FREE eBook

LEARNING groovy

Free unaffiliated eBook created from
Stack Overflow contributors.

#groovy

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: groovy

It is an unofficial and free groovy ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official groovy.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with groovy

Remarks

Groovy is

- is an optionally typed dynamic language for the Java Virtual Machine
- builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby, and Smalltalk
- makes modern programming features available to Java developers with an almost-zero learning curve
- provides the ability to statically type check and statically compile your code for robustness and performance
- supports Domain-Specific Languages and other compact syntax so your code is easy to read and maintain
- makes writing shell and build scripts easy with its powerful processing primitives, OO abilities, and an Ant DSL
- increases developer productivity by reducing scaffolding code when developing web, GUI, database or console applications
- simplifies testing by supporting unit testing and mocking out-of-the-box
- seamlessly integrates with all existing Java classes and libraries
- compiles straight to Java bytecode so you can use it anywhere you use Java

Versions

Version	Release Notes	Release Date
2.4	http://groovy-lang.org/releasenotes/groovy-2.4.html	2015-01-21
2.3	http://groovy-lang.org/releasenotes/groovy-2.3.html	2014-05-05
2.2	http://groovy-lang.org/releasenotes/groovy-2.2.html	2013-11-18
2.1	http://groovy-lang.org/releasenotes/groovy-2.1.html	2013-01-24
2.0	http://groovy-lang.org/releasenotes/groovy-2.0.html	2012-06-28
1.8	http://groovy-lang.org/releasenotes/groovy-1.8.html	2011-04-27

Version	Release Notes	Release Date
1.7	http://groovy-lang.org/releasenotes/groovy-1.7.html	2009-12-22
1.6	http://groovy-lang.org/releasenotes/groovy-1.6.html	2009-02-18
1.5	http://groovy-lang.org/releasenotes/groovy-1.5.html	2007-12-07
1.0		2007-01-02

Examples

Installation or Setup

There are two common ways to install Groovy.

Download

The Groovy binary can be downloaded on the download page of the Groovy website. You can unpack archive and add path to `%GROOVY_HOME%/bin/groovy.bat` to the PATH system environment variable, where %GROOVY_HOME% is the directory where Groovy is unpacked.

SDKMAN

The other option is to use SDKMAN. This option has grown quickly in popularity, and makes managing multiple versions of Groovy very simple. It also supports other applications in the "GR8" ecosphere. This option works very well natively on Linux and Mac, but requires Cygwin on Windows.

Following the instructions on the Groovy download page, you can take the following steps to install SDKMAN.

```
$ curl -s get.sdkman.io | bash
```

Once SDKMAN is installed, you now have access to the `sdk` command. With this command you can do many useful things.

Install Groovy

```
$ sdk install groovy
```

This will install the latest version of Groovy.

List versions of Groovy

```
$ sdk ls groovy
```

This allows you to run a Linux style `ls` command on the Groovy software, listing all of the available options. There is an `*` next to each installed version, and a `>` to indicate your current versions.

Switch versions of Groovy

```
$ sdk use groovy 2.4.7
```

This will change the current version of Groovy to 2.4.7. If you have other versions installed, you can switch to any of those.

You can list the current version of groovy with the `groovy -version` command.

posh-gvm

The initial name of SDKMAN was GVM and posh-gvm is a port of GVM for the Windows Powershell. So, if you develop on a Windows machine and don't want to use SDKMAN on Cygwin, posh-gvm is for you. It works the same as SDKMAN, but instead of `sdk`, the command is `gmv`. So

```
PS C:\Users\You> gmv install groovy
```

will install groovy through posh-gvm on your Windows machine.

Hello World

The Groovy version of Hello World.

```
println 'Hello World!'
```

Hello World In groovy

Following example illustrate the simplest `Hello World` in groovy using script, place the following code snippet in a file, say `helloWorld.groovy`

```
println 'Hello World!'
```

How to execute: In the command line, `groovy helloWorld.groovy`

Output: `Hello World!`

Using Groovy on a Java project

Groovy has access to all java classes, in fact Groovy classes ARE Java classes and can be run by the JVM directly. If you are working on a Java project, using Groovy as a simple scripting language to interact with your java code is a no-brainer.

To make things even better, nearly any Java class can be renamed to .groovy and compiled and run and will work exactly as it did, groovy is close to being a super-set of Java, this is a stated goal of groovy.

Groovy has a REPL. `groovysh` comes with Groovy and can be used to quickly instantiate and test a Java class if your classpath is set up correctly. For instance if your `classpath` pointed to your eclipse "classes/bin" directory, then you could save your file in eclipse, jump to `groovysh` and instantiate the class to test it.

The reasons to use Groovy to do this instead of just Java are: The classloader is GREAT at picking up new classes as they are compiled. You don't generally need to exit/re-start `groovysh` as you develop.

The syntax is TERSE. This isn't great for maintainable code, but for scripts and tests it can cut your code significantly. One of the big things it does is eliminate checked exceptions (or, more accurately, turn all checked exceptions into unchecked exceptions). This turns code like this (Print hello after one second):

```
class JavaClass {
    public static void main(String[] args) {
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            // You shouldn't leave an empty catch block, but who cares if this was
interrupted???
        }
        System.out.println("Hello!");
    }
}
```

into Groovy's:

```
Thread.sleep(1000)
print "Hello!"
```

Groovy also has very tight initialization syntax. This allows you to specify data just as you like it without thinking about it:

In Java to initialize a map you should probably do something like this:

```
String[] init = { "1:Bill", "2:Doug", "3:Bev" };
// Note the rest of this can be put in a function and reused or maybe found in a library, but
I always seem to have to write this function!
Map m = new HashMap<Integer, String>();
for(String pair : int) {
    String[] split = pair.split(":");
    m.put(new Integer(split[0]), split[1])
}
```

This isn't bad, but it's something else to maintain. In groovy you would just use:

```
Map map = { 1 : "Bill", 2 : "Doug", 3 : "Bev" }
```

And you are done. List syntax is just as easy.

