Introduction to Groovy

Jason Winnebeck

Introduction

- Runs on the Java Virtual Machine (JVM)
- Dynamically typed language
- Everything looks like an object, even primitives
- Inspired by Java, Python, Ruby, and Smalltalk
- Supports the majority of Java syntax, and adds to it
- First language (besides Java) to take part in Java Community Process (JCP) as JSR 241
- Syntax support for domain-specific languages (DSLs)
- Metaprogramming (static and runtime extensions/modifications to existing types)

Target Audience

Due to supporting most Java syntax and ability to integrate with any existing JVM code, Groovy is particularly useful for existing Java developers wanting to leverage dynamic language features while retaining use of the full Java ecosystem.

Hello, Groovy!

One can directly run scripts like this as "groovy Hello.groovy"

```
def name = "Groovy"                    //semicolon optional
def greeting = "Hello, $name!"         //String interpolation
assert "Hello, Groovy!" == greeting //== does .equals, handling null
assert !("Hello, Groovy!".is(greeting)) //identity check

//optional typing, def is like Object
String typed = "Hello"
def untyped = typed
assert untyped instanceof String

//defining methods is same as Java; def maps to "Object"
void sayHello( def greeting ) {
  //dynamic call checked at runtime
  greeting = greeting.replace( "Hello", "Bonjour" )
  //methods added to Object, include println
  this.println( greeting )
  println greeting //"this" optional as in Java, also () optional
}

sayHello(greeting)
```

Data Structures

```
//Lists
def intList = [1, 2]
def emptyList = []
def emptyArrayList = [] as ArrayList

//modify in place
intList << 3
assert [1,2,3] == intList
assert 3 in intList //calls contains method
assert ( intList[0] == 1 ) && ( intList[-1] == 3 )

assert [1,2,3,4] == intList + 4        //add method
assert [1,2,3,4,5] == intList + [4,5] //addAll method
assert [1,3] == intList - 2            //remove method

//Maps, with 3 ways to access
def ages = [Alice: 20, Bob: 25, Sally: 12]
assert [ages.get("Alice"), ages["Alice"], ages.Alice].every { it == 20 }
ages.Alice += 2
assert ages["Alice"] == 22

ages += [John: 60]
assert ages.John
```

Beans

```groovy
import groovy.transform.Canonical

@Canonical //add equals, hashCode, and toString
class Person {
  String name
  int age
  def extraInfo //same as Object
}

def alice = new Person( name: "Alice", age: 25, extraInfo: "A girl" )

assert alice.name == "Alice"
assert "Person(Alice, 25, A girl)" == alice.toString() //uses Canonical

def bob = new Person( name: "Bob", age: 40, extraInfo: alice )

//Cast map to bean?!?!
def map = [name: "Jason", age: 30]
def jason = map as Person

//slicing a collection of beans into a collection of fields
alice.age -= 10 //now alice is 15
assert [15, 40, 30] == [alice, bob, jason].age
```

Operators

Groovy supports operator overloading, by mapping to methods, i.e:

Name	Code	"Equivalent"
Plus	a+b	a.plus(b)
equals	a == b	if (a == null) return b == null else return a.equals(b)
Spread	a*.b	List ret = []; for (it in a) { ret.add(it.b); } return ret;
Safe Navigation	a?.b	(a == null) ? null : b
in	a in b	a.contains(b)
Elvis	a ?: b	a ? a : b (Groovy truth, so null, empty collections/strings, and 0 are false)
Spaceship	a <=> b	a.compareTo(b)
Compare	a < b	a.compareTo(b) < 0
Regex Find	a =~ b	new Pattern(b).matcher(a).matches()
getAt	a[b]	a.getAt(b)
getProperty	a.b	a.getProperty("b")
invokeMethod	a.b()	a.invokeMethod("b")

Operators Example

They may seem simple, but can save A LOT:

```
//"Java" method: Get the age of Bob's parent
int age = -1;
if ( database != null ) {
  Person bob = database.getPerson( "Bob" );
  if ( bob != null ) {
    Person parent = bob.getParent();
    if ( parent != null ) {
      age = parent.getAge();
    }
  }
}

//Groovy method using Safe Navigation and Elvis
age = database.getPerson( "Bob" )?.parent?.age ?: -1;
//If database had a getAt method:
age = database["Bob"]?.parent?.age ?: -1;
```

