DZone Refcardz

CONTENTS INCLUDE:

Groovy

By Dierk König

ABOUT GROOVY

Groovy is a dynamic language for the Java™ Virtual Machine (JVM). It shines with full object-orientation, scriptability, optional typing, operator customization, lexical declarations for the most common data types, advanced concepts like closures and ranges, compact property syntax and seamless Java™ integration. This reference card provides exactly the kind of information you are likely to look up when programming Groovy.

STARTING GROOVY

Install Groovy from http://groovy.codehaus.org and you will have the following commands available:

Command	Purpose
groovy	Execute Groovy code
groovyc	Compile Groovy code
groovysh	Open Groovy shell
groovyConsole	Open Groovy UI console
java2groovy	Migration helper

The **groovy** command comes with **-h** and **--help** options to show all options and required arguments. Typical usages are:

Execute file **MyScript.groovy**

```
groovy MyScript
```

Evaluate (e) on the command line

```
groovy -e "print 12.5*Math.PI"
```

Print (p) for each line of input

```
echo 12.5 | groovy -pe
"line.toDouble() *  Math.PI"
```

Inline edit (i) file **data.txt** by reversing each line and save a backup

```
groovy -i.bak –pe
"line.reverse()" data.txt
```

GROOVY/JAVA INTEGRATION

From Groovy, you can call any Java code like you would do from Java. It's identical.

From Java, you can call Groovy code in the following ways. Note that you need to have the **groovy-all.jar** in your classpath.

Cross-compilation

Use groovyc, the **<groovyc/>** ant task or your IDE integration to compile your groovy code together with your Java code. This enables you to use your Groovy code as if it was written in Java.

Eval

Use class **groovy.util.Eval** for evaluating simple code that is captured in a Java String: **(int) Eval**.xyz(1,2,3,"x+y+z");

GroovyShell

Use groovy.util.GroovyShell for more flexibility in the Binding and optional pre-parsing:

```
GroovyShell shell= new GroovyShell();
Script scpt = shell.parse("y = x*x");
Binding binding = new Binding();
scpt.setBinding(binding);
binding.setVariable("x", 2);
scpt.run();
(int) binding.getVariable("y");
```

Chapter 11 of *Groovy in Action* has more details about integration options. Here is an overview:

Integration option	Features/properties
Eval/GroovyShell	for small expressions + reloading, security
GroovyScriptEngine	for dependent scripts + reloading - classes, security
GroovyClassLoader	the catch-all solution + reloading, security
Spring Beans	integrates with Spring + reloading
JSR-223	easy language switch but limited in API - reloading, security requires Java 6

LANGUAGE ELEMENTS

Classes & Scripts

A Groovy class declaration looks like in Java. Default visibility modifier is public

```
class MyClass {
   void myMethod(String argument)  {
   }
}
```

→

Groovy

www.dzone.com

Language Elements (Classes and Scripts), continued

When a .groovy file or any other source of Groovy code contains code that is not enclosed in a class declaration, then this code is considered a Script, e.g.

```
println "Hello World"
```

Scripts differ from classes in that they have a Binding that serves as a container for undeclared references (that are not allowed in classes).

```
println text   //  expected in Binding
result = 1     //  is put into Binding
```

Optional Typing

Static types can be used like in Java and will be obeyed at runtime. **Dynamic** typing is used by replacing the type declaration with the **def** keyword. Formal parameters to method and closure declarations can even omit the **def**.

Properties

Properties are declared as fields with the **default visibility modifier**, no matter what type is used.

```
class MyClass {
   String stringProp
   def    dynamicProp
}
```

Java-style getters and setters are compiled into the bytecode automatically.

Properties are referred to like

```
println obj.stringProp  //  getter
obj.dynamicProp = 1   //  setter
```

regardless of whether **obj** was written in Java or Groovy, the respective getters/setters will be called.