The other really big advantage is groovy's closure syntax. It's amazingly terse and fun, somewhat more difficult to maintain, but for scripts that's not a priority. As an example, here is some groovy code to find all `.txt` files that contain the word `Hello` in the current directory:

```
println new File('.').files.findAll{ it.name.endsWith('.txt') && it.text.contains('Hello')
}.collect{ it.name }
```

This example uses a few "Groovy" tricks:

- `.files` refers to the `getFiles()` method - groovy can switch between getter/setter and property syntax at will
- `it.` refers to the current element of an iteration. `{ it }` is a shortcut for `{ it -> it }`, e.g. :

  [1, 2, 3].collect{ it ^ 2 } == [1, 4, 9]
- `it.text` (where `it` is a file) uses a method groovy adds to `File` to retrieve the entire text of the file. This is amazingly helpful in scripts.

Hello world Shebang (linux)

Given a hello.groovy file with content:

```
#!/usr/bin/env groovy
println "Hello world"
```

Can be executed from the command line if given execution permission as

```
$ ./hello.groovy
```

Using inject() On List To Create CSV String

In Groovy, the inject() method is one of the cumulative methods that allows us to add (or inject) new functionality into any object that implements the inject() method. In the case of a Collection, we can apply a closure to a collection of objects uniformly and then collate the results into a single value. The first parameter to the inject() method is the initial value of the cumulation and the second parameter is the closure.

In this example, we will take a List of Strings as a parameter and output the values of those strings delimited by commas. I have used this functionality to append a list of values to a REST query string and, if you modify it a bit, I've used it to include values into a SQL statement as part of a IN clause. Here is the code to do this:

```
public String convertToCSV( List<String> list ) {
    if (list == null) {
        return ""
    }
    return list.inject( '' ) { result, item ->
        result + ( result && item ? ',' : '' ) + ( item ? "${item.trim()}" : '' )
    }
}

assert convertToCSV( null ) == ""
assert convertToCSV( ["aaa", "bbb  ", null, "  ccc  "] ) == "aaa,bbb,ccc"
```

In this example, the first parameter to the inject() method is a zero length string, which means that when processing the first element of the list, result is also a zero length string. This resolves to false in the first ternary evaluation which is why we don't get a comma at the beginning of the string. With each consecutive iteration through the elements of the list, result becomes the concatenation of itself, a comma and then the next item until we reach the last item in the list.

The advantage of this approach is that you don't need a variable outside of a looping construct to hold the concatenated String result. The implication being that this can lead to side effects in your code. With the inject() approach, this behavior is injected and the collection collates the result of the calls to the closure for you. The downside of this approach can be readability. But with some experience, it becomes easier to read and understand, and I hope this example helps you obtain that goal.

Read Getting started with groovy online: https://riptutorial.com/groovy/topic/966/getting-started-with-groovy

Chapter 2: AST Transformations

Examples

@CompileStatic

Enables a code to be statically compiled. Its bytecode will be closer to Java's, thus having better performance, though some dynamic features won't be available.

```
@groovy.transform.CompileStatic
class ListMath {
    def countSize(List<String> strings) {
        strings.collect { it.size() }.sum()
    }
}


assert new ListMath().countSize(["a", "bb", "ccc"]) == 6
```

Read AST Transformations online: https://riptutorial.com/groovy/topic/4635/ast-transformations

Chapter 3: Closure Memoize Methods

Syntax

- closure.memoize()
- closure.memoizeAtMost(n)
- closure.memoizeAtLeast(n)
- closure.memoizeBetween(n, m)

Remarks

Memoization is a method of caching the result of a closure invocation. The memoize function applied to a closure returns a new closure whose return value is cached according to its input parameters. The caches used for the three tweaked variants of memoization methods are LRU caches, that is the least recently used element is removed from the cache first.

Examples

Simple memoization

```
def count = 0

nonmemoized = { long n -> println "nonmemoized: $n"; count++ }

nonmemoized(1)
nonmemoized(2)
nonmemoized(2)
nonmemoized(1)
assert count == 4


def mcount = 0

memoized = { long n -> println "memoized: $n"; mcount++ }.memoize()

memoized(1)
memoized(2)
memoized(2)
memoized(1)
assert mcount == 2
```

Read Closure Memoize Methods online: https://riptutorial.com/groovy/topic/6308/closure-memoize-methods

Chapter 4: Closures

Examples

Closure with explicit parameters

```
def addNumbers = { a, b -> a + b }
addNumbers(-7, 15) // returns 8
```

Closure with implicit parameters

```
['cat', 'dog', 'fish'].collect { it.length() }
```

`it` is the default name of the parameter if you have a single parameter and do not explicitly name the parameter. You can optionally declare the parameter as well.

```
['cat', 'dog', 'fish'].collect { animal -> animal.length() }
```

Converting Methods to Closures

A method can be converted to a closure using the **&** operator.

```
def add(def a, def b) { a + b }

Closure addClosure = this.&add
assert this.add(4, 5) == addClosure(4, 5)
```

Closure with custom target for method calls with implicit receiver

```
class MyHello {
  def sayHello() {
    "Hello, world"
  }
}

def cl = { sayHello() }
cl() // groovy.lang.MissingMethodException
cl.delegate = new MyHello()
cl(); // "Hello, world"
```

Used extensively by Groovy DSLs.

