute Groovy files (see Figure 1-1). To start the console, type `groovyConsole` at the command line.

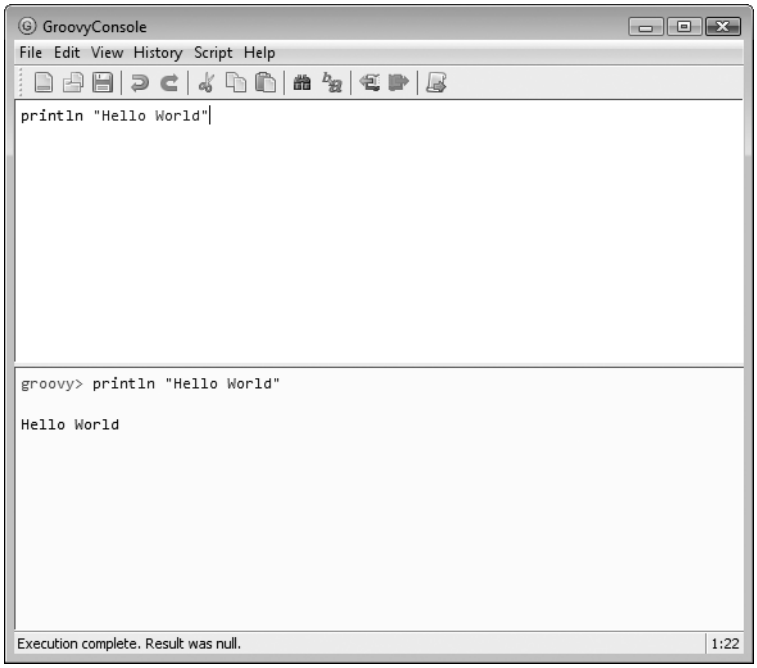

Figure 1-1. *Groovy console showing the editor in the top pane and the output in the bottom pane*

To execute all the code in the console, press Ctrl+R on your keyboard or choose Script ➤ Run from the menu. If you wish to execute a selection of the code, highlight only the code you wish to execute and press Ctrl+Shift+R or choose Script ➤ Run Selection.

You can use the console to edit and save `.groovy` files for later compilation. The console also serves as a great learning tool for experimenting with Groovy because it enables you to see the result of your program instantly, without having to compile and run it in separate steps. Compare this to Java, where any executable class must have a static `main` method and needs to be compiled and executed in two separate steps. It is important to note that Groovy does a lot of work behind the scenes in order to execute your scripts on the fly. Remember that Groovy produces Java bytecode, which has to adhere to the JVM's object model.

1-8. How Do I Use groovyc and groovy?

You can call the Groovy compiler directly on your scripts by using the command `groovyc *.groovy`. This will generate one or more `*.class` files that can be executed with the `java` command. (You need to make sure to have the `groovy-1.5.x.jar` file on your class path when executing a Groovy-generated `.class` file.)

You can also compile and execute Groovy scripts in one step by using the command `groovy *.groovy`. Unlike the `groovyc` command, this won't generate `.class` file(s) on the file system but, rather, the bytecode will be generated in memory and executed immediately.

You might wonder how Groovy can generate executable bytecode from a script that has no `main` method. After all, the bytecode is running on the JVM, so it has to have an executable `main` method somehow. The answer to this is that before compiling your Groovy script, the Groovy compiler will feed it to the Groovy parser, which will generate an abstract syntax tree (AST) out of it in memory. Then the Groovy compiler will compile the AST (which will have an executable `main` method) into Java bytecode. Finally, your bytecode is run in a standard way through an invocation of the `java` command.

It might be helpful to compile a simple Groovy script into Java bytecode and decompile it with a decompiler to see all the code that the Groovy parser generates. You don't need to understand the generated code—which can be overwhelming for beginners—but it helps to appreciate the amount of work that Groovy does to achieve its dynamic nature.

1-9. Is There IDE Support for Groovy?

