

Hello Groovy

Russel Winder

russel@russel.org.uk

Abstract

In which the venerable program ‘Hello World’ introduces some features of the Groovy programming language, a dynamic language that runs on the JVM symbiotically with Java.

1 The Plethora

‘Hello World’ (HW) seems to have first represented itself on page 6 of *The C Programming Language* by Kernighan and Ritchie (1978). It appeared as:

```
#include <stdio.h>
main ()
{
    printf ("hello, world\n" );
}
```

A short representation, but then short does not mean bad. In fact, these days, for a given functionality, short is definitely considered good, and long bad.

HW has since represented itself in many languages for many contexts. A quick perusal of the website <http://www2.latech.edu/~acm/HelloWorld.shtml> shows HW in 193+1 (193 listed and 1, Groovy, submitted) different languages – at the time of writing anyway, HW may have represented itself further in the interim. Although HW is hugely infectious, it is not a virus.

Interestingly, HW appears to have a sense of humour. It has represented itself in C for X, Gnome, KDE even Windows, presumably to prove that console output is simple and GUI output hard. For example, `helloX.c` (see <http://www.paulgriffiths.net/program/c/hello.html>) is a representation of HW in C for the X Windowing System. This representation is 135 lines of code, 25 of which are comment lines. The coding doesn’t seem excessive but it is far to long to show here – sizist maybe but there we are, space is at a premium.

So HW has a habit of representing itself in unusual and often bizarre ways to make a point about a programming language. HW is clearly a teacher. Though not certificated as the possession of a qualification might be considered pretentious – especially amongst some people who contribute to the accu-general mailing list. Being uncertificated doesn’t make HW a bad teacher, but it will make it difficult to find a position in a school. Universities though are less strict about teachers having qualifications and so HW may still be able to find a position there, possibly in textbooks.

Unfortunately, despite the various anti-discrimination laws, publishers are now definitely discriminating against HW, generally on the grounds of boredom and ‘oh no, not that f***** program again’. This has not daunted HW, HW is resourceful. Not only does it look for new outlets, this article for example, it is always on the lookout for new programming languages to represent itself in. For this article, HW has represented itself in a number of short ways to show how groovy the Groovy programming language is.

2 Groovying On

When Sun introduced Java to the world in the mid 1990s, they also introduced the Java Virtual Machine (JVM). Sun is reputed to have stated that the only programming language for the JVM was

Java. However, it seems that this may be a myth. Myth or not, many people thought the idea short-sighted and overly introverted and started developing other languages targeted at execution on the JVM.

People ported Python (Jython) and Ruby (JRuby) to work as dynamic programming languages (aka scripting languages) for the JVM. However, whilst these languages have a following, they have not caught on in any big way. It may well be that the lack of catching on is because Python and Ruby are based on different types to those supported by the JVM, leading to the need for type/representation conversion. However, the idea of a dynamic programming language for the JVM has caught on. Big Time. We have BeanShell, PNuts and, of course, Groovy, to name the most well-known three. HW has almost certainly been represented in all of these languages even if not listed on the HW site so that probably means 193+3 rather than the figure quoted earlier.

The success of 'scripting languages', aka dynamic languages, working symbiotically with Java has recently caused Sun to publicly state its enthusiasm for mixed language working on JVM-based systems. This shows Sun are not averse to doing and thinking the right thing even if they didn't originate the idea.

So where does HW stand on all this? Actually, it is totally agnostic, it cares not a jot about language wars, it will represent itself in any language. Java is one example that is apposite for this article:

```
public class HelloWorld {
    public static void main ( final String[] args ) {
        System.out.println ( "hello, world" );
    }
}
```

Of course this is horrendously verbose and complicated, especially if this is the first program you see when learning a programming language. It is hard to argue against this point when you observe HW representing itself in Python:

```
print 'hello, world'
```

and Ruby:

```
puts 'hello, world'
```

Compare these to HW's first representation in Groovy:

```
println ( 'hello, world' )
```

and we see that Groovy is as good as Python and Ruby and a lot better than Java, even on Java's own turf.

3 The Many Groovy Ways of HW

As mentioned earlier, HW, being a program with a wry sense of humour, has come up with a series of representations of itself designed to show various features of the Groovy programming language. These are not serious representations – we have already seen the serious version above – they are simply presented to show that Groovy is, well, groovy.