Wrapping behavior around a closure with a method

There are frequent behavior patterns that can result in a lot of boilerplate code. By declaring a method that takes a `Closure` as a parameter, you can simplify your program. As an example, it is a common pattern to retrieve a database connection, start a transaction, do work, and then either commit the transaction, or rollback the connection (in case of error), then finally close the connection:

```
def withConnection( String url, String user, String pass, Closure closure) {
    Connection conn = null
    try {
        conn = DriverManager.getConnection( url, user, pass )
        closure.call( conn )
        conn.commit()
    } catch (Exception e) {
        log.error( "DB Action failed", e)
        conn.rollback()
    } finally {
        conn?.close()
    }
}


withConnection( DB_PATH, DB_USER, DB_PASS ) { Connection conn ->
    def statement = conn.createStatement()
    def results = statement.executeQuery( 'SELECT * FROM users' )
    // ... more processing ...
}
```

Create closures, assign to properties and call

Let's create a map and a closure to print hello

```
def exMap = [:]

def exClosure = {
    println "Hello"
}
```

Assign closure to a property in map

```
exMap.closureProp = exClosure
```

Calling closure

```
exMap.closureProp.call()
```

Output

```
Hello
```

Another Example - Lets create a class with basic property and assign same closure to object of it

```
class Employee {
    def prop
}
```

```
def employee = new Employee()

employee.prop = exClosure
```

Call closure through that property

```
employee.prop.call()
```

Output

```
Hello
```

Read Closures online: https://riptutorial.com/groovy/topic/2684/closures

Chapter 5: Collection Operators

Examples

Iterate over a collection

Lists

```
def lst = ['foo', 'bar', 'baz']
// using implicit argument
lst.each { println it }

// using explicit argument
lst.each { val -> println val }

// both print:
// foo
// bar
// baz
```

Iterate with index

```
def lst = ['foo', 'bar', 'baz']
// explicit arguments are required
lst.eachWithIndex { val, idx -> println "$val in position $idx" }

// prints:
// foo in position 0
// bar in position 1
// baz in position 2
```

Maps

```
def map = [foo: 'FOO', bar: 'BAR', baz: 'BAZ']

// using implicit argument
map.each { println "key: ${it.key}, value: ${it.value}"}

// using explicit arguments
map.each { k, v -> println "key: $k, value: $v"}

// both print:
// key: foo, value: FOO
// key: bar, value: BAR
// key: baz, value: BAZ
```

Create a new list using collect

```
def lst = ['foo', 'bar', 'baz']
lst.collect { it } // ['foo', 'bar', 'baz']

lst.collect { it.toUpperCase() } // ['FOO', 'BAR', 'BAZ']
```

To collect keys or values from a maps

```
def map = [foo: 'FOO', bar: 'BAR', baz: 'BAZ']
def keys = map.collect { it.key } // ['foo', 'bar', 'baz']
def vals = map.collect { it.value } // ['FOO', 'BAR', 'BAZ']
```

The above example is equivalent to calling `map.keySet()` and `map.values()`

Filter a list with findAll

```
def lst = [10, 20, 30, 40]

lst.findAll { it > 25 } // [30, 40]
```

Find the first element matching a condition

```
def lst = [10, 20, 30, 40]

lst.find { it > 25 } // 30. Note: it returns a single value
```

Create maps with collectEntries

From lists

```
def lst = ['foo', 'bar', 'baz']

// for each entry return a list containing [key, value]
lst.collectEntries { [it, it.toUpperCase()] } // [foo: FOO, bar: BAR, baz: BAZ]

// another option, return a map containing the single entry
lst.collectEntries { [(it): it.toUpperCase()] } // [foo: FOO, bar: BAR, baz: BAZ]
```

From maps

```
def map = [foo: 'FOO', bar: 'BAR', baz: 'BAZ']

map.collectEntries { [it.key*2, it.value*2] } // [foofoo: FOOFOO, barbar: BARBAR, bazbaz:
BAZBAZ]

// using explicit arguments k and v
map.collectEntries { k, v -> [k*2, v*2] } // [foofoo: FOOFOO, barbar: BARBAR, bazbaz: BAZBAZ]
```

Apply transformation to nested collections

Apply the transformation to non-collection entries, delving into nested collections too and preserving the whole structure.

```
def lst = ['foo', 'bar', ['inner_foo', 'inner_bar']]

lst.collectNested { it.toUpperCase() } // [FOO, BAR, [INNER_FOO, INNER_BAR]]
```

Flatten a nested list

```
def lst = ['foo', 'bar', ['inner_foo', 'inner_bar']]

lst.flatten() // ['foo', 'bar', 'inner_foo', 'inner_bar']
```

Remove duplicates

```
def lst = ['foo', 'foo', 'bar', 'baz']

// *modifies* the list removing duplicate items
lst.unique() // [foo, bar, baz]

// setting to false the "mutate" argument returns a new list, leaving the original intact
lst.unique(false) // [foo, bar, baz]

// convert the list to a Set, thus removing duplicates
lst.toSet() // [baz, bar, foo]

// defining a custom equality criteria. For example: to elements are equal if have the same
first letter
println lst.unique() { it[0] } // [foo, bar]. 'bar' and 'baz' considered equal
```

Build a map from two lists

```
nrs = [1, 2, 3, 4, 5, 6, 7, 8, 9]
lets = ['a', 'b', 'c', 'd', 'e', 'f']

println GroovyCollections.transpose([nrs, lets])
        .collect {le -> [(le[0]):le[1]]}.collectEntries { it }

or

println [nrs,lets].transpose().collectEntries{[it[0],it[1]]}

// [1:a, 2:b, 3:c, 4:d, 5:e, 6:f]
```

Read Collection Operators online: https://riptutorial.com/groovy/topic/5103/collection-operators

Chapter 6: Currying

Syntax

- closure.curry(parameter)
- closure.rcurry(parameter)
- closure.ncurry(index, parameters ...)

