Getting Groovy without the bad clothes

by James Williams

What is Groovy?

Groovy is a dynamically-typed scripting language for the Java Virtual Machine that serves the dual function of providing Java developers an easy entry point to scripting languages and the inverse for folks who use scripting languages.

How to get Groovy?

You can download Groovy at http://groovy.codehaus.org. To avoid having to run things in the same directory, it might be a good idea to add the location to the classpath. Inside the zip, you'll find the executables:

groovy	groovy interpreter for pre-made scripts
groovyc	compiles Groovy to Java byte code, afterwards the code can be run with the java keyword
groovysh	command-line interpreter
groovyConsole	GUI interpreter that allows loading and saving scripts

Why Groovy?

Why not use JRuby, Jython, or some other scripting language instead? You can, it's all a matter of choice. There is already a number of scripting languages, all of which have their own adherents and detractors, pros and cons.

Groovy is an good choice because it uses a Java-like syntax, can be embedded in Java, can be compiled down to byte code, and can inter-operate and subclass Java classes. The API relationship between Groovy and Java is like that of the major Romance languages (French, Spanish, Italian) but closer. Learning the first greatly reduces the barrier to entry of the second.

Your First Program

Hello World in Groovy looks very similar to Hello World in Java:

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

Ok, so it is exactly the same as Hello World in Java. The key point is that Groovy is similar enough to recognize about 95% of Java code. Why you would want to write all that in Groovy is beyond me but you can if you like. Groovy is all about writing more concise code, so lets do that. Groovy only requires parentheses when they are absolutely required(conditional statements and no parameter functions). Semicolons are also optional and only required when separating two or more statements on one line. Code can also run outside a class instance.[Technically it compiles the code and injects it into a class but for our purposes, we'll assume the former].

Our five line program, 80% of which was environment setup, is now:

```
System.out.println “Hello, World!”
```

The print functions have been specially registered so they can be called without the full classpath. Now we have:

```
println “Hello, World!”
```

Declarations

Variables and functions must have an access modifier, a type or def. For functions, def acts like public. These are NOT required for function parameters.

```
// These are fine
def a
int i
public x
```

N-Factorial

When it comes to declaring functions, the use of parentheses is required. Here is the corresponding code for n-factorial. Though we can specify the return type, we can let Groovy determine the type at runtime.

```
public fact(a) {
        if (a>1)
                return a*fact(a-1)
        else
                return 1
}
```

As we just saw, if-else functions just like a Java if-else statement. While loops also are a direct transplant. Do...while doesn't work at all in Groovy and the for loop has a slightly different syntax.

For-loop

Groovy's for-loop seems to be more in line with the for-each loop from Java 1.5

```
public class ForExample {
	public static void main(args) {
		for(arg in args)
			println arg
	}
}
```

Specifying the type of the interval variable is not needed.

Switch-case

In Java, the cases for a switch-case can only be Strings, integers, or enumerations. Groovy will take anything including classes.

```
b = ["Word",42]

for (i in b) {
	switch (i) {
		case String.class:
			println "I am a String"
			break
		case Integer.class:
			println "I am an Integer"
			break
		default:
		break
	}
}
```

Don't worry about the b-assignment line for now. We'll talk about that next. As we can see, the switch checks the class type of the variable and acts accordingly. This feature allows use to write more concise expressive code without 50 million if-statements.

Collections

Groovy has built-in support for Java Collections. By default, they are all either ArrayLists or Maps unless specified.

ArrayLists

For Groovy collections, there is no new keyword or curly braces, just the elements of the array or hash separated by commas.

names = [“John”, “Jake”, “Kate”]
ages   = [10, 12, 14]

To declare the arrays with static types, we can use the as keyword.

names2 = names as String[]
ages2 = [11,13,15] as int[]

In the case of the ages and ages2 ArrayLists, the integers are promoted transparently to the Integer class. As in Java, primitives types can not be added to a dynamic array. Luckily, we don't have that problem.

Adding, Removing and Accessing Elements

In addition to the add and remove functions that are present in Java, Groovy ArrayLists add a couple shortcuts.

f = ['A', 'B', 'C']
f += ['D']   //is the same as f.add(new Character('D')
f - = ['C']

An ArrayList can also be referenced like a plain array.

f[2]   // returns the same value as f.get(2)

HashMaps

Returning to our previous example, we an declare a hashmap by listing the key, followed by a colon, then the value. In this case, our key is the name and the value is the age.

names_ages = [“John”:10, “Jake”:12, “Kate”:14]

Groovy's hashmap unfortunately doesn't have any shortcuts for setting values so we have to use the put.

names_ages.put(“Joe”,15)

To retreive values, we can use the get function, or reference the key like a class variable.

names_ages.Joe   //is the same as names_ages.get(“Joe”)

If there is punctuation or anything that would cause Groovy to parse incorrectly, you must enclose it in single quotes

name_phone = ['Jean-Jacques':5551212, 'Jean-Marie':5557687]

name_phone.'Jean-Jacques'

Empty ArrayList and HashMaps can be declared with:

emptyArrayList = []

emptyHash = [:]

Closures

A closure is a reusable block of code with a built-in parameter it. Additional parameters can be specified if needed. Closures can be called on individual objects, are first-class objects themselves, and storeable in arrays and hashs.