Multimethods

Methods are dispatched by the **runtime** type, allowing code like

```
class Pers {
   String name
   boolean equals(Pers other) {
      name == other.name
   }
}
assert new Pers(name:'x') == new Pers(name:'x')
assert new Pers(name:'x') != 1
```

OPERATORS

Customizable Operators

Operators can be customized by implementing/ overriding the respective method.

Operator	Method
a + b	a.plus(b)
a – b	a.minus(b)
a * b	a.multiply(b)
a / b	a.div(b)
a % b	a.mod(b)
a++ ++a	a.next()

Customizable Operators, continued

Operator	Method
a-- --a	a.previous()
a**b	a.power(b)
a|b	a.or(b)
a&b	a.and(b)
a^b	a.xor(b)
~a	~a a.bitwiseNegate() // sometimes referred to as negate | +a a.positive() // sometimes referred to as unaryMinus | -a a.negative() // sometimes referred to as unaryPlus
a[b]	a.getAt(b)
a[b] = c	a.putAt(b, c)
a << b	a.leftShift(b)
a >> b	a.rightShift(b)
a >>> b	a.rightShiftUnsigned(b)
switch(a){ case b: } [a].grep(b) if(a in b)	b.isCase(a) *// b is a classifier*
a == b	a.equals(b)
a != b	! a.equals(b)
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
a as B	a.asType(B)

Hot Tip

Actively look for opportunities to implement operator methods in your own Groovy class. This often leads to more expressive code. Typical candidates are ==, <=>, +, -, <<, and isCase(). See also ***Ranges***.

Special Operators

Operator	Meaning	Name
a ? b : c	if (a) b else c	ternary if
a ?: b	a ? a : b	Elvis
a?.b	a==null ? a : a.b	null safe
a(*list)	a(list[0], list[1], ...)	spread
list*.a()	[list[0].a(), list[1].a() ...]	spread-dot
a.&b	reference to method b in object a as closure	method closure
a.@field	direct field access	dot-at

SIMPLE DATATYPES

Numbers

All Groovy numbers are objects, not primitive types. Literal declarations are:

Type	Example literals
java.lang.Integer	15, 0x1234ffff
java.lang.Long	100L, 100l
java.lang.Float	1.23f, 4.56F
java.lang.Double	1.23d, 4.56D
java.math.BigInteger	123g, 456G
java.math.BigDecimal	1.23, 4.56, 1.4E4, 2.8e4, 1.23g, 1.23G

Simple Datatypes (Numbers), continued

Coercion rules for math operations are explained in *Groovy in Action*, chapter 3. Some examples to remember are:

Expression	Result type
1f * 2f	Double
1f / 2f	Double
(Byte)1 + (Byte)2	Integer
1 * 2L	Long
1 / 2	BigDecimal (0.5)
(int)(1/2)	Integer (0)
1.intdiv(2)	Integer (0)
Integer.MAX_VALUE+1	Integer
2**31	Integer
2**33	Long
2**3.5	Double
2G + 1G	BigInteger
2.5G + 1G	BigDecimal
1.5G == 1.5F	Boolean (true)
1.1G == 1.1F	Boolean (false)

Strings

```
'literal String'
'''literal
multiline String'''

def lang = 'Groovy'
"GString for $lang"
"$lang has ${lang.size()} chars"

"""multiline GString with
late eval at ${-> new Date()}"""
```

Placeholders in GStrings are dereferenced at declaration time but their text representation is queried at GString→String conversion time.

```
/String with unescaped \ included/
```

Regular Expressions

The regex find operator **=~**
The regex match operator **==~**
The regex *Pattern* operator ***~String***

Examples:

```
  def twister = 'she sells sea shells'
// contains word 'she'
  assert twister =~ 'she'
// starts with 'she' and ends with 'shells'
  assert twister ==~ /she.*shells/
// same precompiled
  def pattern = ~/she.*shells/
  assert pattern.matcher(twister).matches()

// matches are iterable
// words that start with 'sh'
  def shwords = (twister =~ /\bsh\w*/).collect{it}.join(' ')
  assert shwords == 'she shells'

// replace through logic
  assert twister.replaceAll(/\w+/){
        it.size()
    } == '3 5 3 6'

// regex groups to closure params
// find words with same start and end
  def matcher = (twister =~ /(\w)(\w+)\1/)
  matcher.each { full, first, rest ->
      assert full in ['sells','shells']
      assert first == 's'
  }
```