Remarks

- Currying a closure produces a new closure with one or more of it's parameters having a fixed value
- Left or right currying a closure that has no parameters or index based currying a closure that has less than two parameters throws an `IllegalArgumentException`

Examples

Left currying

```
def pow = { base, exponent ->
    base ** exponent
}
assert pow(3, 2) == 9

def pow2 = pow.curry(2) //base == 2
assert pow2(3) == 8
```

Right currying

```
def dividable = { a, b ->
    a % b == 0
}
assert dividable(2, 3) == false
assert dividable(4, 2) == true

def even = dividable.rcurry(2) // b == 2
assert even(2) == true
assert even(3) == false
```

Index based currying

```
def quatNorm = { a, b, c, d ->
    Math.sqrt(a*a + b*b + c*c + d*d)
}
assert quatNorm(1, 4, 4, -4) == 7.0

def complexNorm = quatNorm.ncurry(1, 0, 0) // b, c == 0
```

```
assert complexNorm(3, 4) == 5.0
```

Currying closure with no explicit parameter

```
  def noParam = {
      "I have $it"
  }

  def noParamCurry = noParam.curry(2)
  assert noParamCurry() == 'I have 2'
```

Currying closure with no parameters

```
def honestlyNoParam = { ->
    "I Don't have it"
}

// The following all throw IllegalArgumentException
honestlyNoParam.curry('whatever')
honestlyNoParam.rcurry('whatever')
honestlyNoParam.ncurry(0, 'whatever')
```

Read Currying online: https://riptutorial.com/groovy/topic/4400/currying

Chapter 7: Domain Specific Languages

Examples

Language capabilities

The Jenkins Pipeline DSL is used as an example for such a language:

```
node {
  git 'https://github.com/joe_user/simple-maven-project-with-tests.git'
  def mvnHome = tool 'M3'
  sh "${mvnHome}/bin/mvn -B -Dmaven.test.failure.ignore verify"
  archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true
  junit '**/target/surefire-reports/TEST-*.xml'
 }
```

The purpose of this DSL is the define and execute Jenkins build jobs (or better pipelines) in a more natural language.

Writing a domain specific language in Groovy benefits by Groovy's core features like:

- Optionality (e.g. omit parentheses)
- Operator overloading
- Meta programming (e.g. resolving missing properties or methods)
- Closures and delegation strategies
- Compiler customization
- Scripting support and integration capabilities

Read Domain Specific Languages online: https://riptutorial.com/groovy/topic/5948/domain-specific-languages

Chapter 8: Groovy code golfing

Introduction

Tips for golfing in Groovy

Examples

Spread dot operator(*.)

Spread dot operator can be used instead of collect method

```
(1..10)*.multiply(2) // equivalent to (1..10).collect{ it *2 }
d = ["hello", "world"]
d*.size() // d.collect{ it.size() }
```

Parallel processing using Gpars

Gpars offers intuitive ways to handle tasks concurrently

```
import groovyx.gpars.*
GParsPool.withPool { def result = dataList.collectParallel { processItem(it) } }
```

Read Groovy code golfing online: https://riptutorial.com/groovy/topic/10651/groovy-code-golfing

Chapter 9: Groovy Truth (true-ness)

Remarks

Groovy evaluates conditions in **if**, **while** and **for** statements **the same way as Java does for standard Java conditions** : in Java you must provide a boolean expression (an expression that evaluates to a boolean) and the result is the result of the evaluation.

In Groovy , the result is the same as in Java for thoses conditions (no examples provided, this is standard Java).

The other **truthfulness evaluation mechanism** shown by examples can be summarized as:

- numbers: a zero value evaluates to false, non zero to true.
- objects: a null object reference evaluates to false, a non null reference to true.
- Character : a character with a zero value evaluates to false, true otherwise.
- String : a string evaluates to true if not null and not empty, false if null or empty (applies to GStrings and CharSequences too).
- Collections and Maps (including subclasses **List**, **Map**, **Set**, **HashSet** ...) : also takes into account the size, evaluates to true if the collection is not null and not empty, false if null or empty.
- Enumerations and Iterators evaluates to true if not null and they are some more elements (groovy evaluates **hasMoreElements** or **hasNext** on the object), false if null or no more elements.
- Matcher : a matcher evaluates to true if there is at least one match, false if not match is found.
- Closure : a closure evaluates to the evaluation of the result returned by the closure.

The asBoolean method can be overriden in a user defined class to provide custom boolean evaluation.

Examples

Numbers boolean evaluation

for numbers, a zero value evaluates to false, non zero to true

```
    int i = 0
...
    if (i)
        print "some ${i}"
    else
        print "nothing"
```

will print "nothing"

Strings boolean evaluation

a string (including GStrings) evaluates to true if not null and not empty, false if null or empty

```
def s = ''
...
if (s)
    println 's is not empty'
else
    println 's is empty'
```

will print: 's is empty'

Collections and maps boolean evaluation

Collections and Maps evaluates to true if not null and not empty and false if null or empty

```
/* an empty map example*/
def userInfo = [:]
if (!userInfo)
    userInfo << ['user': 'Groot', 'species' : 'unknown' ]
```

will add `user: 'Groot' , species : 'unknown'` as default userInfo since the userInfo map is empty (note that the map is not null here)

With a null object, the code is lightly different, we cannot invoke << on userInfo because it is null, we have to make an assignment (refer also to Object boolean evaluation):

```
/* an example with a null object (def does not implies Map type) */
def userInfo = null
if (!userInfo)
    userInfo = ['user': 'Groot', 'species' : 'unknown' ]
```