Dynamic / Duck Typing

```
class File implements Closeable {
  void open() { println "File open" }
  void close() { println "File close" }
  void setText( String text ) { println "File $text" }
}

class DialogBox {
  void open() { println "DialogBox open" }
  void close() { println "DialogBox close" }
  void setText( String text ) { println "DialogBox $text" }
}

//showText works with File or DialogBox, even though they don't
//share an interface or base class
void showText( def destination, String text ) {
  destination.open()
  destination.text = text
  destination.close()
}

showText( new File(), "Text" )
showText( new DialogBox(), "Text" )
```

Dynamic Dispatch

Because Groovy does dynamic dispatch, the method chosen at runtime matches the runtime type, not the compile time type (as in Java):

```
//remember: return keyword is optional
String foo(String arg) { 'String version called' }
String foo(Object o) { 'Object version called' }

Object o = 'a String'
assert 'String version called' == foo( o )
```

Java code would have called the Object version, because the type of o is Object. Groovy eliminates need for "instanceof dispatch" and "double dispatch" patterns.

Closures

In Groovy, code is a first class object:

```
int count = 0;
//Unlike Java, closures can access non-final variables and modify
def counter = { if (it > 10) ++count }

counter( 5 ); counter( 12 ); counter( 15 );
assert 2 == count

//default is single parameter "it", but you can define parameters
def flexCounter = { val, test -> if ( val > test ) ++count }
flexCounter( 10, 0 )
assert 3 == count

//Optionally, parameters can have types:
flexCounter = { int val, int test -> if ( val > test ) ++count }

//If closure is last parameter to a method, it can be outside the ()
[1,2,3].each { count += it }
assert 10 == count;

def tostr = count.&toString //acts like { count.toString() }
assert "10" == tostr()
```

Closures - Extra Features

```
//Memoization: remember results of prior calls
def download = { String url ->
  println "Downloading $url"
  new URL( url ).text }.memoizeAtMost( 5 ) //LRU cache of 5 pages

download "http://gillius.org/" //actually does it
download "http://gillius.org/" //uses cached version

//Curry, remove some closure parameters
class MailSender {
  void sendMail( String from, String to, String subject, String body ) {
    println "From $from, Subject $subject: Dear $to, $body"
  }
}

def sender = new MailSender()
sender.sendMail( "machine", "admin", "Error", "Oh No!" )

//If we call sendMail a lot with the same parameters, make a short version
def sendErrorReport = sender.&sendMail.curry( "machine", "admin", "Error" )
//From machine, Subject Error: Dear admin, The web server is on fire!
sendErrorReport( "The web server is on fire!" )
```

Closures and Dynamic Typing

Using dynamic typing and closures to ensure a safe open/close pattern:

```
void withResource( def closeable, Closure action ) {
  try {
    closeable.open()
    //Set delegate to place closeable's symbols in closure's scope
    action.setDelegate( closeable )
    action() //same as action.call()
  } finally {
    closeable.close()
  }
}

//Use our previous File class with open, close, and setText
withResource( new File() ) {
  //First try X (local variable), X (enclosing scope variable),
  //then this.X, then delegate.X, then owner.X; owner is enclosing class
ref
  text = "Calls setText on the File"
  delegate.text = "Calls setText on the File" //equivalent
}
```

Metaprogramming
(The fun way, at runtime)

Dynamically add/call methods at runtime:

```
class Athlete { def name
                def jump() { "Jump!" } }

//dynamically add methods to classes, all instances:
Object.metaClass.debug = {
  def normalProps = new HashMap( delegate.properties )
  normalProps.remove( "class" ); normalProps.remove( "metaClass" )
  "<${delegate.class.simpleName}: $normalProps>"
}

def athlete = new Athlete( name: "Bob" )
assert "<Athlete: [name:Bob]>" == athlete.debug()

//Call method by string
def methodName = "jump"
assert "Jump!" == athlete."$methodName"()

//What if you needed to call a method that wanted "leap"?
//Dynamically add method to INSTANCE:
athlete.metaClass.leap = { jump() }

assert "Jump!" == athlete.leap()
```

Metaprogramming

(The real-world way; compile-time capable)

Cleaner/Faster way, also how Groovy adds *tons* of methods to standard JDK classes:

```
class AthleteCategory {
  //Adds method to type of first parameter; traditionally named self
  static def debug( Object self ) {
    def normalProps = new HashMap( self.properties )
    normalProps.remove( "class" ); normalProps.remove( "metaClass" )
    return "<${self.class.simpleName}: $normalProps>"
  }
  static Athlete plus( Athlete self, String name ) {
    self.name += name
    return self
  }
}

def athlete = new Athlete( name: "Bob" )
//use affects only the code in its scope
use ( AthleteCategory ) {
  assert "<Athlete: [name:Bob]>" == athlete.debug()
  athlete += "by"; //use our new plus operator
  assert "<Athlete: [name:Bobby]>" == athlete.debug()
}
```

