Groovy Goodness

Experience the Groovy programming language through code snippets

Hubert A. Klein Ikkink

Groovy Goodness Notebook

Experience the Groovy programming language through code snippets

Hubert A. Klein Ikkink (mrhaki)

This book is for sale at http://leanpub.com/groovy-goodness-notebook

This version was published on 2023-04-19

Tweet This Book!

Please help Hubert A. Klein Ikkink (mrhaki) by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Groovy Goodness Notebook with Groovy Goodness blog posts bundled into one book. #groovy @mrhaki

The suggested hashtag for this book is #groovygoodnessnotebook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#groovygoodnessnotebook

Also By Hubert A. Klein Ikkink (mrhaki)

Grails Goodness Notebook

Gradle Goodness Notebook

Spocklight Notebook

Awesome Asciidoctor Notebook

Ratpacked Notebook

This book is dedicated to my lovely family. I love you.

Contents

Strings

Turn Methods into Closures

Groovy supports closures and they are very useful when we create Groovy applications. For example we can pass closures as arguments to methods to execute them. We can create closures ourselves, but we can also convert a method to a closure with the `.&` operator. And we can use the converted method just like a normal closure. Because Groovy can use Java objects we can also convert a Java method into a closure.

Let's start with a simple Java class:

```
public class JavaObject {
 public static void javaSays(final String s) {
  System.out.println("Java says: Hello " + s + "!");
 }
}
```

With the following script we use this Java class and convert the `javaSays` method to a closure:

```
// Simple list with names.
def names = ['groovy', 'grails', 'mrhaki']

// Simple closure.
names.each { println 'Normal closure says: Hello ' + it + '!' }

// Groovy method to convert to closure.
def groovySays(s) {
 "Groovy says: Hello ${s}!"
}
// Use .& syntax to convert method to closure.
names.each(this.&groovySays)

// Convert Java method to closure and use it.
def javaSays = JavaObject.&javaSays
names.each javaSays
```

If we run this script we get the following output:

```
Normal closure says: Hello groovy!
Normal closure says: Hello grails!
Normal closure says: Hello mrhaki!
Groovy says: Hello groovy!
Groovy says: Hello grails!
Groovy says: Hello mrhaki!
Java says: Hello groovy!
Java says: Hello grails!
Java says: Hello mrhaki!
```

Original post written on August 19, 2009

New Dollar Slashy Strings

Groovy already has a lot of ways to define a String value, and with Groovy 1.8 we have another one: the dollar slashy String. This is closely related to the slashy String definition we already knew (which also can be multi-line by the way, added in Groovy 1.8), but with different escaping rules. We don't have to escape a slash if we use the dollar slashy String format, which we would have to do otherwise.

```
def source = 'Read more about "Groovy" at http://mrhaki.blogspot.com/'

// 'Normal' slashy String, we need to escape / with \/
def regexp = /.*"(.*)".*\/(.*)\//

def matcher = source =~ regexp
assert matcher[0][1] == 'Groovy'
assert matcher[0][2] == 'mrhaki.blogspot.com'

// Dollar slash String.
def regexpDollar = $/.*"(.*)".*/(.*)//$

def matcherDollar = source =~ regexpDollar
assert matcherDollar[0][1] == 'Groovy'
assert matcherDollar[0][2] == 'mrhaki.blogspot.com'

def multiline = $/
Also multilines
are supported.
/$
```

Original post written on April 27, 2011

Get to Know More About a GString

One of Groovy's great features is the GString. With the GString we can write strings containing expressions that are evaluated. We create a GString if our string is inside double quotes. We can found out information about the expressions in our GString with some simple methods and properties:

```
def user = 'mrhaki'
def language = 'Groovy'

def s = "Hello ${user}, welcome to ${language}."

assert 2 == s.valueCount
assert ['mrhaki', 'Groovy'] == s.values
assert 'mrhaki' == s.getValue(0)
assert 'Groovy' == s.getValue(1)
assert 32 == s.length()
assert 'Hello ' == s.strings[0]
assert ', welcome to ' == s.strings[1]
assert '.' == s.strings[2]
assert 'Hello mrhaki, welcome to Groovy.' == s
```

Original post written on July 14, 2010

Check if String is a Number

Groovy adds several methods to the `String` class to see if the string value is a number. We can check for all kind of number type like `Integer`, `Double`, `BigDecimal` and more.

```
assert '42'.isNumber()
assert '42'.isInteger() && '42'.isLong() && '42'.isBigInteger()
assert '42.42'.isDouble() && /42.42/.isBigDecimal() && '42.42'.isFloat()
```

Original post written on January 3, 2010

What Character Are You?