And with a null Map:

```
/* The same example with a null Map */
Map<String,String> userInfo = null
if (!userInfo)
    userInfo = ['user': 'Groot', 'species' : 'unknown' ]
```

Object boolean evaluation

a null object reference evaluates to false, a non null reference to true, but for for strings, collections, iterators and enumerations it also takes into account the size.

```
def m = null

if (!m)
    println "empty"
else
```

```
    println "${m}"
```

will print "empty"

```
def m = [:]

if (!m)
    println "empty"
else
    println "${m}"
```

The map is not null but empty, this code will print "empty"

After doing

```
m << ['user' : 'Groot' ]
```

it will print the Map:

```
[user:Groot]
```

Overriding boolean evaluation in a user defined class

Sometimes it may be useful to have a specific asBoolean definition in your own program for some kind of objects.

```
/** an oversimplified robot controller */
class RunController {

    def complexCondition
    int position = 0

    def asBoolean() {
        return complexCondition(this);
    }
    def advanceTo(step) {
        position += step
    }
}
def runController = new RunController(complexCondition : { c -> c.position < 10 } )

assert runController
runController.advanceTo(5)
assert runController
runController.advanceTo(5)
// The limit has been reached : the controller evaluates to false
assert !runController
```

This code shows an oversimplifed robot controller who checks that the position of the robot does not exceeds 10 (with a closure for condition evaluation)

Character evaluation

a Character evaluates to true if it's value is not zero, false if zero

```
assert ! new Character((char)0)
assert ! new Character('\u0000Hello Zero Char'.charAt(0))
assert  new Character('Hello'.charAt(0))
```

Matcher evaluation

a Matcher evaluates to true if it can find at least one match, false if no match is found

```
// a match is found => true
assert 'foo' =~ /[a-z]/
// the regexp does not match fully => no match => false
assert !( 'foo' ==~ /[a-z]/ )
// a match is found => true
assert 'foo' =~ /o/
// no match => false
assert !( 'foo' =~ /[A-Z]/ )
```

Closure evaluation

The evaluation of a closure is the evaluation of the result of the closure.

All rules applies : if the closure returns a null , zero number or empty String, Collection, Map or Array it evaluates to false otherwise to true.

```
// Closure return non zero number => true
assert { 42 }()
// closure returns 0 => false
assert ! ( { 0 }())
// closure returns null => false
assert !( { }())
```

Read Groovy Truth (true-ness) online: https://riptutorial.com/groovy/topic/5117/groovy-truth--true-ness-

Chapter 10: JSON

Examples

Parse a json string

```
import groovy.json.JsonSlurper;

def jsonSlurper = new JsonSlurper()
def obj = jsonSlurper.parseText('{ "foo": "bar", "baz": [1] }')

assert obj.foo == 'bar'
assert obj.baz == [1]
```

Parse a json file

```
import groovy.json.JsonSlurper;

def jsonSlurper = new JsonSlurper()

File fl = new File('/path/to/fils.json')

// parse(File file) method is available since 2.2.0
def obj = jsonSlurper.parse(fl)

// for versions < 2.2.0 it's possible to use
def old = jsonSlurper.parse(fl.text)
```

Write a json to string

```
import groovy.json.JsonOutput;

def json = JsonOutput.toJson([foo: 'bar', baz: [1]])

assert json == '{"foo":"bar","baz":[1]}'
```

In addition to maps, lists and primitives `groovy.json.JsonOutput` also supports a *POJOs* serialitzation:

```
import groovy.json.JsonOutput;

class Tree {
    def name
    def type
}

Tree willow = new Tree(name:'Willow',type:'Deciduous')
Tree olive = new Tree(name:'Olive',type:'Evergreen')

assert JsonOutput.toJson(willow) == '{"type":"Deciduous","name":"Willow"}'
assert JsonOutput.toJson([willow,olive]) ==
'[{"type":"Deciduous","name":"Willow"},{"type":"Evergreen","name":"Olive"}]'
```

Pretty-print a json string

```
import groovy.json.JsonOutput;

def json = JsonOutput.toJson([foo: 'bar', baz: [1]])

assert json == '{"foo":"bar","baz":[1]}'

def pretty = JsonOutput.prettyPrint(json)

assert pretty == '''{
    "foo": "bar",
    "baz": [
        1
    ]
}'''
```

Write a json to a file

```
import groovy.json.JsonOutput;

def json = JsonOutput.toJson([foo: 'bar', baz: [1]])

new File("/tmp/output.json").write(json)
```

Read JSON online: https://riptutorial.com/groovy/topic/5352/json

Chapter 11: Memoized Functions

Examples

Memoized functions

Memoizing is basically a way to cache method results. This can be useful when a method is often called with the same arguments and the calculation of the result takes time, therefore increasing performance.

Starting from Groovy 2.2, methods can be annoted with the `@Memoized` annotation.

Imagine the following class:

```
class MemoDemo {
  def timesCalculated = 0

  @Memoized
  def power2(a) {
    timesCalculated++
    a * a
  }
}
```

Now upon the first call of this method with a number it hasnt been called with before, the method will be executed:

```
assert power2(2) == 4
assert timesCalculated == 1
```

However, if we call it again with the same argument:

```
assert power2(2) == 4
assert timesCalculated == 1
```

`timesCalculated` has remained unchanged, yet the method returned the same result. However, calling it with a different argument:

```
assert power2(3) == 9
assert timesCalculated == 2
```

results in the body of the method being called again.