3.1 The State of Variables

Groovy has variables and so the obvious first variant representation of HW in Groovy is to use a variable to hold the string value to be printed and then to print the value of the variable:

```
String theString = 'hello, world'
println ( theString )
```

This representation of HW uses static (compile-time) typing. Groovy being a dynamic language does not need compile time type checking, HW can therefore use a variable of type determined at run time:

```
def theString = 'hello, world'
println ( theString )
```

The type of a dynamically typed variable can change during execution since it is the type of the last assigned value that determines the type of the variable. Some people find this hard to deal with and stick to static typing always. However this is not dynamic and not Groovy. Having said this, even in a really Groovy program not all variables are dynamically typed, it is often right and proper to have statically typed variables. Nonetheless, using `def` and dynamic typing is the usual way of working with explicitly initialized variables since it avoids repetition of information – the literal being assigned has a type so why specify the type again? Coupling `def` with the use of `final` to enforce single assignment, leads to a declarative style of programming which is considered Very Groovy.

Languages like Python and Ruby do not require specific declaration of variables, they just have declaration by usage. HW, being a good teacher, answers the question ‘What happens in Groovy?’ by trying the experiment and representing itself:

```
theString = 'hello, world'
println ( theString )
```

The program still works. Why? Each Groovy script has a binding object associated with it, so in this representation of HW `theString` refers to a variable in the binding – if there is no variable of that name already in the binding then one is created. Usually the binding is used for passing information from operating system to script, or embedding system to embedded system, but it can be used as a global shared memory. OK, global shared memory is generally considered bad (quite rightly except in certain very specific circumstances) so using the binding in this way is definitely not groovy even if it is Groovy. HW raises the issue simply to show there is an issue.

The above variables were scalars, HW now represents itself to show that we can have arrays of things:

```
String[] theStrings = ( 'hello' , 'world' )
println ( theStrings(0) + ' , ' + theStrings(1) )
```

The literal on the right hand side of the initialization is not actually an array, it is a list (`java.util.List` – well `java.util.ArrayList` actually but, as in Java, we use the interface and not the class). However, because of the static type of the variable is specified as array there is an automatic coercion from `java.util.List` to array.

What happens if we have a variable of dynamic type? In this situation, to ensure the assigned value is of array type, we have to manually coerce as HW shows in this representation:

```
def theStrings = ( 'hello' , 'world' ) as String[]
println ( theStrings(0) + ' , ' + theStrings(1) )
```

The syntax of coercion is clearly very un-Java like. For various reasons which need not concern us here, Java-style casting is not allowed in Groovy.

You are probably asking the question: ‘Why use arrays if we can use lists directly?’

‘Exactly.’ HW responds and represents itself thus:

```
def theStrings = ( 'hello' , ' , ' , 'world' )
println ( theStrings(0) + ' , ' + theStrings(1) )
```

Indexing into a list is supported by Groovy – lists are sequences like arrays so indexing makes a great deal of sense. Using lists rather than arrays is generally considered more Groovy but the ability to have arrays is required for using some Java APIs.

3.2 Repeating the Groove

Given that Groovy has lists, it must provide facilities for iteration enabling us to parameterize over the length of the list. Here is HW representing itself using the basic for loop in Groovy:

```
def theStrings = ( 'hello' , ' , ' , 'world' )
for ( i in 0..<theStrings.size() ) { print ( theStrings(i) ) }
println ( )
```

The expression `0..<theString.size()` is the sequence of integers starting with 0 and ending with 1 less than the upper limit. Classic idiom for iterating over zero-origin sequences. Of course, this sort of indexed

looping is, quite rightly, frowned upon these days as being too low level for the task of iterating through all the element of a collection. Instead, for doing such an iteration, we should use a ‘foreach’ construct:

```
def theStrings = ( 'hello' , ' , ' , 'world' )
for ( item in theStrings ) { print ( item ) }
println ( )
```

Actually these two loop constructs are no different in language terms, they are both foreach loops. The first is actually *for each i in the range [0, theStrings.size ())*. So there is only the one type of for loop in Groovy – Groovy does not support the `for (int i ; i < theStrings.size () ; ++i)` variety of loop that Java does. Well not yet anyway, it may soon. This is all, obviously, a bit different to the way Java does things, but this is just more Groovy – dynamic rather than static, operations on data structures as a whole rather than detailed knowledge of the underlying structure, declarative rather than imperative.