Regular Expressions, continued

Symbol	Meaning
.	any character
^	start of line (or start of document, when in single-line mode)
$	end of line (or end of document, when in single-line mode)
\d	digit character
\D	any character except digits
\s	whitespace character
\S	any character except whitespace
\w	word character
\W	any character except word characters
\b	word boundary
()	grouping
(x|y)	x or y as in (Groovy|Java|Ruby)
\1	backmatch to group one, e.g. find doubled characters with (.)\1
x*	zero or more occurrences of x.
x+	one or more occurrences of x.
x?	zero or one occurrence of x.
x{m,n}	at least "m" and at most "n" occurrences of x.
x{m}	exactly "m" occurrences of x.
[a-f]	character class containing the characters 'a', 'b', 'c', 'd', 'e', 'f'
[^a]	character class containing any character except 'a'
(?is:x)	switches mode when evaluating x; i turns on ignoreCase, s single-line mode
(?=regex)	positive lookahead
(?<=text)	positive lookbehind

COLLECTIVE DATATYPES

Ranges

Ranges appear inclusively like 0..10 or half-exclusively like 0..<10. They are often enclosed in parentheses since the range operator has low precedence.

```
assert (0..10).contains(5)
assert (0.0..10.0).containsWithinBounds(3.5)

for (item in 0..10) { println item }
for (item in 10..0) { println item }
(0..<10).each { println it }
```

Integer ranges are often used for selecting **sublists**. Range boundaries can be of any type that defines **previous()**, **next()** and implements **Comparable**. Notable examples are String and Date.

Lists

Lists look like arrays but are of type **java.util.List** plus new methods.

```
[1,2,3,4]          == (1..4)
[1,2,3] + [1]      == [1,2,3,1]
[1,2,3] << 1       == [1,2,3,1]
[1,2,3,1] - [1]    == [2,3]
[1,2,3] * 2        == [1,2,3,1,2,3]
[1,[2,3]].flatten()  == [1,2,3]
[1,2,3].reverse()    == [3,2,1]
[1,2,3].disjoint([4,5,6])    == true
[1,2,3].intersect([4,3,1])   == [3,1]
[1,2,3].collect{ it+3 }      == [4,5,6]
[1,2,3,1].unique().size()    == 3
[1,2,3,1].count(1)   == 2
[1,2,3,4].min()      == 1
[1,2,3,4].max()      == 4
[1,2,3,4].sum()      == 10
[4,2,1,3].sort()     == [1,2,3,4]
[4,2,1,3].findAll{it%2 == 0} == [4,2]
def anims=['cat','kangaroo','koala']
anims[2] == 'koala'
def kanims = anims[1..2]
anims.findAll{it =~ /k.*/} ==kanims
anims.find{ it =~ /k.*/}   ==kanims[0]
anims.grep(~/k.*/)         ==kanims
```

→

Collective Datatypes, continued

Lists

The sort() method is often used and comes in three flavors:

Sort call	Usage
col.sort()	natural sort for comparable objects
col.sort { it.propname }	applying the closure to each item before comparing the results
col.sort { a,b -> a <=> b }	closure defines a comparator for each comparison

Lists can also be indexed with **negative** indexes and reversed ranges.

```
def list = [0,1,2]
assert list[-1] == 2
assert list[-1..0] == list.reverse()
assert list == [list.head()] + list.tail()
```

Sublist assignments can make a list grow or shrink and lists can contain varying data types.

```
list[1..2] = ['x','y','z']
assert list == [0,'x','y','z']
```