Read Memoized Functions online: https://riptutorial.com/groovy/topic/6176/memoized-functions

Chapter 12: Memoized Functions

Examples

Memoize on closures

Since *Groovy* 1.8 a convenient `memoize()` method is added on closures:

```
// normal closure
def sum = { int x, int y  ->
    println "sum ${x} + ${y}"
    return x + y
}
sum(3, 4)
sum(3, 4)
// prints
// sum 3 + 4
// sum 3 + 4

// memoized closure
def sumMemoize = sum.memoize()
sumMemoize(3, 4)
// the second time the method is not called
// and the result it's take from the previous
// invocation cache
sumMemoize(3, 4)
// prints
// sum 3 + 4
```

Memoize on methods

Since *Groovy 2.2* `groovy.transform.Memoized` annotation is added to convenient memoize methods with simply adding the `@Memoized` annotation:

```
import groovy.transform.Memoized

class Calculator {
    int sum(int x, int y){
        println "sum ${x} + ${y}"
        return x+y
    }

    @Memoized
    int sumMemoized(int x, int y){
        println "sumMemoized ${x} + ${y}"
        return x+y
    }
}

def calc = new Calculator()

// without @Memoized, sum() method is called twice
calc.sum(3,4)
calc.sum(3,4)
```

```
// prints
// sum 3 + 4
// sum 3 + 4

// with @Memoized annotation
calc.sumMemoized(3,4)
calc.sumMemoized(3,4)
// prints
// sumMemoized 3 + 4
```

Read Memoized Functions online: https://riptutorial.com/groovy/topic/6471/memoized-functions

Chapter 13: RESTClient

Introduction

Groovy's HTTP Client usage, examples and pitfalls.

Examples

GET Request

```
@Grab(group='org.codehaus.groovy.modules.http-builder', module='http-builder', version='0.7' )

import groovyx.net.http.RESTClient

try {
    def restClient = new RESTClient("http://weathers.co")
    def response = restClient.get(path: '/api.php', query: ['city': 'Prague'])
    println "Status     : ${response.status}"
    println "Body       : ${response.data.text}"
} catch (Exception e) {
    println "Error      : ${e.statusCode}"
    println "Message    : ${e.response.data}"
}
```

Read RESTClient online: https://riptutorial.com/groovy/topic/8919/restclient

Chapter 14: Safe Navigation Operator

Examples

Basic usage

Groovy's *safe navigation operator* allows to avoid `NullPointerException`s when accessing to methods or attributes on variables that may assume `null` values. It is equivalent to `nullable_var == null ? null : nullable_var.myMethod()`

```
def lst = ['foo', 'bar', 'baz']

def f_value = lst.find { it.startsWith('f') }    // 'foo' found
f_value?.length()    // returns 3

def null_value = lst.find { it.startsWith('z') }    // no element found. Null returned

// equivalent to  null_value==null ? null : null_value.length()
null_value?.length()    // no NullPointerException thrown

// no safe operator used
null_value.length()    // NullPointerException thrown
```

Concatenation of safe navigation operators

```
class User {
  String name
  int age
}

def users = [
  new User(name: "Bob", age: 20),
  new User(name: "Tom", age: 50),
  new User(name: "Bill", age: 45)
]

def null_value = users.find { it.age > 100 }    // no over-100 found. Null

null_value?.name?.length()    // no NPE thrown
//  null ?. name  ?. length()
// (null ?. name) ?. length()
// (    null    ) ?. length()
// null

null_value?.name.length()    // NPE thrown
//  null ?. name  . length()
// (null ?. name) . length()
// (    null    ) . length()  ===> NullPointerException
```

the safe navigation on `null_value?.name` will return a `null` value. Thus `length()` will have to perform a check on `null` value to avoid a `NullPointerException`.

Read Safe Navigation Operator online: https://riptutorial.com/groovy/topic/5116/safe-navigation-operator

Chapter 15: Spaceship Operator

Examples

Basic usage

the spaceship operator returns `-1` when the left operator is smaller, `0` when the operators are equal and `1` otherwise:

```
assert 10 <=> 20 == -1
assert 10 <=> 10 == 0
assert 30 <=> 10 == 1

assert 'a' <=> 'b' == -1
assert 'a' <=> 'a'== 0
assert 'b' <=> 'a' == 1
```

It is equivalent to the Comparable.compareTo method:

```
assert 10.compareTo(20) == (10 <=> 20)
assert 'a'.compareTo('b') == ('a' <=> 'b')
```

Spaceship operator for custom sortings

```
class User {
  String name
  int age
}

def users = [
  new User(name: "Bob", age: 20),
  new User(name: "Tom", age: 50),
  new User(name: "Bill", age: 45)
]

// sort by age
users.sort { a, b -> a.age <=> b.age }
```

Usage with Comparator and SortedSet

```
Comparator cmp = [ compare:{ a, b -> a <=> b } ] as Comparator
def col = [ 'aa', 'aa', 'nn', '00' ]
SortedSet sorted = new TreeSet( cmp )
sorted.addAll col
assert '[00, aa, nn]' == sorted.toString()
```

Read Spaceship Operator online: https://riptutorial.com/groovy/topic/4394/spaceship-operator

Chapter 16: Spread Operator

Remarks

In most cases, the spread operator `*.` is identical to calling `.collect { it.________ }`.

```
def animals = ['cat', 'dog', 'fish']
assert animals*.length() == animals.collect { it.length() }
```

But if the subject is null, they behave a differently:

```
def animals = null
assert animals*.length() == null
assert animals.collect { it.length() } == []
```

Examples

Calling a method

```
assert ['cat', 'dog', 'fish']*.length() == [3, 3, 4]
```

Note that when mixing types in the collection if the method not exists on some of the elements, a `groovy.lang.MissingMethodException` could be thrown:

```
['cat', 'dog', 'fish',3]*.length()
// it throws groovy.lang.MissingMethodException: No signature of method:
java.lang.Integer.length()
```

