Fluently Groovy
Java application development using Groovy's simple syntax

Skill Level: Introductory

Andrew Glover (aglover@stelligent.com)
President
Stelligent Incorporated

26 Mar 2008

This tutorial is for Java™ developers unfamiliar with Groovy, who want a quick and easy introduction to the basics. Get started with Groovy's simplified variation of the Java syntax and learn about essential features like native collections, built-in regular expressions, and closures. Write your first Groovy class, and then see how easily you can use JUnit to test it. You'll walk away from this one-hour tutorial with a fully functioning Groovy development environment and the skills to use it. Best of all, you'll have learned first-hand how to use Groovy and Java code together in your everyday Java application development.

Section 1. Before you start

Learn what to expect from this tutorial and how to get the most out of it.

About this tutorial

Groovy is a lot like Java 2.0, if someone set out to completely rewrite the Java language today. Rather than replacing Java, Groovy complements it, providing a simpler, slicker syntax where the type checking is done dynamically at runtime. You can use Groovy to write Java applications on the fly, to glue together Java modules, or even to extend existing Java applications — you can even use Groovy to unit test your Java code. And the beauty of it is, Groovy lets you do all these things faster —
sometimes a lot faster — than you would if you were writing pure Java code.

In this tutorial, you'll get to know Groovy, a dynamic language that sits as comfortably on the Java platform as does the Java language itself.

Objectives

This tutorial guides you step-by-step through the fundamental concepts of Groovy. You will learn about Groovy collections, Groovy classes, and, of course, the Groovy syntax. When you are done with this one-hour tutorial, you will understand the benefits of using Groovy with Java (and vice versa), and you'll be set to start using Groovy in your everyday Java development.

Prerequisites

To get the most from this tutorial, you should be familiar with Java syntax and the basic concepts of object-oriented development on the Java platform.

System requirements

To follow along and try out the code for this tutorial, you need a working installation of either

- Sun's JDK 1.5.0_09 (or later); or
- IBM developer kit for Java technology 1.5.0 SR3

In addition, this tutorial assumes you are using the Eclipse IDE. You do not need to have Groovy installed as the tutorial walks you through the Groovy Eclipse plug-in installation.

The recommended system configuration for this tutorial is as follows:

- A system supporting either the Sun JDK 1.5.0_09 (or later) or the IBM JDK 1.5.0 SR3 with at least 500 MB of main memory
- At least 20 MB of disk space to install the software components and examples covered

The instructions and examples in the tutorial are based on a Microsoft Windows operating system. All the tools covered in the tutorial also work on Linux and Unix systems.
Section 2. About Groovy

In this section you’ll learn the basics of Groovy: what it is, how it relates to the Java language and the JVM, and some of the highlights of writing Groovy code.

What is Groovy?

Groovy is an alternate language for the JVM — alternate meaning that you can use Groovy for Java programming on the Java platform in much the same way you would use Java code. Groovy code combines well with Java code when writing new applications, and can also be used to extend existing ones. Groovy currently is in version 1.5.4 and works on the Java 1.4 and Java 5 platforms, as well as Java 6.

One nice thing about Groovy is that its syntax is very similar to the syntax in the Java language. While Groovy’s syntax was inspired by languages like Smalltalk and Ruby, you can think of it as a simpler, more expressive variation on the Java language. (Ruby is different from Groovy in this regard, because its syntax is quite unlike Java syntax.)

Many Java developers take comfort in the similarity between Groovy code and Java code. From a learning standpoint, if you know how to write Java code, you already kind of know Groovy. The main difference between Groovy and the Java language is that Groovy lets you write less code (sometimes far less!) to accomplish the same tasks you might labor over in your Java code.

Groovy shortcuts

As you begin playing with Groovy, you’ll find that it makes everyday programming activities much quicker. You’ll know a lot about Groovy’s syntactic shortcuts by the time you’re done with this tutorial. For now, just consider these highlights:

- Groovy’s relaxed Java syntax allows you to drop semi-colons and modifiers.
- Everything in Groovy is public unless you state otherwise.
- Groovy permits you to define simple scripts without having to define a formal class object.
- Groovy adds some magical methods and shortcuts on normal everyday Java objects to make them easier to work with.
• Groovy's syntax also permits you to drop a variable's type.

Groovy extras

While Groovy permits you to drop a few elements from Java's syntax, it also adds new features, like native collections, built-in regular expressions, and closures. In normal Java code, if you want to create a list of items, you first import `java.util.ArrayList` (for example), and then programmatically initialize an `ArrayList` instance, and _then_ add items to it. In Groovy, lists and maps are built into the syntax — you don't need to import anything. Regular expressions also don't require additional imports or objects; they are created via a special Groovy syntax.

About closures

Closures are an exciting addition to any Java developer's bag of tricks. While these magical constructs are slated to become part of normal Java syntax in an upcoming Java release (most likely Java 7), they are already available in Groovy today. You can think of a _closure_ as an executable code block that can be defined and then executed at a later point. You can do many neat things with these powerful constructs, though they're most known for making iteration easier. Start using Groovy and it's possible you'll never need to type an instance of `Iterator` again.

Dynamic Groovy

Technically speaking, Groovy is one of those loosely typed, dynamic languages you've probably been hearing so much about lately. In this regard, Groovy is quite different from the Java language, which is a statically typed language. In Groovy, types are optional, so you don't have to type `String myStr = "Hello";` to declare a `String` variable.

What's more, Groovy code can alter itself at runtime quite easily. This essentially means that, at runtime, objects can easily be endowed with new methods and properties. This entire realm of programming, known as _metaprogramming_, is well supported in Groovy. You'll have the chance to learn more about Groovy's dynamic nature as you progress through this tutorial. For now, suffice it to say that you'll be surprised by how easily Groovy facilitates working with XML or normal `java.io.File` instances.