Groovy adds a couple of methods to the `Character` class to determine of the character is a letter, digit, whitespace, upper- or lowercase.

```
def str = 'a1cB \n.9'
def characters = str.chars  // Convert to char[]

assert characters[0].isLetter()  // 'a'
assert characters[1].isDigit()  // '1'
assert characters[2].isLowerCase()  // 'c'
assert characters[3].isUpperCase()  // 'B'
assert characters[4].isWhitespace()  // ' '
assert characters[5].isWhitespace()  // '\n'
assert !characters[6].isLetterOrDigit()  // '.'
assert characters[7].isLetterOrDigit()  // '9'
```

Original post written on December 29, 2009

Access Strings with Subscript Operator

Groovy adds a lot of support to the `String` class. The `getAt` method is added and that means we can use the subscript operator (`[]`) to access parts of a String.

```
def s = 'Accessing Strings in Groovy is easy.'

assert 'A' == s[0]
assert 'A' == s.getAt(0)
assert 'Groovy' == s[21..26]  // We can use ranges.
assert 'easy.' == s[s.indexOf('ea')..-1]

// We can also use each method on a String.
s[21..26].each { println "$it-" } // Output: G-r-o-o-v-y-
```

Original post written on October 31, 2009

Capitalize Strings

Groovy 1.7.3 adds the `capitalize()` method to the `String` class. This will capitalize the first letter of the String:

```
assert 'MrHaki' == 'mrHaki'.capitalize()
assert 'Groovy' == 'groovy'.capitalize()
assert 'Groovy is Gr8!' == 'groovy is Gr8!'.capitalize()
```

Original post written on June 14, 2010

Uncapitalize Strings

Since Groovy 2.4.8 we can use the `uncapitalize` method on `CharSequence` objects. The `capitalize` method was already available for a long time, but now we have the opposite as well.

In the following example we see that the `uncapitalize` method only replaces the first letter of a `String` value to lower case:

```
assert 'Groovy'.uncapitalize() == 'groovy'
assert 'MrHaki'.uncapitalize() == 'mrHaki'

String message = 'Groovy Rocks!'
assert message.uncapitalize() == 'groovy Rocks!'
```

Written with Groovy 2.4.8.

Original post written on January 16, 2017

Base64 Encoding

The `byte[]` and `String` classes in Groovy's GDK have methods to Base64 encode and decode Strings.

```
def s = 'Argh, Groovy you say, mate?'

String encoded = s.bytes.encodeBase64().toString()
assert 'QXJnaCwgR3Jvb3Z5IHlvdSBzYXksIG1hdGU/' == encoded

byte[] decoded = encoded.decodeBase64()
assert s == new String(decoded)
```

Run this script on Groovy web console.

Original post written on November 4, 2009

Check if a String Only Contains Whitespaces

In Groovy we can check if a `String` value only contains whitespaces with the `isAllWhitespace()` method. The method checks for spaces, but also takes into account tab and newline characters as whitespace.

```
assert ''.allWhitespace
assert '  '.allWhitespace
assert '\t '.allWhitespace
assert ' \r\n '.allWhitespace

assert !'mrhaki'.allWhitespace
```

Original post written on September 9, 2013

Convert String to Boolean

Groovy adds the `toBoolean()` method to the `String` class. If the value of the string is *true*, *1* or *y* the result is true, otherwise it is false.

```
assert "y".toBoolean()
assert 'TRUE'.toBoolean()
assert '  trUe  '.toBoolean()
assert " y".toBoolean()
assert "1".toBoolean()

assert ! 'other'.toBoolean()
assert ! '0'.toBoolean()
assert ! 'no'.toBoolean()
assert ! '  FalSe'.toBoolean()
```

Original post written on November 13, 2009

Padding Strings

Groovy extends the `String` class with a couple of padding methods. These methods allows us to define a fixed width a `String` value must occupy. If the `String` itself is less than the fixed width then the space is padded with spaces or any other character or `String` we define. We can pad to the left or the right of the `String` or both left and right and put the `String` in the center.