Accessing a property

```
class Vector {
    double x
    double y
}
def points = [
    new Vector(x: 10, y: -5),
    new Vector(x: -17.5, y: 3),
    new Vector(x: -3.3, y: -1)
]

assert points*.x == [10, -17.5, -3.3]
```

Note: The `*` is optional. We could also write the above statement as in the below line and Groovy compiler would still be happy about it.

```
assert points.x == [10, -17.5, -3.3]
```

Its null-safe

If there is a `null` object on the collection it not throws a `NPE`, it returns a `null` instead:

```
assert ['cat', 'dog', 'fish', null]*.length() == [3, 3, 4, null]
```

Using it directly in a `null` object it's also null-safe:

```
def nullCollection = null
assert nullCollection*.length() == null
```

Read Spread Operator online: https://riptutorial.com/groovy/topic/2725/spread-operator

Chapter 17: String Interpolation

Syntax

- $
- ${}
- ${->}

Examples

Basic

```
def str = 'nice'
assert "Groovy is $str" == 'Groovy is nice'
```

Dotted Expression

```
def arg = [phrase: 'interpolated']
assert "This is $arg.phrase" == 'This is interpolated'
```

Eager expression

```
def str = 'old'
def interpolated = "I am the ${str} value"
assert interpolated == 'I am the old value'
str = 'new'
assert interpolated == 'I am the old value'
```

Lazy expression

We can have lazy interpolation in Strings. This is different than normal interpolation as the GString can potentially have different values, depending on the closure, whenever it is converted into a String.

```
def str = 'old'
def interpolated = "I am the ${ -> str} value"
assert interpolated == 'I am the old value'
str = 'new'
assert interpolated == 'I am the new value'
```

Expression

```
def str = 'dsl'
def interpolated = "Groovy ${str.length() + 1} easy ${str.toUpperCase()}"
assert interpolated == 'Groovy 4 easy DSL'
str = 'Domain specific language'
```

```
assert interpolated == 'Groovy 4 easy DSL'
```

Read String Interpolation online: https://riptutorial.com/groovy/topic/3125/string-interpolation

Chapter 18: Strings and GString literals

Syntax

- 'Single quoted string'
- "Double quoted string"
- '''Multiline string'''
- """Triple double quoted string"""
- /Slashy string/
- $/Dollar slash string/$

Remarks

Groovy has two string types the java `java.lang.String` and `groovy.lang.GString`, as well as multiple forms of string literals (see syntax and examples).

The main difference between the two types of strings is that GString supports string interpolation.

Examples

Single quoted string

```
def str = 'Single quoted string'
assert str instanceof String
```

Double quoted string (without interpolation placeholder)

```
def str = "Double quoted string"
assert str instanceof String
```

Double quoted string (interpolation)

```
def param = 'string'
def str = "Double quoted ${param}"
assert str instanceof GString
assert str == 'Double quoted string'
```

The parameter is by default resolved eagerly, this means this applies:

```
def param = 'string'
def str = "Double quoted ${param}"
param = 'another string'
assert str == 'Double quoted string'
```

In order to load the parameter lazily every time the string is used, this can be done:

```
def param = 'string'
def str = "Double quoted ${ -> param}"
assert str == 'Double quoted string'
param = 'lazy load'
assert str == 'Double quoted lazy load'
```

Multiline string

```
def str = '''multiline
string'''
assert str instanceof String
```

Multiline string (extra trailing newline)

```
def str = '''
multiline
string'''
assert str.readLines().size() == 3
```

Multiline string (without extra trailing newline)

```
def str = '''\
multiline
string'''
assert str.readLines().size() == 2
```

Triple double quoted string

```
def param = 'string'
def str = """
multiline
$param
"""
assert str instanceof GString
assert str.readLines().size() == 3
assert str == '''
multiline
string
'''
```

Slashy string (no interpolation placeholder)

```
def str = /
multiline string
no need to escape slash
\n
/
assert str instanceof String
assert str.readLines().size() == 4
assert str.contains('\\n')
```

Slashy string (interpolation)

```
def param = 'string'
def str = /
multiline $param
no need to escape slash
\n
/
assert str instanceof GString
assert str.readLines().size() == 4
assert str.contains('\\n')
assert str.contains('string')
```

Dollar slash string

```
def param = 'string'
def str = $/
multiline $param
no need to escape slash
\n
$
$$
/$
assert str instanceof GString
assert str.readLines().size() == 6
assert str.contains('\\n')
assert str.contains('$')
```

Read Strings and GString literals online: https://riptutorial.com/groovy/topic/3409/strings-and-gstring-literals

Chapter 19: Ternary and Elvis Operators

Remarks

The Elvis operator evaluates based on *Groovy-Truth* of the condition-part.

Examples

Standard form vs Elvis form

```
// long form
String sayHello(String name){
    "Hello, ${name ? name : 'stranger'}."
}

// elvis
String sayHello(String name){
    "Hello, ${name ?: 'stranger'}."
}
```

Notice that the "elvis" format omits the "true" term because the original comparison value is to be used in the "true" case. If `name` is Groovy `true`, then it will be returned as the value of the expression.

Usage (with condition) in assignment

```
def results = []
(1..4).each{
    def what = (it%2) ? 'odd' :  'even'
    results << what
}
assert results == ['odd', 'even', 'odd', 'even']
```

Here, the if-condition (in `(parentheses)`) is slightly more complex than just testing for existence/Groovy-Truth.

Read Ternary and Elvis Operators online: https://riptutorial.com/groovy/topic/3912/ternary-and-elvis-operators

Chapter 20: Traits

Introduction

Traits are structural construction objects in the Groovy language. Traits enable implementation of interfaces. They are compatible with static type checking and compilation Traits are behaved as interfaces with default implementations and state. Declaration of a trait is by using the **trait** keyword. ---------- Traits methods scope support only **public** and **private** methods.