Most major Java IDEs offer support for Groovy through downloadable plug-ins. In the following two recipes, I cover adding Groovy support to Eclipse and IntelliJ IDEA. Other plug-ins exist for NetBeans, jEdit, Oracle JDeveloper, TextMate, and others. Please check Groovy's documentation web site at `http://groovy.codehaus.org/Documentation` for instructions on adding Groovy support to these IDEs.

1-10. How Do I Integrate Groovy with Eclipse?

The Eclipse IDE can be downloaded for free from `http://www.eclipse.org/downloads` and requires Java 5 JRE or higher to run. If you are using Eclipse version 3.2 or above, you can add the Groovy plug-in by following these steps:

1. From the Help menu, choose Software Updates ➤ Find and Install ➤ Search for new features to install.

2. Click the New Remote Site option and type **Groovy** in the Name field and `http://dist.codehaus.org/groovy/distributions/update` in the URL field.

3. Deselect all the sites to include in the search except for the Groovy site you just added. Click the Finish button. In the search results window, place a check mark next to Groovy and click Next. Accept the terms of the license agreement and click Finish to complete the installation. You will be prompted to restart Eclipse for the plug-in to install correctly.

Upon a restart of Eclipse, you can add Groovy support to an existing Java project by right-clicking on the project and choosing Add Groovy Nature. This does two things to your project: it adds `groovy-all-1.5.x jar` to your class path and creates a `bin-groovy` directory that will hold Groovy's generated class files. If you wish to change the location where Groovy's classes will be generated or to disable generation of Groovy classes altogether, right-click on your project and choose Properties, and then in the left pane click Groovy Project Properties.

To create a new Groovy file, right-click on the package where you want your Groovy file to be created and choose New ➤ Other. In the Filter Text field, type **Groovy** to see two types of Groovy files you can create: Groovy Class and Groovy Unit Test. Choose Groovy Class, give it a name, and click Finish. The Groovy plug-in will provide syntax coloring and autocompletion for your Groovy code, as shown in Figure 1-2.

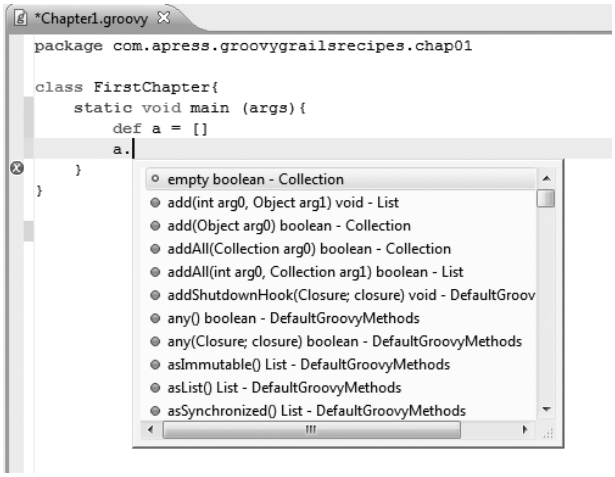

Figure 1-2. *Eclipse Groovy plug-in showing syntax highlighting and code completion*

To compile and execute a Groovy script, right-click in the editor window or on the script name in the Project Explorer, and choose Run As ➤ Groovy, as shown in Figure 1-3. The console window will show the output of your script.

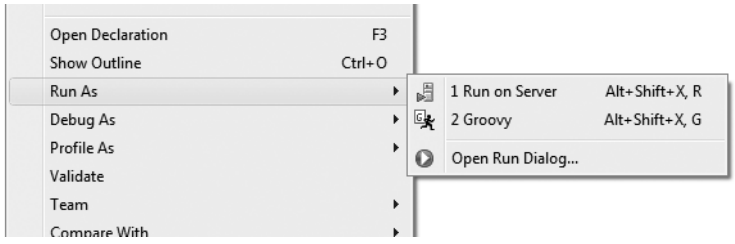

Figure 1-3. *Running a Groovy script in Eclipse*

1-11. How Do I Integrate Groovy with IntelliJ IDEA?