These methods are especially useful when we create Groovy scripts that run on the console and we want to format some output.

```
assert '   Groovy   ' == 'Groovy'.center(12)
assert 'Groovy      ' == "Groovy".padRight(12)
assert '      Groovy' == /Groovy/.padLeft(12)

assert '---Groovy---' == "Groovy".center(12, '-')
assert 'Groovy * * *' == "Groovy".padRight(12, ' *')
assert 'Groovy Groovy Groovy' == 'Groovy'.padLeft(20, 'Groovy ')

def createOutput = {
    def table = [
        // Page,    Response time, Size
        ['page1.html',        200, 1201],
        ['page2.html',         42, 8853],
        ['page3.html',         98, 3432],
        ['page4.html',        432, 9081]
    ]

    def total = { data, index ->
        data.inject(0) { result, row -> result += row[index] }
    }
    def totalTime = total.curry(table, 1)
    def totalSize = total.curry(table, 2)

    def out = new StringBuffer()
    out << ' Summary '.center(15, "*") << '\n\n'
    out << 'Total pages:'.padRight(25)
    out << table.size().toString().padLeft(6) << '\n'
    out << 'Total response time (ms):'.padRight(25)
    out << totalTime().toString().padLeft(6) << '\n'
    out << 'Total size (KB):'.padRight(25)
    out << totalSize().toString().padLeft(6) << '\n\n'

    out << ' Details '.center(15, "*") << '\n\n'
    table.each {
        out << it[0].padRight(14)
        out << it[1].toString().padLeft(5)
        out << it[2].toString().padLeft(8)
        out << '\n'
    }
    out.toString()
}

assert '''\
```

```
*** Summary ***

Total pages:                   4
Total response time (ms):    772
Total size (KB):           22567

*** Details ***

page1.html      200    1201
page2.html       42    8853
page3.html       98    3432
page4.html      432    9081
''' == createOutput()
```

Original post written on September 20, 2009

Working with Lines in Strings

In Groovy we can create multiline strings, which contain line separators. But we can also read text from an file containing line separators. The Groovy String GDK contains method to work with strings that contain line separators. We can loop through the string line by line, or we can do split on each line. We can even convert the line separators to the platform specific line separators with the `denormalize()` method or linefeeds with the `normalize()` method.

```
def multiline = '''\
Groovy is closely related to Java,
so it is quite easy to make a transition.
'''

// eachLine takes a closure with one argument, that
// contains the complete line.
multiline.eachLine {
    if (it =~ /Groovy/) {
        println it  // Output: Groovy is closely related to Java,
    }
}

// or eachLine has a closure with two argument, the current line
// and the line count.
multiline.eachLine { line, count ->
    if (count == 0) {
        println "line $count: $line"  // Output: line 0: Groovy is closely related to Java,
    }
}

def platformLinefeeds = multiline.denormalize()
def linefeeds = multiline.normalize()

// Read all lines and convert to list.
def list = multiline.readLines()
assert list instanceof ArrayList
```

```
assert 2 == list.size()
assert 'Groovy is closely related to Java,' == list[0]

def records = """\
mrhaki\tGroovy
hubert\tJava
"""

// splitEachLine will split each line with the specified
// separator. The closure has one argument, the list of
// elements separated by the separator.
records.splitEachLine('\t') { items ->
    println items[0] + " likes " + items[1]
}
// Output:
// mrhaki likes Groovy
// hubert likes Java
```

Run this script in Groovy web console.

Original post written on November 1, 2009

Getting Parts Of A String Enclosed By Strings

Groovy 3 adds the `takeBetween` method to the `String` class. With this method we can get all the characters that are enclosed by string values. We can specify one enclosed string value and then all text between the the first occurrence of the string and the second occurrence is returned. If multiple parts are enclosed by the string values we can also specify which occurrence we want. If the text is enclosed by different string values we can use a variant of `takeBetween` that takes two string values to indicate the boundaries of the text we want. Also with two different enclosed string values we can use an argument to get the n-th occurrence of the string that is found.\ Since Groovy 3 we can also use `takeBefore` and `takeAfter` to get the string before or after a given string value. All three methods will return an empty string if no text can be found.