Maps

Maps are like lists that have an arbitrary type of key instead of integer. Therefore, the syntax is very much aligned.

```
def map = [a:0, b:1]
```

Maps can be accessed in a conventional square-bracket syntax or as if the key was a property of the map.

```
assert map['a'] == 0
assert map.b    == 1
map['a'] = 'x'
map.b    = 'y'
assert map == [a:'x', b:'y']
```

There is also an explicit get method that optionally takes a default value.

```
assert map.c == null
assert map.get('c',2) == 2
assert map.c == 2
```

Map iteration methods take the nature of Map.Entry objects into account.

```
map.each { entry ->
   println entry.key
   println entry.value
}
map.each { key, value ->
   println "$key $value"
}
for (entry in map) {
   println "$entry.key $entry.value"
}
```

GPath

Calling a property on a list returns a list of the property for each item in the list.

```
employees.address.town
```

returns a list of town objects.

To do the same with method calls, use the spread-dot operator.

```
employees*.bonus(2008)
```

calls the bonus method on each employee and stores the result in a list.

CLOSURES

Closures capture a piece of logic and the enclosing scope. They are first-class objects and can receive messages, can be returned from method calls, stored in fields, and used as arguments to a method call.

Use in method parameter

```
def forEach(int i, Closure yield){
    for (x in 1..i) yield(x)
}
```

Use as last method argument

```
forEach(3) { num -> println num }
```

Construct and assign to local variable

```
def squareIt = { println it * it}
forEach(3, squareIt)
```

Bind leftmost closure param to fixed argument

```
def multIt = {x, y -> println x * y}
forEach 3, multIt.curry(2)
forEach 3, multIt.curry('-')
```

Closure parameter list examples:

Closure	Parameters
{ ... }	zero or one (implicit 'it')
{-> ... }	zero
{x -> ... }	one
{x=1 -> ... }	one or zero with default
{x,y -> ... }	two
{ String x -> ... }	one with static type

Closure.isCase(b) sends b to the closure and returns the call result as boolean. Use as in

```
switch ('xy'){
    case {it.startsWith('x')} :...
}
[0,1,2].grep { it%2 == 0 }
```

GDK

Methods for java.lang.Object

Get object info

```
println obj.dump()
```

or in a GUI

```
import groovy.inspect.swingui.*
ObjectBrowser.inspect(obj)
```

Print properties, methods, and fields of obj

```
println obj.properties
println obj.class.methods.name
println obj.class.fields.name
```

Two ways to invoke a method dynamically

```
obj.invokeMethod(name, paramsAry)
obj."$name"(params)
```

GDK

GDK (Methods for java.lang.Object). continued

Further methods

```
is(other) // identity check
isCase(candidate) //default:equality
obj.identity {...}; obj.with {...}
print();   print(value),
println(); println(value)
printf(formatStr, value)
printf(formatStr, value[])
sleep(millis)
sleep(millis) { onInterrupt }
use(categoryClass) { ... }
use(categoryClassList) { ... }
```

Every object is *iterable* in Groovy—even if it was implemented in Java. See *Groovy in Action*, chapter 9 on what strategy Groovy applies to make this happen.

Not only can you use any obj in loops like

```
for (element in obj) { ... }
```

but you can also apply the following iterative objects methods:

Returns	Purpose
Boolean	any {...}
List	collect {...}
Collection	collect(Collection collection) {...}
(void)	each {...}
(void)	eachWithIndex {item, index-> ...}
Boolean	every {...}
Object	find {...}
List	findAll {...}
Integer	findIndexOf {...}
Integer	findIndexOf(startIndex) {...}
Integer	findLastIndexOf {...}
Integer	findLastIndexOf(startIndex) {...}
List	findIndexValues {...}
List	findIndexValues(startIndex) {...}
Object	inject(startValue) {temp, item -> ...}
List	grep(Object classifier) // uses classifier.isCase(item)

Hot Tip Implement the iterator() method that returns an Iterator object to give your own Groovy class meaningful ***iterable*** behavior with the above methods.