IntelliJ is a commercial Java IDE from JetBrains. A full-featured 30-day trial can be downloaded for free from `http://www.jetbrains.com/idea/download`. If you are using IntelliJ IDEA version 7.0 or higher, you are in luck. JetBrains has added a new plug-in called JetGroovy that adds Groovy and Grails support to IntelliJ. To install, follow these steps:

1. From the File menu, choose Settings ➤ Plugins.
2. Type **Groovy** in the Search field and select the JetGroovy check box. Click the OK button to download and install the plug-in. You will be prompted to restart IntelliJ for the changes to take effect.

To add Groovy support to an existing project, right-click on the project and choose Add Framework support. Select the check box next to Groovy and click OK. You will now see the `groovy-all-1.5.x jar` file added to your class path.

To create a new Groovy class or script, right-click on the `src` folder and choose New ➤ Groovy ➤ Groovy Class or Groovy Script. Like Eclipse's Groovy plug-in, the IntelliJ IDEA Groovy plug-in adds syntax highlighting and code completion to your Groovy files, as shown in Figure 1-4.

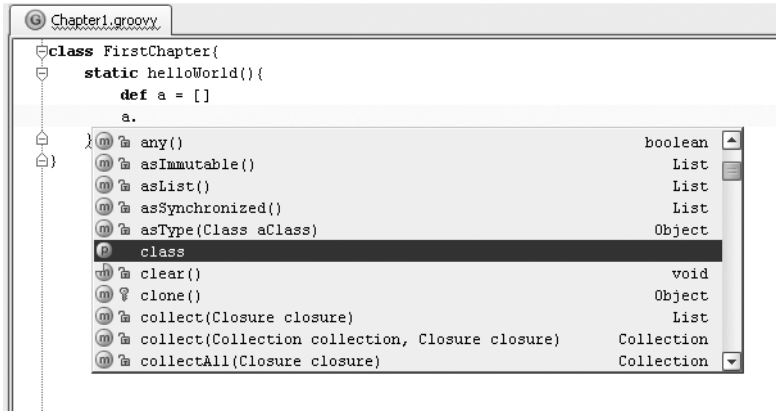

Figure 1-4. *IntelliJ IDEA JetGroovy plug-in showing syntax highlighting and code completion*

To compile a Groovy source file, right-click in the editor window and choose Compile "*ClassName*".groovy. To compile the file and execute it at the same time, choose Run "*ClassName*" from the same menu, as shown in Figure 1-5.

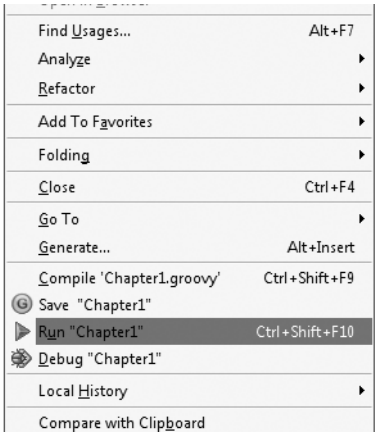

Figure 1-5. *Compiling and executing a Groovy script using the IntelliJ IDEA JetGroovy plug-in*

Summary

This chapter has explained the shortcomings of Java and how Groovy elegantly addresses these issues. After all, why bother learning a new language if there is no added value to it? Now that you have Groovy installed on your machine and integrated with your favorite IDE, you are ready to start the wonderful journey of Groovy. Don't worry if you haven't learned much about Groovy yet; I will cover the language in detail throughout the rest of this book.

Because most people learning Groovy are Java users, and because this book assumes some Java knowledge, the next chapter is dedicated to explaining Groovy to Java developers, illustrating the differences between Java and Groovy, and easing the transition from Java syntax to Groovy syntax.