Crossroads
The ACM Student Magazine

A Crash Overview of Groovy

by ***Kevin Henry***

Introduction

As computer programmers, we are often confronted with a decision between a "programming language" (C, C++, Java, and so on) and a "scripting language" (Bourne shell, Perl, Python, and so on).

Scripting languages allow for light-weight programs that are often quick and easy to write, which makes writing a script more attractive than writing a full-fledged program for some common tasks; however, there are times when we want the convenience of a scripting language without giving up the features of a programming language.

Groovy is "an agile dynamic language for the Java 2 Platform" [1] that attempts to combine the convenience of scripting with the functionality of Java. Groovy has "many of the features that people like so much in languages like Python, Ruby and Smalltalk, making them available to Java developers using a Java-like syntax" [1]. One reason for Java's popularity as a programming language is the Application Programmer Interfaces (APIs) [11]. Each API is a set of related classes that are useful to programmers in Java, similar in concept to the C++ Standard Template Library (STL). The Java APIs are available to Groovy programmers with simplified script-like syntax.

This article exemplifies some major features of Groovy syntax and mentions some other common applications of Java that Groovy supports. Readers are encouraged to visit the Groovy web site and investigate further.

Groovy Syntax

Several syntactical features absent from Java are quite useful in other languages. These features are present in Groovy and allow users access to the Java API. One might argue that some of these syntactical shortcuts do not belong in Java and were left out deliberately because they encourage poor design. However, though a language should facilitate good design whenever possible, ultimately the burden of good design rests with the programmer.

Dynamic Typing & Syntax Shortcuts

In languages with static typing, a variable must first be declared as a certain data type before it can be assigned a value. Furthermore, assigning a value of one type to a variable declared as another type usually causes an error. This is the case with the most popular programming languages (like C, C++, and

Java). In a language with dynamic typing, however, the data type of a variable is determined by the value assigned to it. Dynamic typing is a very convenient feature of most popular scripting languages. For example, consider this Java program that uses static typing of variables:

JavaST.java

```
import java.util.StringTokenizer;
public class JavaST {
    public static void main( String[] args) {
        String string = "Some string we'll tokenize";
        StringTokenizer tok = new StringTokenizer( string);
        while( tok.hasMoreTokens())
            System.out.println( tok.nextToken());
    }
}
```

The variable `string` is an object of the class `String`, and `tok` is an object of the class `StringTokenizer`. `StringTokenizer` must be imported from the `java.util` package before it can be used in the program. Groovy takes care of this step automatically.

The same task is much easier in Groovy because of dynamic typing. Groovy also provides syntactic shortcuts for common tasks in Java, like screen output. The example below represents the above example using dynamic typing and the simplified output syntax:

GroovyST.groovy

```
string = "Some string we'll tokenize"
tok = new StringTokenizer( string)
while( tok.hasMoreTokens())
    println tok.nextToken()
```

The same variables hold the same data, but their type is determined by what data is assigned to them. The output of the script is the same as the program:

```
Some
string
we'll
tokenize
```

Given that both code examples accomplish the same result, which is more convenient to write?

Native Syntax for Lists and Maps

Most scripting languages have native syntax for creating **list** and **map** data structures. The C++ STL and core Java API provide these data structures and Groovy conveniently provides a native syntax for lists and maps.

Here's an example that creates a list and performs some operations using list and **range** syntax:

List.groovy

```
list = [ 10, 20, -5, 100, -987]
println list
println list[ 2]
println list[ list.size-1 .. 0]
```

And the output is:

```
[10, 20, -5, 100, -987]
-5
[-987, 100, -5, 20, 10]
```

Groovy also has native syntax to create a map. A key into the map can be treated as property as long as the key is a string that identifies the value to return:

Map.groovy

```
map = [ hello: 5, whatever: 8, "who am i?": "kevin"]
println map
println map.whatever
```

This time the output is:

```
{whatever=8, hello=5, who am i?=kevin}
8
```

Remember that the map data structure only specifies that the values will be accessible by their keys; it does not specify the order of key-value pairs in the map.