Of course, we can use the Java-style foreach loop with its static typing approach:

```
def theStrings = ( 'hello' , ' , ' , 'world' )
for ( String item : theStrings ) { print ( item ) }
println ( )
```

but it is considered boring and tedious by Groovy programmers. Actually boring and tedious is generally the view Groovy programmers have about Java. Groovy programmer do know and use Java, much Groovy programing is about using classes and objects from the Java Platform after all, but they only program in Java when necessary – mostly in the same way that C++ programmers use assembler.

3.3 Closing in on Closures

Actually most Groovy programmers would probably think all the above was tedious and boring since it doesn’t involve the programing construct that really separates Groovy and Java: *closures*.

Closures are Totally Groovy – despite the idea actually being old and available in many programming languages. The biggest win for closures is that they allow a much more declarative expression of algorithms. HW has to admit that the following is a representation of itself but really doesn’t show why closures are just so cool:

```
def theStrings = ( 'hello' , ' , ' , 'world' )
theStrings.each { print ( it ) }
println ( )
```

Here HW uses the method `each` applied to a list and being passed a closure to be executed for each item of the list. `it` is a special variable name in a closure being the implicit closure parameter.

‘Aha’, you say, ‘there is no method `each` on lists in Java.’

‘Indeed,’ replies HW, ‘but Groovy can add methods to standard Java classes using its groovy meta-object protocol.’

In this particular instance, Groovy has added the `each` method to Java list types exactly so that this technique of iterating over lists and applying closures can be used.

You could think of the above representation of HW thus:

```
def theStrings = ( 'hello' , ' , ' , 'world' )
theStrings.each ( { print ( it ) } )
println ( )
```

but people invariably remove the optional parentheses to avoid having extra syntactic clutter. Actually it is a wee bit more complicated than this but HW has not been willing to come up with the seriously over-contrived representation required. There are limits, even to ridiculousness.

Returning to it, if we want to explicitly name the closure parameter then we can, as HW shows here:

```
def theStrings = ( 'hello' , ' , ' , 'world' )
theStrings.each { item -> print ( item ) }
println ( )
```

This is probably the more usual way of working with closures.

Of course, this algorithm, however it is expressed, involves a lot of output statements. Many feel (and the argument is a good one) that minimizing the number of output statements is generally a good thing. So concatenating the strings before outputting is probably a good thing. Groovy extends the Java list types with a `join` method so we can achieve concatenation without knowing how many items in the list:

```
def theStrings = ( 'hello' , ' ' , 'world' )
println ( theStrings.join ( "" ) )
```

In Groovy, closures are first class entities so we can have variables of closure type and pass closures around as parameters. Here HW presents a fairly gratuitous representation of itself highlighting initialization of an object that has a closure member and then execution of that closure via the class data member:

```
class ClosureWrapper {
    Closure action
    def execute () { action () }
}
new ClosureWrapper ( action : { println 'hello, world' } ) .execute ()
```

The initialization is interesting here. The class declares a data member and a method. The data member is declared to be a property – the member will be a private member and accessors will be automatically generated to fulfil the requirements that the class be a ‘bean’. No constructor is declared. On initializing the new `ClosureWrapper` object we provide a constructor parameter. From a Java mindset this is clearly an error that the compiler will detect. Distinctly not Groovy. Groovy is quite happy with this code since it allows a map to be used to initialize all properties of a new instance. In this case the parameter is a single value map: the colon separates the two components of the map value, `action` is the key and the closure is the value associated with that key. The key must be the name of a property of the class of course! This technique for initialization of properties using a map leads to some seriously useful idioms.

3.4 OSsification

Many people believe that scripting languages must be able to script the execution of operating system jobs à la `bash`, `ksh`, etc. Although Groovy is a general purpose dynamic language (like Python and Ruby), it can be used for scripting and so must be able to do the job. Here HW represents itself in a way designed to show what is possible:

```
print ( ( 'echo' , 'hello, world' ) .execute () .text )
```

Applying the `execute` method to a list of strings creates a process object (`java.lang.Process`) on which Groovy has defined the `getText` method – two more instances of Groovy adding methods to standard Java classes. This means we can write code as though it is accessing a variable when in fact it is calling an accessor: using `text` as a property accesses the member variable if it exists or if it doesn't then the method `getText` is called if it exists. Metaclasses and beans at work in an extremely constructive way. Very Groovy.