Groovy Method Invocation

Every Groovy call follows the same logic, which enables all of the magic seen thus far:

Given x.method(A,B,C):

- If x does not not extend GroovyObject:
  - "Java" call: MetaClassRegistry.getMetaClass(x.class).invokeMethod("method", A, B, C)
- if x does **not** have method "method", **or** if implements GroovyInterceptable:
  - x.invokeMethod("method", A, B, C)
- else ("GroovyObject" call)
  - x.metaClass.invokeMethod("method", A,B,C)

Domain Specific Languages

Using closures, duck typing, dynamic calling, and metaprogramming, you can build powerful DSLs with Groovy.

```
//Example from Groovy 1.8 Release Notes:
show = { println it }
square_root = { Math.sqrt(it) }

def please(action) {
  [the: { what ->
    [of: { n -> action(what(n)) }]
  }]
}

please show the square_root of 100
//a b c d maps to a(b).c(d):
please(show).the(square_root).of(100)
```

Builders

Builders are possible via the invokeMethod overload:

```
import groovy.xml.MarkupBuilder
def grades = [ CS1: 'A', CS2: 'B', English: "C" ]

def builder = new MarkupBuilder()
builder.html {
  head {
    title "My First Website"
  }
  body {
    p style: "color:blue;", "Here are my grades:"
    ul {
      grades.each {k, v ->
        li "I got an $v in $k"
      }
    }
  }
}
```

Builders

The result of the program on the previous slide:

```
<html>
  <head>
    <title>My First Website</title>
  </head>
  <body>
    <p style='color:blue;'>Here are my grades:</p>
    <ul>
      <li>I got an A in CS1</li>
      <li>I got an B in CS2</li>
      <li>I got an C in English</li>
    </ul>
  </body>
</html>
```

XMLRPC Example

Create XMLRPC server, configure with closures, and call from a client:

```
//Download xmlrpc library and dependencies from Maven Central
@Grab("org.codehaus.groovy:groovy-xmlrpc:0.8")
import groovy.net.xmlrpc.*

//Create Server
def server = new XMLRPCServer()
server.add = {a,b -> a+b} //define functionality

def serverSocket = new ServerSocket(0) //pick any port
server.startServer(serverSocket)

//CLIENT
def serverProxy = new XMLRPCServerProxy(
    "http://localhost:${serverSocket.localPort}" )
assert 11 == serverProxy.add( 5, 6 )

server.stopServer()
```

Real World Groovy

- Dynamic languages rule when handling formats like XML/JSON
- One of the "star" projects of Groovy is Grails, a convention-based Model-View-Controller (MVC) framework for web applications
  - Includes the Grails Object Relational Mapping (GORM), a very powerful database to object mapping
- Griffon: like Grails but with Swing
- Grape: Download libraries on the fly
- Groovy++: Static typing extensions to Groovy to get virtually all Groovy features but performance equal to Java where possible

Groovy Command Line

Java developers may feel more comfortable in Java/Groovy than using UNIX tools like grep, find,cut,awk,sed,etc., or are on Windows. Groovy can do similar tasks at the shell.

Print expensive fruits (price > 3):

```
Prices.csv:
Fruit,Tomato,1
Meat,Chicken,5
Fruit,Banana,6
```

```
groovy -a , -n -e "if (split[0]=='Fruit' && split[2].toInteger() > 2)
println split[1]" Prices.csv
```

Result = Banana

References and Useful Sites

1. Groovy homepage: http://groovy.codehaus.org/
2. Groovy in Action. Koenig et al. 2007 http://manning.com/koenig/ (second edition due Fall 2012)
3. JSR 241: The Groovy Programming Language. http://jcp.org/en/jsr/detail?id=241
4. Domain Specific Languages for unit manipulations: http://groovy.dzone.com/news/domain-specific-language-unit-

Other Languages

There are many languages that can run on the JVM, but there are two that are interesting from a Groovy perspective:

1. Scala, because of its popularity and performance, but mostly because the original creator of Groovy said "I can honestly say if someone had shown me the Programming in Scala book by by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy." Gosling also reportedly endorsed it in 2008.
2. Kotlin, a language in development from JetBrains which has the goals of being "simpler than the most mature competitor – Scala" and includes inspiration from Groovy such as type-safe Groovy-style builders (normaly Groovy builders are dynamically typed).