Closures

Macros and function pointers are familiar to C programmers. One use of a macro is to perform a simple function without explicitly writing the function and calling it from code. Groovy provides **closure** syntax. Closures are anonymous code fragments that can be assigned to variables. They can even be passed as arguments to other methods. In this way closures provide much of the same capabilities that macros and function pointers provide in C.

Here is an example of a closure that calculates the area of a triangle given the base and height. The closure is defined and assigned to the variable `triangle_area`. (This saves the programmer from explicitly defining a class with a static method to perform whatever function the closure will perform.) Invoke a closure by its `call()` method with the appropriate number of arguments. A closure returns the value of the last statement in the code.

Triangle.groovy

```
triangle_area = { base, height -> 0.5 * base * height }
a = triangle_area.call( 5, 4)
println a
```

The output is:

```
10.0
```

Closures can include more than one executable statement. Otherwise they would be of very limited use. Here is another example of a closure that finds the minimum value in a list using Groovy's **for** loop syntax to iterate through the elements of a list:

MinValue.groovy

```
listMin = { someList ->
    min = someList[0]
    for( i in someList)
        if( i < min)
            min = i    return min;}myList = [ 8, 6, 7, 5, 3, 0, 9]
println listMin.call( myList)
```

The output is the minimum value in the list:

```
0
```

This section introduced closures but barely scratched the surface of their true potential. For example, closures maintain the lexical scope of where they are defined. The Groovy web site provides an in-depth exploration of the powerful closure syntax (See [1]).

Operator Overloading

Operator overloading is arguably the most powerful feature of C++, a feature Java's designers chose not to include in Java. Groovy does not provide the arbitrary operator overloading that lends itself to abuse in C++ but pairs nineteen common operators to method names so that using an operator will have the same effect as a method call with the corresponding name [2]. The following operators correspond to method names:

Operator	Method
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.multiply(b)
a / b	a.divide(b)
a++ or ++a	a.next()
a-- or --a	a.previous()
a[b]	a.getAt(b)
a[b] = c	a.putAt(b, c)
a << b	a.leftShift(b)

The following comparison operators are also provided and handle `null` values safely. That is, they do not cause a `NullPointerException` to be thrown if one or both values being compared are null. Furthermore, when comparing two different numeric types, Groovy will promote the smaller type to the larger type before performing the comparison.

Operator	Method
a == b	a.equals(b)
a != b	! a.equals(b)
a === b	a == b in Java (i.e. a and b refer to same object instance)
a <=> b	a.compareTo(b)
a > b	a.compareTo(b) > 0
a >= b	a.compareTo(b) >= 0
a < b	a.compareTo(b) < 0
a <= b	a.compareTo(b) <= 0

The result of this syntax is that users are not permitted a complete arbitrary operator overloading. For example, a user cannot overload parentheses to calculate a factorial as in C++ (See [3]). However, by writing a `plus()` method and a `multiply()` method users gain overloaded + and * operators for objects of their classes.

ModOps.groovy

```
class ModMath {    int val;
    int mod;

    ModMath( mod, val) {
        this.mod = mod;
        this.val = val % mod;
    }

    int plus( j) {
        return (val + j) % mod;
    }

    int multiply( j) {
        return (val * j) % mod;
    }
}

six = new ModMath( 7, 6);   // 6 mod 7
println six + 5;            // 6 + 5 = 11 mod 7 = 4
println six * 5;            // 6 * 5 = 30 mod 7 = 2
```

The output is:

```
42
```

In this way Groovy encourages rational use of operator overloading; if users are determined to abuse operator overloading to the point of implementing a `plus()` method that performs subtraction, then Groovy will not help.

Scripting Syntax for Regular Expressions

Java has a Regular Expressions API that provides regular expression functionality, but it is given in a scripting syntax similar to the regular expressions syntax in Perl or Python. Groovy provides an operator to match a regular expression (=~) and to create a `Matcher` (=) that will perform the matching.