3.5 Swinging Groovily

All the above are console based but what about GUI based representations? Groovy supports builders, and a builder for constructing Swing/AWT interfaces is part of the standard distribution. Builders use all the facilities of a dynamic language with a meta-object protocol to simplify building hierarchical systems. Add to this the map initialization of properties construction technique and Swing programming becomes easy:

```
def frame = new groovy.swing.SwingBuilder () .frame (
    title : 'Hello' ,
    defaultCloseOperation : javax.swing.JFrame.EXIT_ON_CLOSE ) {
    label ( "hello, world" )
}
frame.pack ()
frame.visible = true
```

Note that we use `frame.visible = true` where in Java you would have to say `frame.setVisible (true)`. As noted earlier, Groovy allows properties to be accessed as though they were data members rather than having to use accessor calls.

The ‘wow factor’ here is partly how the use of closures and hash parameter properties initialization shorten the code but is mainly the fact that `SwingBuilder` does not actually have any methods such as `frame`, `label`, etc. The metaclass associated with the `SwingBuilder` object knows how to turn what appear to be method calls into the construction of Swing/AWT components. It does this by reflection and understanding the naming conventions rather than by explicitly mapping method call names to component type names. Very flexible. Most Groovy.

Anyone having any experience with Swing/AWT must surely agree with HW that this way of constructing Swing/AWT GUIs is just ‘cool’ and ‘groovy’.

4 1, 2, 3, Testing

Interestingly HW is actually a bit of an advocate of unit testing though it isn’t too sure about test-driven development but that may be because HW can be represented in so many ways without having to develop. This is one of the wonders of HW.

So we are agreed that all the representations of HW need to be tested to ensure semantic correctness. Clearly we must separate the console-based representations from the GUI-based representations. For the console-based representations we can construct a program that tests all of the programs in a given directory:

```
class helloWorld_Test {
    static void main ( args ) {
        def testClass = '''
class TestHelloWorld extends GroovyTestCase {
    private final expected = 'hello, world'
    private final shell = new GroovyShell ()
    private final saveSystemOut = System.out
    private final buffer = new ByteArrayOutputStream ()
    private final outputFromScript = new PrintStream ( buffer )
    void setUp () { buffer.reset () ; System.setOut ( outputFromScript ) }
    void tearDown () { System.setOut ( saveSystemOut ) }
    void evaluateFile ( String filename ) { shell.evaluate ( new File ( filename ) ) }
    String getOutput () { return buffer.toString ().trim () }
'''
        new File ( '.' ) .eachFile { s ->
            def script = s.name.trim ()
            if ( script =~ /^helloWorld_(a-z).*\.groovy$/ ) {
                def methodName = 'test' + script.substring ( script.indexOf ( '_' ) , script.lastIndexOf ( '.' ) )
                testClass += " void ${methodName} () { evaluateFile ( '${script}' ) ; assertEquals ( output , expected ) }\n"
            }
        }
        testClass += ' ; return TestHelloWorld'
        def shell = new GroovyShell ()
        shell.runTest ( shell.evaluate ( testClass ) )
    }
}
```

Wow, this is getting seriously serious.

This program constructs a program (comprising a class definition and a return statement) as a string, then evaluates the string (using a `GroovyShell`) which compiles the program and loads it into the running JVM. The result of this compilation and load is a reference to the class object which is then used (by a `GroovyShell`) to execute the program as a JUnit test. The `GroovyShell` is just so Groovy. Actually this is very true, the `GroovyShell` is arguably the single most important class in the Groovy system since it is responsible for executing any and all Groovy programs.

Reactions to all this range from ‘Wow, just how cool is that.’ to ‘OK, isn’t that just obviously the right thing to do.’ depending on whether you are new to dynamic programming languages or an old hand.

As you have probably guessed already, triple single quotes or triple double quotes start a multi-line string, i.e. one that can include end-of-lines as data. The example here is used to present the literal that is the fixed text part of the generated program.