Two of a kind

Anything you write in Groovy can be compiled into a normal Java class file and re-used in Java code. Likewise, anything you write in normal Java code can be reused in Groovy. As a result, you can easily use Groovy to write unit tests for Java
code, for instance. And if you write a handy utility in Groovy, you can also use that utility in your Java programs.

Section 3. The case for Groovy

Learning a new language is no small undertaking, even if it is Groovy. In this section, discover more about incentives for learning Groovy. Additionally, get your first look at some Groovy code and see how it compares to Java programming.

Why learn Groovy?

Even with all of Groovy's similarities to the Java language, it is a different language. You might be wondering why you should take the time to learn it. The short answer is that Groovy is a more productive language. It offers a relaxed syntax with some special features that enable you to code things more quickly.

Just to take one example, once you see how easy it is to navigate collections using Groovy, you'll never work with them in Java again. Being able to code quickly in Groovy also means receiving feedback sooner, not to mention the satisfaction of crossing tasks off of your to-do list. At a high level, if you can put code in front of stake-holders more quickly, you can give them more releases in a shorter time. In essence, Groovy lends itself more to Agile development than Java does.

Getting started is easy

If the idea of adopting a new language is still daunting, consider how easy it is to integrate Groovy with your development environment. You don't need to install a new runtime utility or specialized IDE. In fact, you're just one jar away from having Groovy on your classpath.

Also, Groovy is an open source language managed by a community of passionate Java developers. Because Groovy is licensed under the Apache Software License, Version 2.0, you are free to use it for the development of free, as well as proprietary, software.

Groovy versus the Java language

You wouldn't buy a motorcycle without taking it for a test ride, so before I ask you to
install Groovy, I'll show you some code. First, I give you a quick reminder of what it is like to create, compile, and run a standard Hello World example in Java; then you see the same procedure in Groovy code. Comparing the two examples makes it easy to appreciate the difference between the two languages.

Hello World in Java code

The prototypical Hello World example in Java code looks something like this:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Compile and run the Java example

I've omitted a package for this simple `HelloWorld` class and tried not to use any superfluous coding conventions for printing to the console. The next step is to compile this class via `javac`, as follows:

```bash
c:>javac HelloWorld.java
```

And finally, I run the resulting compiled class:

```bash
c:>java HelloWorld
```

So far so good — you passed Java 101 conceivably long ago, so this was just a refresher. Next, see the same process in Groovy.

Hello World in Groovy code

As I mentioned earlier, Groovy permits a relaxed Java syntax — for instance, you don't need to define classes for simple actions like printing "Hello World!"

What's more, Groovy makes everyday coding activities easier; for instance, Groovy permits you to type `println` instead of `System.out.println`. Groovy is smart enough to know you mean `System.out` when you type `println`.

Consequently, writing the Hello World program in Groovy is as simple as this:

```groovy
println "Hello World!"
```
Note that there is no class structure around the phrase. There isn't a method structure either! I also used `println` instead of `System.out.println`.

**Run the Groovy example**

Assuming I've saved my code into a file called `MyFirstExample.groovy`, I can run this example by simply typing

```
c:>groovy MyFirstExample.groovy
```

That's all it takes to get the words "Hello World!" printed out on my console.

**Shortcuts in action**

You might notice that I didn't have to compile the `.groovy` file. That's because Groovy belongs to the family of languages known as *scripting languages*. One of the defining qualities of scripting languages is that they can be interpreted at runtime. (In Java, bytecode is also interpreted and generated as a result of compiling source code. The difference is that scripting languages interpret source code *directly*.)

Groovy permits you to drop the compilation step entirely, though you *could* do it, if you wanted to, using the Groovy compiler, `groovyc`. Compiling Groovy code with `groovyc` yields normal Java bytecode that I can then run via the `java` command. This is a key aspect of Groovy that is often overlooked: Anything you write in Groovy can be compiled and run via the normal Java runtime.

As for running the code, if I wanted to be even more terse I could type

```
c:>groovy -e "println 'Hello World!'"
```

That would achieve the same result without any file definition at all!

---

**Section 4. Getting started with Groovy**

In this section you actually get started with some Groovy programming. First, learn how easy it is to install Groovy (via the Eclipse Groovy plugin), and then start on some easy examples that will help you get to know Groovy.
Groovy's easy install

To get going with Groovy quickly, all you need to do is install Eclipse's Groovy plugin. Open Eclipse and select the Help menu item > Software Updates > Find and Install....

Figure 1 shows the dialog box you are presented with after you did the steps above:

**Figure 1. Eclipse Feature Updates dialog**

Navigate the options

Next, you are presented with a dialog box containing two choices. Select the radio button that says **Search for new features to install**. Click the Next button and then select **New Remote Site**.... You are presented with a new dialog box, which has two fields to fill in: the name of the new location and the URL for that location, as Figure 2 shows:

**Figure 2. Make sure you provide a valid URL for your new remote site**
Type "Groovy plugin" for the name and "http://dist.codehaus.org/groovy/distributions/update/" for the location. Hit OK, then ensure that the item named "Groovy plugin" is selected in the Sites to include in search box that follows — your list should look something like what you see in Figure 3.

Figure 3. Remote sites listing in Eclipse
Wrap it up

After you've clicked the **Finish** button, you should be presented with a Search Results dialog box. Ensure once again that you've selected the "Groovy plugin" box and hit the **Next** button. Figure 4 shows this step:

**Figure 4. Selecting the Groovy plugin**
After you confirm a number of things, you'll download the plugin, after which you may have to restart Eclipse.

Creating a Groovy project

Once Eclipse has restarted, you can create your first Groovy project. Be sure to create two source folders — one called "groovy" and another called "java." The Groovy code you write goes in the groovy folder and the Java code goes in the java folder. I find it helpful to separate the two, as Figure 5 illustrates:

**Figure 5. Two source folders Java and Groovy**
Importing Groovy to your project

Once the project has been created, right-click its icon and you should see a Groovy option as Figure 6 shows. Select that option and then select the Add Groovy Nature option. Doing so imports the necessary Groovy libraries, compiler, and runner to your project.
Figure 6. Adding the Groovy nature in Eclipse

Creating a Groovy class

Creating a Groovy class is simple. Select the groovy folder and right-click it. Select New and then Other as Figure 7 demonstrates:

Figure 7. Creating a Groovy class via the New menu
Name your class

From there, find the Groovy folder and select **Groovy Class** — you should see a dialog box like the one in Figure 8.
Click the **Next** button and you'll be asked to give the class a name. Type **HelloWorld**.