Files and I/0

Often-used filesystem methods

```
def dir = new File('somedir')
def cl  = {File f -> println f}
dir.eachDir cl
dir.eachFile cl
dir.eachDirRecurse cl
dir.eachFileRecurse cl
dir.eachDirMatch(~/.*/, cl)
dir.eachFileMatch(~/.*/, cl)
```

Files and I/0, continued

Often used reading methods

```
def file = new File('/data.txt')
println file.text
```

(also for Reader, URL, InputStream,Process)

```
def listOfLines = file.readLines()
file.eachLine { line -> ... }
file.splitEachLine(/\s/) { list -> }
```

```
file.withReader  { reader -> ... }
```

(also for Reader, URL, InputStream)

```
file.withInputStream { is -> ...}
```

(also for URL)

Often-used writing methods

```
out << 'content'
```

for out of type File, Writer, OutputStream, Socket, and Process

```
file.withWriter('ASCII') {writer -> }
file.withWriterAppend('ISO8859-1'){
    writer ->  ... }
```

Reading and writing with Strings

```
def out = new StringWriter()
out << 'something'
def str = out.toString()
def rdr = new StringReader(str)
println rdr.readLines()
```

Connecting readers and writers

```
writer << reader
```

Special logic for writable objects, e.g. writeTo()

```
writer << obj
```

Transform (with closure returning the replacement) and filter (with closure returning boolean)

```
reader.transformChar(writer){c -> }
reader.transformLine(writer){line-> }
src.filterLine(writer){line-> }
writer << src.filterLine {line -> }
```

For src in File, Reader, InputStream

Threads & Processes

Two ways of spawning new threads

```
def thread = Thread.start { ... }
def t = Thread.startDaemon { ... }
```

Two ways of talking to an external process ('cmd /c' is for Windows platforms only)

```
today = 'cmd /c date /t'
        .execute().text.split(/\D/)
proc = ['cmd','/c','date']
        .execute()
Thread.start {System.out << proc.in}
Thread.start {System.err << proc.err}
proc << 'no-such-date'  + "\n"
proc << today.join('-') + "\n"
proc.out.close()
proc.waitForOrKill(0)
```

XML

Reading XML

Decide to use the parser (for state-based processing) or the slurper (for flow-based processing)

```
def parser = new XmlParser()
def slurper = new XmlSlurper()
```

Common parse methods:

parse(org.xml.saxInputSource)
parse(File)
parse(InputStream)
parse(Reader)
parse(String uri)
parseText(String text)

The parse methods of parser and slurper return different objects (Node vs. GPathResult) but you can apply the following methods on both:

```
result.name()
result.text()
result.toString()
result.parent()
result.children()
result.attributes()
result.depthFirst()
result.iterator() // see GDK hot tip
```

Shorthands for children, child, and attribute access:

Shorthand	Result
['elementName']	All child elements of that name
.elementName	
[index]	Child element by index
['@attributeName']	The attribute value stored under that name
.'@attributeName'	
.@attributeName	

Reading the first ten titles from a blog:

```
def url= 'http://'+
  'www.groovyblogs.org/feed/rss'
def rss = new XmlParser().parse(url)
rss.channel.item.title[0..9]*.text()
```

Writing XML

Groovy (Streaming-) MarkupBuilder allows you to produce proper XML with logic while keeping a declarative style.

```
def b=new groovy.xml.MarkupBuilder()
b.outermost {
  simple()
    'with-attr' a:1, b:'x', 'content'
    10.times { count ->
       nesting { nested count }
    }
}
```

SQL

Connecting to the DB

Getting a new Sql instance directly. For example, a HSQLDB

```
import groovy.sql.Sql
def db = Sql.newInstance(
  'jdbc:hsqldb:mem:GInA',
  'user-name',
  'password',
  'org.hsqldb.jdbcDriver')
```