For example, below is a script that demonstrates string substitution to squeeze space characters. First it creates a regular expression for more than one space. Next it creates a matcher for that expression and performs a substitution:

Squeeze.groovy

```
string = "Why  would       someone  type with  so    many     spaces?"
println string

matcher = string =~ " +"fixed = matcher.replaceAll(" ")println fixed
```

The output is:

```
Why  would       someone  type with  so    many     spaces?
Why would someone type with so many spaces?
```

Java regular expressions (described in [4]) are the basis for regular expression support in Groovy.

More Support to Java Programmers

Groovy provides the syntactical elements described above (and many more) to give users the convenient syntax of a scripting language with access to the Java API. Groovy also provides support to Java programmers in several other respects by including extensions to support the following programming tasks that Java programmers frequently encounter.

Markup

Groovy includes support for a variety of markup languages by providing builder classes for XML, HTML, SAX, W3C DOM, Apache Ant tasks, and so on. Given a markup the builder will create a document conforming to the appropriate document standard. Groovy even includes a SwingBuilder class that will create a GUI using Java Swing classes according to a markup describing the GUI to build. See GroovyMarkup in [5] and Ant Scripting in [6].

Path expression

Groovy includes GPath, a language to manipulate tree structured data, including XML. Using GPath, a programmer can specify a path to an element and attributes of that element [7].

Unit tests with JUnit

By creating a class that extends `groovy.util.GroovyTestCase`, users can write test cases that take advantage of the JUnit test framework, Groovy's extensions of syntax, and eleven more assertions added by Groovy to the assertions provided by JUnit [8]. A user can also script his or her test cases using the Apache products Ant or Maven.

Groovlets

Groovy lets users write servlets in Groovy ("Groovlets") and gives users `GroovyServlet` to compile, load, and cache their Groovlet until they change the source code. In this way Groovlets behave like Java servlets [9].

Beans

Groovy provides a very simple syntax for working with Beans and manipulating their properties. Users use the `Property` keyword to identify the properties of their GroovyBean. Groovy creates the appropriate member data of a user's Bean class and provides the equivalent of get and set methods as appropriate to the visibility of each property [10].

Conclusion

This article introduced Groovy and highlighted some of its major features. The most important features are scripting-language syntax, full availability of the Java API to Groovy scripts, and Groovy support for several tasks commonly undertaken by Java developers at the present time.

This article is by no means a comprehensive introduction to Groovy. The best resource available is the Groovy web site, which contains a wealth of information about the language as well as links to code examples and many other articles.

Thanks

Special thanks to Tobias DiPasquale of the **Cipher Block Chain Gang**, for making me aware of Groovy and suggesting this article.

References

1
"Groovy - Home" at **http://groovy.codehaus.org/**

2
"Groovy - Operator Overloading" at **http://groovy.codehaus.org/Operator+Overloading**

3
Bjarne Stroustrup demonstrated overloading parentheses to calculate a factorial during a lecture at Lebanon Valley College on 10 April 2000.

4
"Lesson: Regular Expressions," from The Java Tutorial, at **http://java.sun.com/docs/books/tutorial/extra/regex/**

5
"Groovy Markup" at **http://groovy.codehaus.org/GroovyMarkup**

6
"Ant Scripting" at **http://groovy.codehaus.org/Ant+Scripting**

7
"GPath" at **http://groovy.codehaus.org/GPath**

8
"Groovy - Unit Testing" at **http://groovy.codehaus.org/Unit+Testing**

9
"Groovy - Groovlets" at **http://groovy.codehaus.org/Groovlets**

10

"Groovy - Groovy Beans" at **http://groovy.codehaus.org/Groovy+Beans**

11 "Java Technology" at **http://java.sun.com/**

Biography

Kevin Henry (**khenry@gwu.edu**) is a M.S. student in the Department of Computer Science, The George Washington University. Kevin's primary interests include object-oriented software engineering, cryptography, and high-performance computing.