We can create custom closures or add our code to pre-defined closure targets. The general for format of the closure body is

{ [optional parameters] -> [code] }

Any of these is valid for defining a closure that prints Hello and the name:

```
hello = { println “Hello” + it }
hello2 = { it -> println “Hello ${it}”}              // ${variable} is how to include variables
                                                    //without a bunch of + signs
hello3 = {println “Hello ${it}”}
```

The “it” variable is a special variable that every closure automatically has. As we saw above, it can be declared as a parameter to pass but it is not required. Let's redo our n-factorial function from part one to make it groovier:

```
fact = { if (it == 1)
            return 1
      else return it*fact(it-1)
}
```

We can call closures like functions, or with the call keyword. Any of these are valid:

```
hi()*                  hi.call()
hello “John”           hello(“John”)          hello.call “John”       hello.call(“John”)
fact 5                 fact(5)                fact.call 5             fact.call(5)
```

The real power of closures is realized when combined with collections and primitive objects. Instead of using a for or while loop, we could use:

```
3.times { println “Hi” }
1.upto(5) { println it }
20.downto(15) { println it}
```

Collections have a bunch of predefined closure targets but for brevity, I'll only talk about my favorites: each and findAll.

each iterates through the collection and executes the closure code on each element. findAll returns a new collection with the elements that satisfy the conditions of the closure.

```
[1,2,3].each {println it*it}            //prints 1
                                        //       4
                                        //       9
```

```
[“Jake”,”John”,”Cody”,”Christine”].findAll { it.startsWith(“C”) }
// returns [“Cody”, “Christine”]
```

Automatic getters and setter

We can assign values like a normal class variable or by using the getXXX pattern:

```
mathGrade.name = “John”
mathGrade.setSubject(“Math”)
mathGrade.grade = “A-”
```

Returning values can be done in the same manner:

```
println mathGrade.name              // returns “John”
println mathGrade.getGrade()        // returns “A-”
```

Overloaded constructors

There is really no need for multiple class constructors in Groovy. If we want to declare values in our constructor, we list the properties(or public variables) and their values, separated by colons(like a hash map)

```
artGrade = new GradeReport(name:'Kristen', subject:'Art', grade:'B+')
```

Using the above statement is the same as calling a no parameter GradeReport constructor and calling the setters for the name, subject, and grade. If we want a specific constructor that will not share code with the other constructors, we can declare it like a normal overloaded constructor.

XML Files

Consuming and creating XML documents is deceptively simple in Groovy (even more so that JDOM). Groovy uses what it calls builders to read and create structured documents and graphical user interfaces. The root element in this case is the albums element which takes a single attribute “owner” and its respective value. Everything between its curly braces are children of the root element. The element definitions resemble function calls and, as a result, are easy to read.

```
def builder = new groovy.xml.MarkupBuilder()
builder.albums(owner:'James') {
        album(name:'Flowers') {
                photo(name:'rose.jpg')
        }
        album(name:'people') {
                photo(name:'john.jpg', desc:'picture of John')
        }
}
```

The above builder generates the following text.

```
<albums owner='James'>
        <album name='flowers'>
                <photo name='rose.jpg' />
        </album>
        <album name='people'>
                <photo name='john.jpg' desc='picture of John' />
        </album>
</albums>
```

Reading files is a little more complicated. Given the following file:

```
<inventory>
      <section name="fiction">
            <book author="Jean-Paul Satre" title="La Peste" />
            <book author="Jean-Paul Satre" title="The Stranger" />
      </section>
      <section name="non-fiction">
            <book author="Bill Clinton" title="My Life" >Pages</book>
            <book author="Hilary Clinton" title="It Takes A Village..." />
      </section>
</inventory>
```

To list all the books and their content from each section using Groovy, we use the following script.

```
inventoryXML = new XmlParser().parse("inventory.xml")

for (section in inventoryXML) {
        println "Section: " + section['@name']
        for(book in section) {
                println book['@title'] + " by " + book['@author']
                if (book.text() != "")
```

```
                              println "Content: " + book.text()
          }
}
```

Attributes are retrieved with the an @ and the attribute name encased in single quotes and brackets. Content, specifically non-element data, can be retrieved using the text() function. For simplicity, I used the names of the elements but it’s not required. section in inventory is really telling groovy to retrieve the direct children of the root element.

Files

In lieu of the calls needed to several constructs in Java. In Groovy we need only one. The following code opens a file and prints its contents to standard output:

```
f = new File('someFile.txt').newReader()
println f.getText()
```

The top line creates a BufferedReader, which in addition to the functions from the Java API, includes a getText() function. Similarily, we can write to a file just as easily:

```
f = new File('someFile.txt).newWriter()
f.writeLine(“Hello, world!”)
f.close()
```

Unlike the BufferedReader which can be left to be garbage-collected, BufferedWriters, as you probably already know, will need to be explicitly flushed or closed. Returning to our XML example, only a couple changes are needed to write the XML to a file:

```
f = new File('/home/jwill/Desktop/dd.xml').newWriter()
builder = new groovy.xml.MarkupBuilder(f)
builder.albums(owner:'James') {
          album(name:'Flowers') {
                    photo(name:'rose.jpg')
          }
          album(name:'people') {
                    photo(name:'john.jpg', desc:'picture of John')
          }
}
f.close()
```

We're done...for now.