For now, you can leave the **HelloWorld** Groovy class in the default package, as Figure 9 shows.

**Figure 9. Don't worry about packages for now!**
While this may seem like a lot of steps, it isn't all that different from creating a normal Java class.

Hello World! A Java program written in Groovy

Upon clicking the Finish button you should be presented with a code snip like this:

```java
class HelloWorld {
    static void main(args) {
    }
}
```
That looks an awful lot like the Java HelloWorld example from earlier. Note, though, that it contains no public modifiers. Also, if you look closely at the argument to the main method, note that there isn't a type.

Compile the program

Now go ahead and put a println "Hello World" inside of the main method so it looks like this:

```groovy
class HelloWorld {
    static void main(args) {
        println "Hello World"
    }
}
```

You should be able to right-click within the source code editor and select the Compile Groovy File option, as Figure 10 shows.

Figure 10. Right clicking on the Groovy file enables compilation
Run the program

Next, right-click again, and select the Run As option, and then select the Groovy option. You should see "Hello World" printed in your Eclipse console, as Figure 11 shows.

Figure 11. Hello World was never so pleasing
What have you learned?

Okay, so that was a sneaky way to make an important point. Groovy really is Java. The syntax is different — shorter, for the most part — but Groovy code is 100 percent compliant with Java bytecode. The intermingling of these two languages is explored further in the next section.

Section 5. Groovy becomes Java

You've seen your first proof that Groovy code is practically interchangeable with Java code. This section drives this point home as you continue to play with your Groovy-built HelloWorld class.

Hello, Java!

Just to convince yourself that Groovy is Java, go ahead and put public modifiers in front of the HelloWorld class declaration and the method declaration, like this:

```java
public class HelloWorld {
    public static void main(String[] args) {
        println "Hello World"
    }
}
```

Still not convinced?

There's no reason this code shouldn't run like it did before. But, if you still aren't convinced, try putting a String[] before the args parameter:

```java
public class HelloWorld {
    public static void main(String[] args) {
        println "Hello World"
    }
}
```
Don’t stop now

You’ve come this far, so you might as well replace `println` with `System.out.println` — and don’t forget to add parentheses for good measure.

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World")
    }
}
```

Now you’re back to where you started with the original Hello World example written in Java code. The question is, which example was easier to write?

Note how the original Groovy-based `HelloWorld` class lacked any `public` modifiers, any types (no `String[]`), and offered the shortcut `println` without parentheses to boot.

Hello, Groovy!

If you want to, you can turn this process inside out. Just go back to the Java-based Hello World example, remove `everything` in the file except the `System.out` line and then, while you’re at it, remove the `System.out` and the parentheses. Here’s what you’re left with:

```java
println "Hello World"
```

Now, which program is easier to write?

Run it!

Groovy code is 100 percent compliant with Java bytecode, as this exercise proves. Inside Eclipse, select the Run menu option Open Run Dialog.... Select a new Java Application configuration. Make sure the project is your Groovy project. For the Main class, hit the Search button and find your `HelloWorld` class. Note that word class, which implies that the Eclipse Groovy plugin has compiled your `.groovy` file into a `.class` file.

You can see this entire process in action below in Figure 12 — it should look familiar to you if you’ve run Java classes before in Eclipse.

**Figure 12. Groovy code is 100 percent compliant with Java bytecode**
Hit the Run button and what do you see? Indeed, "Hello World!" has never been so revealing.

Section 6. Groovy is Java code without types

It's tempting to say that Groovy is just Java code without the rules, but in fact, it just has fewer rules. This section focuses on one of the concrete aspects of Java programming you leave behind when you start writing your Java applications using Groovy: type definitions.

Why type types?
In Java, if you want to declare a `String` variable, you have to type:

```java
String value = "Hello World";
```

If you think about it, though, the characters to the right of the equals sign already imply that `String` is the type of the variable `value`. Accordingly, Groovy permits you to drop the `String` type variable in front of `value` and replace it with `def`.

```
def value = "Hello World"
```

In essence, Groovy infers an object's type by its value.

**Run it!**

Go ahead and edit your HelloWorld.groovy file to look like this:

```groovy
String message = "Hello World"
println message
```

Run this code and you should see the same old "Hello World" on the console. Now, replace the `String` variable type with `def` and re-run the code. Do you notice the same thing?

Rather than printing the value of `message`, you can print its type with the following call:

```
def message = "Hello World"
println message.class
```

Printing "class java.lang.String" should be a welcome change by now! If you're wondering what happened, well, Groovy figured out that `message` had to be of type `String` because its value is surrounded with quotes.

**More about type inference**

You may have heard that in Groovy "everything is an object" — but what does that mean for types? Well, let's see what happens if you replace the `String` from the previous example with a number, like so:

```
def message = 12
println message.class
```
The numerical value for the `message` variable looks like Java’s primitive `int` type. Running this code reveals that Groovy makes it an `Integer`, however. That’s because "everything is an object" in Groovy, remember?

Every object in Java extends `java.lang.Object`, which is mighty convenient for Groovy. Even in the worst-case scenario where the Groovy runtime can’t ascertain a variable’s type, all it has to do is make it an `Object` and things work out.

Go ahead and play around with this code. Make `message` whatever you’d like it to be: Groovy will do its best to infer its type at runtime.

The implications of being typeless

So, Groovy’s lack of types means less typing. Admittedly, replacing `String` with `def` doesn’t really save you a lot of keystrokes — three is hardly anything to brag about! But at a higher level, over the course of writing a lot of code with more than just variable declarations, lack of types actually does add up to a lot less typing.

More importantly, it means a lot less code to read. Last but not least, the lack of types in Groovy opens up a whole new world of flexibility — flexibility that doesn’t require interfaces or abstract classes, by the way.

See, you only need to use the `def` keyword when declaring a freestanding variable in a method; you don’t need the `def` keyword for parameters in method declarations. You also don’t need them in, say, a `for` loop declaration, which means you don’t need to write `(int x = 0; x < 5; x++)`. Instead, you can drop the `int` and leave it blank.

Section 7. Looping through Groovy

Like most scripting languages, Groovy is often touted as a *more productive* alternative to the Java language. You’ve already seen how Groovy’s lack of types cuts down on typing. In this section, you create and then experiment with a `repeat` function. In the process, you explore more of the ways Groovy trades in verbosity for efficiency.

The better, shorter for loop

Here’s a fun way to appreciate Groovy’s lack of types: First, create a new Groovy
class the same way you created HelloWorld. Call this class MethodMadness and remove the auto-generated class body: you’re going to define a freestanding repeat function instead. Now type this into your console:

```java
def repeat(val){
    for(i = 0; i < 5; i++){
        println val
    }
}
```

At first, this little function may look strange to your Java-trained eyes (in fact, it looks a lot like JavaScript). But what you’re seeing is Java code; it’s just written Groovy-style.

**Inside MethodMadness**

The repeat function takes one variable, val. Note how the parameter doesn’t require a def. The body of the method is, essentially, a for loop.

Calling this function

```java
repeat("hello world")
```

prints "hello world" five times. Note how the for loop allowed me to drop the int. A for loop without a variable type is a bit shorter than the usual Java fare. Now see what happens when you throw ranges into the mix.

**Ranges in Groovy**

A range is a sequence of values. For example, "0..4" denotes the inclusive integers 0, 1, 2, 3, 4. Groovy also supports exclusive ranges, where "0..<4" means 0, 1, 2, 3. You can also create a range of characters: "a..e" is equal to a, b, c, d, e. "a..<e" would be all those values less e.

**Ranges for looping**

Ranges facilitate looping quite nicely. For instance, your previous for loop incremented an integer from 0 to 4 like so:

```java
for(i = 0; i < 5; i++)
```

A range would make that for loop cleaner and nicer to read:
def repeat(val){
    for(i in 0..5){
        println val
    }
}

**Setting up a range**

If you try running this example you may notice a small problem: "Hello World" prints six times instead of five. There are three ways to fix this:

- Limit the inclusive range to 4:

```groovy
for(i in 0..4)
```

- Start from 1 instead of 0:

```groovy
def repeat(val){
    for(i in 1..5){
        println val
    }
}
```

- Change the range from inclusive to exclusive:

```groovy
def repeat(val){
    for(i in 0..<5){
        println val
    }
}
```

Any way you cut it, you’re back to where you started — printing "Hello World" five times.

**Default parameter values**

Thus far, you’ve managed to shorten the `repeat` function by using Groovy's range expression. The function is still a bit limiting, though. What if you wanted to repeat "Hello World" eight times? In fact, what if you wanted to repeat certain values differently — say "Hello World" eight times but "Goodbye Sunshine" only twice?

Having to specify the number of desired repetitions every time you make a call to `repeat` can get old, especially if you are already comfortable with the default behavior (that is, repeating five times).

Groovy supports *default parameter values* which allow you to specify a parameter’s default value in the formal definition of a function or method. Callers to the function
can opt to omit the parameter and accept the default value.

**More complicated parameter values**

Using the `repeat` function from earlier, if you want to provide the option to allow callers to specify a repeat value, you can code it as follows:

```groovy
def repeat(val, repeat=5){
    for(i in 0..<repeat){
        println val
    }
}
```

Calling the function, as follows

```groovy
repeat("Hello World", 2)
repeat("Goodbye sunshine", 4)
repeat("foo")
```

results in "Hello World" being printed two times, "Goodbye sunshine" four times, and "foo" the default amount of five times.

---

**Section 8. Groovy collections**

Of all the handy shortcuts and features that Groovy offers, probably one of the most helpful is its *built in* collections. Think for a second about how you use collections in Java programming — import a `java.util` class here, initialize there, and add some items to it. Those three steps alone add up to a more than a few lines of code.

Groovy, on the other hand, facilitates using collections directly within the language. In Groovy you do not need to import a specialized class, nor do you need to initialize an object. A collection is a native member of the language itself. Groovy also makes working with collections (or lists, if you like) quite easy by intuitively aiding in adding and removing items.

**A range can be a collection**

In the previous section you learned how to use Groovy's ranges to make looping easier. The range expression "0..4" represents a *collection* of digits — 0, 1, 2, 3, and 4. To test this out, create a new class and call it `Ranger`. Keep the class definition and the `main` method declaration. But this time, add the following:
def range = 0..4
println range.class
assert range instanceof List

Note how the assert phrase proves that ranges are instances of java.util.List. Go ahead and run this code to verify the range is now a collection of type List.

### Extensive support

Groovy’s collections support is extensive, and the beauty of it is that underneath the magic, everything is a normal Java object. Every Groovy collection is an instance of java.util.Collection or java.util.Map.

As mentioned earlier, Groovy’s syntax offers native lists and maps. For example, try adding the following two lines of code to the Ranger class:

```groovy
def coll = ["Groovy", "Java", "Ruby"]
assert coll instanceof Collection
assert coll instanceof ArrayList
```

You’ll note that the `coll` object looks like an array in the Java language. Actually, it’s a Collection. To achieve the same instance of a collection in normal Java code, you’d have to do something like this:

```java
Collection<String> coll = new ArrayList<String>();
coll.add("Groovy");
coll.add("Java");
coll.add("Ruby");
```

In Java code, you have to use the `add()` method to add items to the instance of the `ArrayList`.

### Add it up

Groovy gives you a number of ways to add something to a list of items — you can use the `add()` method (because the underlying collection is a normal `ArrayList` type), but there are a number of shortcuts you could also try.

For example, each line in the following code adds something to the underlying collection:

```groovy
coll.add("Python")
```
coll << "Smalltalk"
coll[5] = "Perl"

Note how Groovy enables operator overloading — the \(<\) operator is overloaded to permit adding items into a collection. Also, you can directly add items via a positional argument. In this case, as the collection only had four items in it, the \([5]\) operator places "Perl" in the last spot. Go ahead and print out the collection and see it for yourself.

Retrieving is easy

If you need to obtain a particular item from a collection, you can grab it via a positional argument like the one above. For example, if you wanted to obtain the second item, "Java," you could write something like this (remember that collections and arrays are zero-based):

```
assert coll[1] == "Java"
```

Groovy also permits you to add and subtract collections from each other, like so:

```
def numbers = [1,2,3,4]
assert numbers + 5 == [1,2,3,4,5]
assert numbers - [2,3] == [1,4]
```

Note that in the above snip you’ve created a new collection instance, as the last line in the code implies.

Magic methods to boot

Groovy also adds a few other handy features to collections. For instance, you can take instances of collections and call specialized methods on them, like so:

```
def numbers = [1,2,3,4]
assert numbers.join(",",:) == "1,2,3,4"
assert [1,2,3,4,3].count(3) == 2
```

\(\) and count(\) are just two of the many different convenience methods available to call on any list of items. The spread operator is a particularly handy utility which facilitates calling a method on each item in a collection without having to iterate over the collection.

Given a list of Strings, if you’d like to capitalize all of them, you can write
assert ["JAVA", "GROOVY"] ==
["Java", "Groovy"]').toUpperCase()

Note the `*`. notation. For each value in the above list, `toUpperCase()` is invoked, producing a collection with each `String` instance capitalized.

---

**Section 9. Groovy maps**

In addition to its extensive list-handling capability, Groovy also provides a sturdy mapping mechanism. Just like lists, maps are native data structures. And as always, any mapping mechanism in Groovy is an instance of `java.util.Map` under the hood.

**Mapping in the Java language**

Maps in the Java language are collections of name-value pairs. So, to create a typical map in your Java code, you have to do something like this:

```java
Map<String, String> map = new HashMap<String, String>();
map.put("name", "Andy");
map.put("VPN-#", "45");
```

A `HashMap` instance is holding two name-value pairs, each being an instance of a `String`.

**Mapping via Groovy**

Groovy makes working with maps as easy as working with lists — for instance, you could write the Java-based map above in Groovy as

```groovy
def hash = [name:"Andy", VPN-#:45]
```

Note that the keys in a Groovy map don't have to be `Strings`. In this case, `name` looks like a variable, but behind the scenes Groovy will make it a `String`.

**It's all Java**
Now go ahead and create a new class called `Mapper` and add that code. Then add the following line just to verify you're working with real Java code underneath:

```java
assert hash.getClass() == java.util.LinkedHashMap
```

You can see that Groovy makes use of Java's `LinkedHashMap` type, which means you can put and get items from this hash using normal Java idioms.

```groovy
hash.put("id", 23)
assert hash.get("name") == "Andy"
```

Maps are so groovy

Hopefully you've seen by now that Groovy adds its own magic to any equation; consequently, you can put items into a map using the . notation. If you wanted to add a new name-value pair to the map (say `dob` and "01/29/76") you could do it, like so:

```groovy
hash.dob = "01/29/76"
```

The . notation also works for obtaining items. For instance, here's how to grab the value of `dob`:

```groovy
assert hash.dob == "01/29/76"
```

Certainly the . is more groovy than calling `get()`, don't you think?

Positional maps

You can also use pseudo-positional syntax to put items into a map and obtain items from one, as follows:

```groovy
assert hash["name"] == "Andy"
hash["gender"] = "male"
assert hash.gender == "male"
assert hash["gender"] == "male"
```

Note, though, that when obtaining items from a map via the [] syntax, you must reference items as `Strings`. 
Section 10. Closures in Groovy

Closures are a big topic in the Java world right now, with their likely inclusion in Java 7 still being hotly debated. Some have asked why we need closures in the Java language when they already exist in Groovy. In this section, learn about closures in Groovy. If nothing else, what you learn here will come in handy once closures become a formal part of the Java language syntax.

No more Iterator

While you did quite a bit of coding with collections in the previous sections, you've yet to actually iterate over one. Of course, you know that Groovy is Java, so you can always grab an instance of the old Java Iterator, if you want, and loop over a collection like so:

```groovy
def acoll = ["Groovy", "Java", "Ruby"]
for(Iterator iter = acoll.iterator(); iter.hasNext();)
    println iter.next()
```

You don't actually need the type declaration inside the for loop, though, because Groovy has made iteration a direct member of any collection. In this case, rather than having to obtain an Iterator and directly manipulate it, you can just iterate over a collection directly. What's more, the behavior that is normally intended to reside inside a loop construct (such as the println in the body of for loop) is then put inside a closure. Before we get too far, let's see this in action.

Can you spot the closure?

Taking the same exact code from above, you can iterate over a collection much more concisely, like this:

```groovy
def acoll = ["Groovy", "Java", "Ruby"]
acoll.each{
    println it
}
```

Note how each is called directly on the acoll instance, which you already know is of type ArrayList. After the each call, a new syntax is introduced — {, and then
some code, followed by `}. The block of code signified by the `{}` is what's known as a closure.

**Execute it**

Closures are blocks of executable code. They don’t require names, and they can be executed after they’ve been defined. So, in the case above, the nameless closure whose body has the behavior of printing `it` (I’ll explain what `it` is shortly) is called for every value in the `acoll` collection type.

At a high level, the code between the `{}` is executed three times, thus printing what you see in Figure 13.

**Figure 13. Iteration has never been so easy**

The `it` variable inside the closure is a keyword that points to the individual value of the outside collection being invoked — it is a default value that can easily be overridden by passing a parameter into the closure. So for instance, the following code does the same exact thing, but uses its own item variable:

```java
def acoll = ["Groovy", "Java", "Ruby"]
acoll.each{ value ->
    println value
}
```

In this case, `value` is used instead of Groovy's default `it`.

**Ubiquitously iterating**

Closures appear frequently in Groovy; however, you’ll use them most often when iterating over a series of values. Keep in mind, a series of values can be represented in more ways than just as a list — for example you can iterate over a map, a `String`, a JDBC `Rowset`, a line in a `File`, and more.
If you wanted to iterate over the hash object from the previous section of mapping in Groovy, you could write the following:

```groovy
def hash = [name:"Andy", "VPN-#":45]
hash.each{ key, value ->
    println "${key} : ${value}"
}
```

Note how closures also permit multiple parameters — in this case, the code above contains two parameters (key and value).

**Iteration in Java code**

Just as a friendly reminder, here's how you would do the same iteration using typical Java constructs:

```java
Map<String, String>map = new HashMap<String, String>();
map.put("name", "Andy");
map.put("VPN-#", "45");

for(Iterator iter = map.entrySet().iterator(); iter.hasNext();){
    Map.Entry entry = (Map.Entry)iter.next();
    System.out.println(entry.getKey() + " : " + entry.getValue());
}
```

The above code is verbose compared to Groovy's, isn't it? If you find yourself working a lot with collections, it just makes sense to do it in Groovy.

**Total iteration**

Remember, anything that is a collection or series of some sort can be iterated as shown in the code below.

```groovy
"ITERATION".each{
    println it.toLowerCase()
}
```

**More ways to use closures**

While you'll probably find yourself using closures most often for iteration, they do have other uses. Because a closure is a block of code, closures can be passed around as parameters (which you can't do with functions or methods in Groovy). The fact that closures are executed when called (rather than during definition) makes
them quite useful for particular scenarios.

For instance, create a ClosureExample object via Eclipse and leave in the default class syntax it gives you. In the resulting main() method, add this bit of code:

```groovy
def excite = { word ->
    return "${word}!!"
}
```

This code is a closure named excite. This closure takes one parameter (named word) and returns a String with the word variable along with two exclamation points. Note the use of substitution within the String instance. Using the ${value} syntax within a String tells Groovy to replace the value of a particular variable within the String itself. Think of this syntax as a handy way to do the equivalent of return word + "!!".

**Delayed execution**

Now that you have a closure, it's time to actually use it. You can call closures one of two ways: either directly or via the call() method.

Using your ClosureExample class, go ahead and add the following two lines of code below the closure’s definition:

```groovy
assert "Groovy!!" == excite("Groovy")
assert "Java!!" == excite.call("Java")
```

As you can see, either style of invocation works, though calling it directly is more concise. Don't forget that closures are first-class objects in Groovy too — they can be passed around as parameters and of course, executed at some later point. Replicating the same behavior in normal Java is possible, but not exactly easy. But that's no surprise to you by now, right?

---

**Section 11. Classes in Groovy**

Thus far, you've printed "Hello World" numerous times via Groovy, you've worked with collections, you've iterated over collections using closures, and you've even defined a closure on your own. You have managed to do all of this without discussion of a concept that is fundamental to Java developers — classes.
Of course, you've been working with classes for a while in this tutorial already: the last few examples you coded were in the main() method of various classes. And, as you already know, you can define classes in Groovy just like you do in your Java code. The only difference is, you don't need to put in public modifiers and you can drop types for method parameters. This section introduces you to all the other things you can do with Groovy classes.

The Song class

Start by defining a simple, JavaBeans-like class in Groovy called Song.

The first step, naturally, is to create a class in Groovy called Song. This time, create a package structure for it too — go ahead and create a package name like org.acme.groovy.

Once you've created this class, remove main() that was automatically generated for you by the Groovy plugin.

A song has a few properties — the artist who created it, the name of the song, and a genre, to name a few. Add those properties to your newly created Song class, like so:

```groovy
package org.acme.groovy

class Song {
    def name
    def artist
    def genre
}
```

So far so good, right? It's nothing too complex for the new Groovy developer!

A Groovy class is a Java class

You should recall from earlier in this tutorial that the Groovy compiler creates a normal Java .class for every class you define in Groovy. Remember how you defined a HelloWorld class in Groovy and found the .class and ran it? You could do the same with your newly defined Song class. If you do compile the code via Groovy's groovyc compiler (which, by the way, the Eclipse Groovy plugin is already doing for you), a Song .class file is generated.

All this means that if you want to use your new Song class in another Groovy class or Java class, you'll have to import it (unless, of course, the code that will use Song resides in the same package as Song).
Accordingly, go ahead and create a new class, called `SongExample` and put it into another package structure, say `org.thirdparty.lib`.

You should now be looking at some code that looks like this:

```java
package org.thirdparty.lib

class SongExample {
    static void main(args) {}
}
```

### Class relationships

Now it's time to use that `Song` class. First import the instance and add this code into `SongExample`'s `main()` method.

```java
package org.thirdparty.lib
import org.acme.groovy.Song

class SongExample {
    static void main(args) {
        def sng = new Song(name:"Le Freak", artist:"Chic", genre:"Disco")
    }
}
```

Now that's a `Song` instance you can dance to! Look closely, though, at the initialization of the `Song` class you defined earlier. Do you notice anything special? You should notice the auto-generated constructor.

### Class initialization

Groovy automatically gives you a constructor that takes a map of name-value pairs, which correspond to class properties. This, by the way, is an out-of-the-box feature of Groovy — for any properties you define in a class, Groovy permits you to pass a map full of values into a constructor. The use of a map does have implications; for instance, you don't actually have to initialize every property of an object.

You can just as well add this code, too:

```java
def sng2 = new Song(name:"Kung Fu Fighting", genre:"Disco")
```

You are also free to directly manipulate properties of a class, like so:
def sng3 = new Song()
sng3.name = "Funkytown"
sng3.artist = "Lipps Inc."
sng3.setGenre("Disco")
assert sng3.getArtist() == "Lipps Inc."

Looking at this code, it is evident that not only did Groovy create a constructor that permits passing in a map of properties and their values, but you can semi-directly access properties via the . syntax. Moreover, Groovy also generated normal setter and getter methods.

What's particularly groovy about Groovy, when it comes to property manipulation, is that setters and getters are always called — even if you access a property directly via the . syntax.

**Flexibility at the core**

Groovy is an inherently flexible language. For instance, see what happens when you drop the parentheses from the setGenre() method invocation from the previous panel; like so:

```groovy
sng3.setGenre "Disco"
assert sng3.genre == "Disco"
```

In Groovy, for methods that take parameters, you can optionally drop the parentheses — in some ways, doing so makes the code a bit easier to read.

**Method overriding**

So far you've managed to create a few instances of the Song class. Although, right now they don't do anything particularly interesting. You can print one using the following command:

```groovy
println sng3
```

All that does is print the default toString() implementation of all objects in Java, namely the class and it's hashcode (that is, org.acme.groovy.Song@44f787). Let's see what happens when you override the default toString() implementation to print something a bit nicer.

In your Song class, add the following code:
Based on what you've learned in this tutorial, you were able to drop the `public` modifier on the `toString()` method. You still needed to specify a return type (`String`) in order to actually override the proper method. The body of the method is tersely defined — but where is the `return` statement?

No return needed

You've probably already figured out that in Groovy you can omit the `return` statement. Groovy assumes the last line in a method should be returned. So in this case, a `String` is returned containing the values of the class's properties.

Go ahead and re-run the `SongExample` class. You should see something a bit more interesting when you do. The `toString()` method returns a description instead of a hashcode.

Specialized access

Groovy's auto-generation is handy for a few features, but sometimes you need to override default behavior. For instance, let's say you need to override the `getGenre()` method in the `Song` class, so that the returned `String` is all capitalized.

Providing this new behavior is quite easy, just define a `getGenre()` method. You can have either have the method's declaration return a `String` or you can omit it entirely if you wish. Go ahead and do the simplest possible thing:

```groovy
def getGenre(){
    genre.toUpperCase()
}
```

Like before, your simple method omits a return type and a `return` phrase. Now run your `SongExample` class again. You should see something unexpected ---there is now a nullpointer exception.

Nullpointer safety

If you've followed along so far, you should at some point have added this line to your `SongExample` class:

```groovy
String toString(){
    "${name}, ${artist}, ${genre}"
}
```
assert sng3.genre == "Disco"

The result was an assertion failure when you re-ran SongExample — which is why all that ugly red text printed out in your Eclipse console. (Sorry about pulling such a dirty trick, by the way.)

Fortunately, you can easily fix the error: just add the following line of code to the SongExample class:

```
println sng2.artist.toUpperCase()
```

But wait, now more red text is streaming down your console — what happened?!

**Nasty nulls**

If you remember, the sng2 instance didn't define an artist value. As a result, a Nullpointer exception was generated when you invoked the `toUpperCase()` method.

Luckily, Groovy provides a safety net, via the `?` operator — preceding a method call with `?` is like putting a conditional in front of the call which guards calling a method on a null object.

For example, replace the line `sng2.artist.toUpperCase()` with `sng2.artist?.toUpperCase()`. Note that you are also free to omit the trailing parentheses in this case. (Groovy actually permits you to drop parentheses on methods without parameters, too. Doing so can cause issues if Groovy thinks you are accessing a class' properties rather than a method, however.)

Re-running the SongExample class now demonstrates the `?` operator's usefulness. In this case, you aren't presented with a nasty exception. Now go ahead and place this code in the same class and re-run the whole song.

```
def sng4 = new Song(name:"Thriller", artist:"Michael Jackson")
println sng4
```

**Just being Java**

You'll notice that no exception was generated, though you may have expected one. Even though you didn’t define a genre, the `getGenre()` method invoked `toUpperCase()`.
You remember that Groovy is Java, right? So in Song's toString() s you referenced the genre property itself; hence, getGenre() wouldn't be called. Go ahead and change the toString() method to use getGenre(), and then see what happens.

```java
String toString(){
  "${name}, ${artist}, ${getGenre()}
}
```

Re-running SongExample reveals a similar exception. Now, see what happens when you try fixing it on your own.

Another handy little operator

Hopefully, your fix is similar to mine. As you can see below, I augmented the getGenre() method in the Song class to make use of Groovy's handy ? operator.

```java
def getGenre(){
  genre?.toUpperCase()
}
```

The ? operator is quite useful from time to time and most certainly cuts down on conditional phrases.

Section 12. Unit testing Groovy

This tutorial has emphasized the fact that Groovy is just another variation of Java. You've even seen that you can code normal Java using Groovy and get away with it. To drive this point home one last time, you conclude this tutorial by testing your Song class with JUnit using Java.

Add JUnit to your Eclipse project

In order to follow the examples in this section, you need to add JUnit to your Eclipse project. First, right-click on your project and select Build Path and then select Add Libraries as Figure 14 shows:

**Figure 14. Adding JUnit to a project's build path**
You are presented with the **Add Library** dialog box like the one in Figure 15.

**Figure 15. Select JUnit from the list of libraries**
Select JUnit and hit the **Next** button. You should see a dialog box like the one in Figure 16. Select **JUnit3** or **4** — the choice is entirely up to you — and hit the **Finish** button.

**Figure 16. To JUnit 3 or to JUnit 4**
Set up a new test case

Now that you've got JUnit in your project's classpath, you can write a JUnit test. Right-click on the java source folder and select New followed by JUnit Test Case. Define a package, give the test case a name (like SongTest) and in the Class Under Test section, hit the Browse button.

Note that you can select the Song class that you defined in Groovy. Figure 17 demonstrates this step:

Figure 17. Find the Song class
Select that class and hit **OK** (you should see a dialog box similar to the one in Figure 18) and then hit the **Finish** button in the New JUnit Test Case dialog.

**Figure 18. A new test case for Song**
Define the test method

I chose to use JUnit 4; consequently, I went ahead and defined a test method named `testToString()` like so:

```java
package org.acme.groovy;
import org.junit.Test;

import org.junit.Test;
```
public class SongTest {
    @Test
    public void testToString(){
    }
}

Test toString

Obviously, you want to verify that the `toString()` method is clean, so what's your first step? If you thought to yourself "import the `Song` class" you are thinking too hard — the `Song` class is in the same package. Consequently, the first step is to create an instance of it.

In creating an instance of `Song` to test, note that you don't have the ability to fully initialize an instance via a map passed into a constructor — what's more, if you attempt auto-completion of the instance's setter methods, see that each setter takes an `Object` rather than a `String` (look closely at Figure 19, just in case). Why is that?

Figure 19. All the setters and getters are there
Groovy makes it work

If you remember at the beginning of the tutorial, I said:

Because every object in Java extends java.lang.Object, even in the worst case scenario where Groovy can't ascertain what a variable's type is, all Groovy has to do is make it an Object and things will, believe it or not, work out.

Now recall that when you defined the Song class, you omitted each property's type. Groovy, being pretty groovy, naturally made each property an Object. So, when you attempt to use the Song class in normal Java, you are presented with a lot of getters and setters that all have Object as both a return type and parameter type.

Refining return types

Just for fun, open up the Groovy Song class and change the artist property to be of type String rather than typeless, like so:

```groovy
package org.acme.groovy

class Song {
    def name
    String artist
    def genre

    String toString()
    "${name}, ${artist}, ${getGenre()}"

    def getGenre()
    genre?.toUpperCase()
}
```

Now, go back to your JUnit test, and tap the auto-completion shortcut on your Song instance — do you see what I see?

In Figure 20 (and hopefully in your own follow-along code) the setArtist() method takes a String rather than an Object. Once again, Groovy proves that it is Java under the hood; and thus applies the same rules.

Figure 20. Strings rather than objects
It's always plain old Java

Going back to writing the test, please also note that, by default, Groovy's compiled class properties are private, so you can't directly access them in Java. Instead, you have to use setters like the following:

```java
@Test
public void testToString(){
    Song sng = new Song();
    sng.setArtist("Village People");
    sng.setName("Y.M.C.A");
    sng.setGenre("Disco");
    Assert.assertEquals("Y.M.C.A, Village People, DISCO",
                      sng.toString());
}
```

Coding the remainder of this test case is trivial. What it nicely demonstrates is that anything you do in Groovy can be easily re-used in your Java programs. Don't forget that the opposite is also true. Everything you do and have written in the Java language is also available to you in Groovy.
Section 13. In conclusion

If there’s one thing you take away from this tutorial (other than your first experiences of programming Groovy) it should be the deep understanding that Groovy is Java without a lot of the syntactical rules you’re used to. Groovy is Java without types, without modifiers, without returns, without Iterators, without importing collections. In short, Groovy is Java without a lot of the baggage that can weigh down your Java projects when you really need them to fly.

And yet, under the hood, Groovy is just Java.

I hope you've had fun on this first journey toward Groovy fluency. You've played around with Groovy's syntax, created a few classes that exercised its productivity-enhancing features, and seen how easy it is to test a Groovy class using Java. You've also encountered some of the pitfalls common to first-time Groovy developers, and seen how to work around them without creating too much of a mess.

While you probably don't consider yourself quite fluent in Groovy at this point, you've taken the first steps. You can use what you've learned so far as a basis for writing your own first Groovy programs — after all, you've got your dual Groovy-and-Java programming environment all set up! For an interesting exercise, try setting up your next round of automated builds with Gant, an Ant-based build tool that uses Groovy rather than XML to define the build. Once you’re more comfortable with the language you can try building a Web application module using Groovy on Grails — which, incidentally, is the subject of my next tutorial.
Resources

Learn


- "**Mastering Grails: Build your first Grails application**" (Scott Davis, developerWorks, January 2008): Get started with Grails, a modern Web development framework that seamlessly integrates legacy Java code with the flexibility and dynamism of Groovy.

- "**Secrets of lightweight development success, Part 7: Java alternatives**" (Bruce Tate, developerWorks, September 2005): Find out what makes a language productive, or not, in this high-level discussion of closures, continuations, reflection, and metaprogramming.

- "**What's New in Groovy 1.5**" (InfoQ, Guillaume Laforge, December 2007): Groovy's project manager highlights Groovy's new support for Java 5 annotations, generics and enums.

- "**Groovy-power automated builds with Gant**" (Klaus Berg, JavaWorld, February 2008): Find out why some Java developers are choosing Gant as a more expressive alternative for complex builds.

- **The Disco Blog's Groovy articles** (Andrew Glover, thediscoblog.com): Where Andrew writes about various topics related to Groovy, such as unit testing, metaprogramming, and advanced Groovy development techniques.

Get products and technologies

- **IBM developer kit for Java technology 1.5.0 SR3**: Powerful support for Groovy development on the Java platform.

- **Sun JDK 1.5 or later**: You'll need at least version 1.5.0_09 to follow the examples in this tutorial.

- **Eclipse IDE**: Set up your Groovy development environment easily with the Eclipse Groovy Plugin.

- **Groovy**: Hosted by Codehaus.

- **JUnit**: Examples are compatible with JUnit 3 or JUnit 4.

Discuss

- **The Groovy Zone**: The DZone community for Groovy and Grails developers.

- **Improve your code quality**: Andrew Glover's developerWorks discussion for
developers focused on test-driven development, code quality, and reducing risk.

About the author

Andrew Glover
Andrew Glover is president of Stelligent Incorporated, which helps companies embrace developer testing strategies and continuous integration techniques that enable teams to deliver software faster. Check out Andy's blog for a list of his publications.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.