Examples

Basic Usage

A `trait` is a reusable set of methods and fields that can be added to one or more classes.

```
trait BarkingAbility {
    String bark(){ "I'm barking!!" }
}
```

They can be used like normal interfaces, using `implements` keyword:

```
class Dog implements BarkingAbility {}
def d = new Dog()
assert d.bark() = "I'm barking!!"
```

Also they can be used to implement multiple inheritance (avoiding diamond issue).

Dogs can scratch his head, so:

```
trait ScratchingAbility {
    String scratch() { "I'm scratching my head!!" }
}

class Dog implements BarkingAbility, ScratchingAbility {}
def d = new Dog()
assert d.bark() = "I'm barking!!"
assert d.scratch() = "I'm scratching my head!!"
```

Multiple inheritance problem

Class can implement multiple traits. In case if one trait defines method with the same signature like another trait, there is a multiple inheritance problem. In that case the method from **last declared trait** is used:

```
trait Foo {
  def hello() {'Foo'}
}
trait Bar {
  def hello() {'Bar'}
}

class FooBar implements Foo, Bar {}

assert new FooBar().hello() == 'Bar'
```

Read Traits online: https://riptutorial.com/groovy/topic/6687/traits

Chapter 21: Use ConfigSluper (instead of property files)

Introduction

ConfigSlurper allows you to use another groovy script as a config file for your script instead of using, for example, a .properties file. You can do interesting configurations with typed properties and you don't need to convert from string. You can use lists, maps or a value based on some calculation or closure.

Examples

ConfigSlurper using string, number, boolean or list

In the file myConfig.groovy is the following content.

```
message = 'Hello World!'
aNumber=42
aBoolean=false
aList=["apples", "grapes", "oranges"]
```

Then in your main script you create a ConfigSlurper for your myConfig.groovy file which is really just another groovy script.

```
config = new ConfigSlurper().parse(new File('/path/to/myConfig.groovy').toURL())
```

Then to use the items from the config you can just refer to them.

```
assert 'Hello World!' == config.message
assert 42 == config.aNumber
assert false == config.aBoolean
assert ["apples", "grapes", "oranges"] == config.aList
```

Read Use ConfigSluper (instead of property files) online:
https://riptutorial.com/groovy/topic/8291/use-configsluper--instead-of-property-files-

Chapter 22: Visiblity

Examples

Private fields and methods are not private in groovy

```
class MyClass {
    private String privateField
}

def prvtClss = new MyClass(privateField: 'qwerty')
println prvtClss.privateField
```

will print us 'qwerty'

This issue is known since version 1.1 and there is a bug report on that: http://jira.codehaus.org/browse/GROOVY-1875. It is not resolved even with groovy 2 release.

Read Visiblity online: https://riptutorial.com/groovy/topic/6522/visiblity

Chapter 23: Ways of Iteration in Groovy

Introduction

Groovy has more ways of looping besides supporting the Java iterations.

Groovy extends the `Integer` class with the `step()`, `upto()` and `times()` methods. These methods take a closure as a parameter. In the closure we define the piece of code we want to be executed several times.

It also adds `each()` and `eachWithIndex()` methods to iterate over collections.

Examples

How can I do something n times?

How can I print *hello world* 5 times?

```
5.times{
    println "hello world"
}
```

Each and EachWithIndex

`each` and `eachWithIndex` are methods to iterate over collections.

each have `it`(default iterator) and `eachWithIndex` have `it`,`index`(default iterator, default index).

We can also change the default iterator/index. Please see below examples.

```
def list = [1,2,5,7]
list.each{
    println it
}

list.each{val->
    println val
}

list.eachWithIndex{it,index->
    println "value " + it + " at index " +index
}
```

Read Ways of Iteration in Groovy online: https://riptutorial.com/groovy/topic/9844/ways-of-iteration-in-groovy

Credits

S. No	Chapters	Contributors
1	Getting started with groovy	Andrii Abramov, Ashish Patel, Bill K, cjstehno, Community, Eric Siebeneich, Gergely Toth, IronHorse, lospejos, Michael Schaefer, mnd, Piotr Chowaniec, Rao, rdmueller, SerCe
2	AST Transformations	mnoronha, Will
3	Closure Memoize Methods	Gergely Toth, hippocrene, John Mercier, mnoronha
4	Closures	Andrii Abramov, Anshul, August Lilleaas, Ben, Craig Trader, Eric Siebeneich
5	Collection Operators	Batsu, Bill K, mnoronha, traneHead
6	Currying	Gergely Toth
7	Domain Specific Languages	gclaussn
8	Groovy code golfing	Charanjith A C
9	Groovy Truth (true-ness)	ARA, Piotr Chowaniec
10	JSON	albciff, Batsu, nachoorme, Stefan van den Akker
11	Memoized Functions	mnoronha, OsaSoft
12	RESTClient	sm4
13	Safe Navigation Operator	Batsu, mnoronha
14	Spaceship Operator	Batsu, injecteer, jwepurchase, mnoronha
15	Spread Operator	adarshr, albciff, Batsu, Eric Siebeneich, Martin Neal
16	String Interpolation	Aseem Bansal, Gergely Toth, jdv, mnoronha
17	Strings and GString literals	Gergely Toth, OsaSoft, Rao
18	Ternary and Elvis	cjstehno, fheub, Piotr Chowaniec

	Operators	
19	Traits	NachoB, Piotr Chowaniec, Rotem
20	Use ConfigSluper (instead of property files)	dallyingllama
21	Visiblity	Anton Hlinisty, Gergely Toth
22	Ways of Iteration in Groovy	Anshul, dsharew