In the following example we use the `takeBefore`, `takeAfter` and `takeBetween` methods with different arguments:

```
def text = 'Just saying: "Groovy is gr8!"'

// Return all characters before the first quote.
assert text.takeBefore('"') == 'Just saying: '
// Return everything after the colon.
assert text.takeAfter(': ') == '"Groovy is gr8!"'
// Return everything between two quotes.
assert text.takeBetween('"') == 'Groovy is gr8!'
// Return text between is and !.
assert text.takeBetween('is', '!') == ' gr8'

// When no value can be found
// an empty string is returned.
assert text.takeBefore('?') == ''
```

```
assert text.takeAfter('Java') == ''
assert text.takeBetween('-') == ''
assert text.takeBetween('[', '/') == ''

def sample = 'JVM languages are "Groovy", "Clojure", "Java".'

assert sample.takeBetween('"') == 'Groovy'
// We can also specify which occurrence we
// want for a text between same strings.
assert sample.takeBetween('"', 0) == 'Groovy'
assert sample.takeBetween('"', 1) == 'Clojure'
assert sample.takeBetween('"', 2) == 'Java'


def users = "Users: [mrhaki], [hubert]"

assert users.takeBetween('[', ']') == 'mrhaki'
// We can also specify which occurrence we
// want for a text between to strings.
assert users.takeBetween('[', ']', 0) == 'mrhaki'
assert users.takeBetween('[', ']', 1) == 'hubert'
// When no occurrence an empty string is returned.
assert users.takeBetween('[', ']', 2) == ''
```

Written with Groovy 3.0.2.

Original post written on March 11, 2020

Using the replaceAll Methods from String

Groovy adds two extra `replaceAll` methods to the `String` class. First we can pass a `Pattern` instead of a `String` argument with `replaceAll(Pattern, String)`. And with the other method we can use a closure to replace a value found with `replaceAll(String, Closure)`.

```
def s = "Programming with Groovy is fun!"

assert "Programming with Groovy rocks!" == s.replaceAll(~/is fun!/, "rocks!")  // Groovy extension to \
String.
assert "Programming with Groovy is awesome." == s.replaceAll("fun!", "awesome.")  // java.lang.String.\
replaceAll.

// Replace found String with result of closure.
def replaced = s.replaceAll(/fun/) {
    def list = ['awesome', 'cool', 'okay']
    list[new Random().nextInt(list.size())]
}
assert [
    "Programming with Groovy is awesome!",
    "Programming with Groovy is cool!",
    "Programming with Groovy is okay!"
].contains(replaced)
```

```
// Use closure to replace text and use grouping.
// First closure parameter is complete string and following
// parameters are the groups.
def txt = "Generated on 30-10-2009 with Groovy."
def replacedTxt = txt.replaceAll(/.*(\d{2}-\d{2}-\d{4}).*(Gr.*)./) { all, date, lang ->
    def dateObj = Date.parse('dd-MM-yyyy', date)
    "The text '$all' was created with $lang on a ${dateObj.format('EEEE')}."
}
assert "The text 'Generated on 30-10-2009 with Groovy.' was created with Groovy on a Friday." == repla\
cedTxt
```

Original post written on October 22, 2009

Replace Characters in a String with CollectReplacements

We can use the `collectReplacements(Closure)` method to replace characters in a `String`. We pass a closure to the method and the closure is invoked for each character in the `String` value. If we return `null` the character is not transformed, otherwise we can return the replacement character.

```
def s = 'Gr00vy is gr8'

def replacement = {
    // Change 8 to eat
    if (it == '8') {
        'eat'
    // Change 0 to o
    } else if (it == '0') {
        'o'
    // Do not transform
    } else {
        null
    }
}

assert s.collectReplacements(replacement) == 'Groovy is great'
```

Code written with Groovy 2.1.6

Original post written on September 6, 2013

Text Translation

In Groovy 1.7.3 the `tr()` method is added to the `String` class. With this method we can do translations in String values. We define a source set of characters that need to be replaced by a replacement set of characters. We can also use a regular expression style (remember it is not a real regular expression) to define a range of characters.

If the replacement set is smaller than the source set, than the last character of the replacement set is used for the remaining source set characters.

```
// Source set and replacement set are equal size.
assert 'I 10v3 9r00vy' == 'I love Groovy'.tr('loeG', '1039')

// Regular expression style range
assert 'mrHAKI' == 'mrhaki'.tr('a-k', 'A-K')

// Replacement set is smaller than source set.
assert 'Gr8888' == 'Groovy'.tr('ovy', '8')
```