SQL, continued

Alternative with using a datasource

```
import org.hsqldb.jdbc.*
def source = new jdbcDataSource()
source.database  = 'jdbc:hsqldb:mem:GInA'
source.user      = 'user-name'
source.password  = 'password'
def db  = new groovy.sql.Sql(source)
```

Submitting Queries

When a query contains wildcards, it is wise to use a PreparedStatement. Groovy SQL does this automatically when you supply either the list of values in an extra list or when the statement is a GString. So each method below has three variants:

```
method('SELECT ... ')
method('SELECT ...?,?', [x,y])
method("SELECT ... $x,$y")
```

Returns	Method name	Parameters
boolean	execute	prepStmt
Integer	executeUpdate	prepStmt
void	eachRow	prepStmt { row -> }
void	query	prepStmt { resultSet -> ... }
List	rows	prepStmt
Object	firstRow	prepStmt

In the above, attributes can be fetched from each row by index or by name

```
db.eachRow('SELECT a,b ...'){ row ->
  println row[0] + ' ' + row.b
}
```

Combine with GPath

```
List hits = db.rows('SELECT ...')
hits.grep{it.a > 0}
```

DataSet

For easy DB operations without SQL

```
def dataSet = db.dataSet(tablename)
dataSet.add (
  a: 1,
  b: 'something'
)
dataSet.each { println it.a }
dataSet.findAll { it.a < 2 }
```

In the last statement, the expression in the findAll closure will map directly to a SQL WHERE clause.

META PROGRAMMING

Categories

Group of methods assigned at runtime to arbitrary classes that fulfill a common purpose. Applies to one thread. Scope is limited to a closure.

```
class IntCodec {
  static String encode(Integer self){self.toString()}
  static Integer decode(String self){self.toInteger()}
}
use(IntCodec) {42.encode().decode()}
```

ExpandoMetaClass

Same example but change applies to all threads and unlimited scope.

```
Integer.metaClass.encode << {delegate.toString()}
String.metaClass.decode << {delegate.toInteger()}
42.encode().decode()
```

Meta Programming, continued

Method Invocation Hooks

In your Groovy class, implement the method

```
Object invokeMethod(String name, Object args)
```

to intercept calls to unavailable methods.

Additionally, implement the interface **GroovyInterceptable** to intercept also calls to available methods.

Implement

```
Object   getProperty(String name)
void     setProperty(
             String name, Object value)
```

to intercept property access.

A bit easier to handle are the variants

```
Object methodMissing(
      String name, Object args)
Object propertyMissing(
      String name, Object args)
```

that are called like the name suggests.

Instead of implementing the above methods, they can also be added to the MetaClass of any arbitrary class (or object) to achieve the same effect.

```
Integer.metaClass.methodMissing << {
   String name, Object args ->
   Math."$name"(delegate)
}
println 3.sin()
println 3.cos()
```

ABOUT THE AUTHOR

Dierk König

Dierk is a committer to the Groovy and Grails project since its early days and lead author of the renowned ***Gina*** *(Groovy in Action)* book. He works for Canoo Engineering AG in Basel, Switzerland, as a software developer, mentor, and coach. He enjoys his daily hands-on work in software projects as well as publishing in leading magazines and speaking at international conferences.

Dierk holds degrees in both business administration and computer science, and has worked as a professional Java programmer for over 10 years while always keeping an eye on evolving languages. He is an acknowledged reviewer and/or contributor to numerous leading books on the topics of Extreme Programming, Test-Driven Development, Groovy, and Grails. His strong focus on delivering quality software led him to founding the open-source Canoo WebTest project and managing it since 2001.

RECOMMENDED BOOK

Groovy in Action introduces Groovy by example, presenting lots of reusable code while explaining the underlying concepts. Java developers new to Groovy find a smooth transition into the dynamic programming world. Groovy experts gain a solid reference that challenges them to explore Groovy deeply and creatively.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-09-7
ISBN-10: 1-934238-09-0
50795
9 781934 238097
$7.95

Version 1.0