The generated class `TestHelloWorld` is a subclass of `GroovyTestClass` so that it is treated as a (JUnit-based) test class. The variables (`expected`, `shell`, `saveSystemOut`, `buffer`, `outputFromScript`) and methods (`setUp`, `tearDown`, `getOutput`) implement the infrastructure for capturing the standard output of executing a script via a `GroovyShell` so that the test program can capture the output from executing the various representations of HW – which is achieved using the `evaluateFile` method. All the test methods are written in the closure used in the iteration over all the names of the files in the directory.

As noted earlier, Groovy allows extension of standard Java classes. Here we see use of the `eachFile` method that has been added to `java.io.File` to support using a closure as part of a closure style iteration.

In the closure itself, we are using regular expressions (things that look like regular expressions initiated with a `/` and terminated with a `/`) and the match operator (`=~`) to make decisions about writing the necessary test methods for any given representation of HW in the current directory. The name convention being assumed and applied is that an HW representation will be in a file starting 'helloWorld_', finishing '.groovy' and comprising only lower case letters.

You might have noticed the use of `${methodName}` in a string. These are substitution markers and the string is not a standard Java string as the examples have been so far but is a `GString`, i.e. a string in which substitution markers are substituted.

Having said all this, it remains that the single most important thing about this example is the idiom of iteration using a closure.

5 Building Groovily

In Section 3.5, HW represented itself as a Swing application using builder technology. Builders are so Groovy that HW thought it appropriate to show how Ant (the standard Java system build tool) can be scripted in Groovy avoiding all the need to have any XML files at all. What a lovely lack of angle brackets.

Here is the Groovy script that handles all the building for the Groovy representation of HW used in this article:

```
class build {
    private final ant = new AntBuilder ()
    def init () {
        ant.taskdef ( name : 'groovy' , classname : 'org.codehaus.groovy.ant.Groovy' )
    }
    def test () {
        init ()
        ant.groovy ( src : 'helloWorld_Test.groovy' )
    }
    def clean () {
        ant.delete ( quiet : 'true' ) {
            ant.fileset ( dir : '.', includes : '*~*,*.class' , defaultexcludes : 'no' )
        }
    }
    static main ( args ) {
        def builder = new build ()
        if ( args.length == 0 ) { builder.test () }
        else { args.each { target -> builder.invokeMethod ( target , null ) } }
    }
}
```

Build targets are represented as methods which then use builder technology (including property initialization with maps and nesting using closures) to script calls to the tasks in the Ant library. Anyone at all familiar with using Ant will very quickly see how this is working and appreciate the benefits of using a scripting language rather than XML to define the build.

It is possibly interesting to note the `if` statement in the main method of the class. The `if` branch implements the test target being the default target and the `else` branch, which is a closure being applied in order to all the targets listed on the command line, shows how the metaclass system of Groovy can be used directly to call methods using the string representation of the method name. This is the Groovy metaclass system being used directly. Very dynamic. Very Groovy.

Interestingly, the success of builders in Groovy has caused Ruby to really take builders on board.

6 Getting in the Groove

Languages like C gave way to C++ because people could express things more easily. Java supplanted C++ for many because of ease of expression and portability. Python and Ruby allow much more dynamism, as does Groovy. No one language is better than another per se. Having said that Java people trying Groovy rarely look back at Java as better.

If this article has piqued your interest, and HW hopes it has, the Groovy website home page is <http://groovy.codehaus.org>. Remember though the Groovy documentation is not particularly good yet even though the language itself is excellent. Python and Ruby have had 10 year or more work on their documentation, Groovy is still being build – 2006-07 should see the first formal release of the system itself. Work then starts on making the documentation of good quality. Also work will begin in earnest on the TCK ready to progress JSR-241 which will lead to Groovy becoming an integral part of the standard Sun Java distribution.

A final Note: HW has not been able get a look in on Groovy documentation as yet due to the Wallace/Grommit/cheese fixation of the original Groovy authors. See the Groovy website for more on this problem.

The other final note: HW apologies for becoming a mop-head but meta-object protocols just rock its world.

References

Kernighan, B. and Ritchie, D. (1978), *The C Programming Language*, Prentice–Hall.