Original post written on June 15, 2010

Remove Parts of a String

Groovy has added the `minus()` method to the String class. And because the `minus()` method is used by the - operator we can remove parts of a String with this operator. The argument can be a String or a regular expression Pattern. The first occurrence of the String or Pattern is then removed from the original String.

```
def s = 'Groovy and Strings are fun and versatile.'

assert 'Groovy and Strings are fun' == s - ' and versatile.'
assert 'Groovy and Strings are fun.' == s.minus(" and versatile")
assert 'Groovy  Strings are fun and versatile.' == s - ~/\b\w{3}\b/
```

Original post written on November 2, 2009

Remove Part of String With Regular Expression Pattern

Since Groovy 2.2 we can subtract a part of a String value using a regular expression pattern. The first match found is replaced with an empty String. In the following sample code we see how the first match of the pattern is removed from the String:

```
// Define regex pattern to find words starting with gr (case-insensitive).
def wordStartsWithGr = ~/(?i)\s+Gr\w+/

assert ('Hello Groovy world!' - wordStartsWithGr) == 'Hello world!'
assert ('Hi Grails users' - wordStartsWithGr) == 'Hi users'


// Remove first match of a word with 5 characters.
assert ('Remove first match of 5 letter word' - ~/\b\w{5}\b/) == 'Remove  match of 5 letter word'


// Remove first found numbers followed by a whitespace character.
assert ('Line contains 20 characters' - ~/\d+\s+/) == 'Line contains characters'
```

Code written with Groovy 2.2.

Original post written on November 18, 2013

Taking Or Dropping Number Of Characters From A String

Groovy adds a lot of methods to the Java `String` class. For example we can use the `take` method to get a certain number of characters from the start of a string value. With the `drop` method we remove a given number of characters from the start of the string. In Groovy 3 we can now also take and drop a certain number of characters from the end of a string using the methods `takeRight` and `dropRight`.

In the following example we see how we can use the methods:

```
def s = "Groovy rocks!"

// Drop first 7 characters.
assert s.drop(7) == "rocks!"

// Drop last 7 characters.
assert s.dropRight(7) == "Groovy"


// Take first 6 characters.
assert s.take(6) == "Groovy"

// Take last 6 characters.
assert s.takeRight(6) == "rocks!"
```

Written with Groovy 3.0.2.

Original post written on March 10, 2020

Splitting Strings

In Java we can use the `split()` method of the `String` class or the `StringTokenizer` class to split strings. Groovy adds the methods `split()` and `tokenize()` to the `String` class, so we can invoke them directly on a string. The `split()` method return a `String[]` instance and the `tokenize()` method return a `List`. There is also a difference in the argument we can pass to the methods. The `split()` method takes a regular expression string and the `tokenize()` method will use all characters as delimiter.

```
def s = '''\
username;language,like
mrhaki,Groovy;yes
'''

assert s.split() instanceof String[]
assert ['username;language,like', 'mrhaki,Groovy;yes'] == s.split()  // Default split on whitespace. (\
 \t\n\r\f)
assert ['username', 'language', 'like', 'mrhaki', 'Groovy', 'yes'] == s.split(/(;|,|\n)/)  // Split ar\
gument is a regular expression.

def result = []
s.splitEachLine(",") {
```

```
    result << it  // it is list with result of split on ,
}
assert ['username;language', 'like'] == result[0]
assert ['mrhaki', 'Groovy;yes'] == result[1]

assert s.tokenize() instanceof List
assert ['username;language,like', 'mrhaki,Groovy;yes'] == s.tokenize()  // Default tokenize on whitesp\
ace. ( \t\n\r\f)
assert ['username', 'language', 'like', 'mrhaki', 'Groovy', 'yes'] == s.tokenize("\n;,")  // Argument \
is a String with all tokens we want to tokenize on.
```

Run script on Groovy web console.

Original post written on November 5, 2009

Get Unique Characters in a String

Groovy adds the `toSet()` method to the `String` class in version 1.8. With this method we get a Set of unique String values from the original String value.

```
String s = 'Groovy is gr8!'

assert s.toSet().sort().join() == ' !8Ggiorsvy'
```

Original post written on April 27, 2011

Partial Matches

Groovy 2.0 adds the `matchesPartially()` method to the `Matcher` class. This method returns true if a String value matches the pattern or if it matches the first part of the pattern. So with the `matchesPartially()` we get the result `true` if a String value or a longer String value matches the pattern.

```
def identification = /[A-Z]{2}\-\d{3,5}/

def matcher = 'AB-1234' =~ identification
assert matcher.matchesPartially()

matcher = 'XY-90' =~ identification
assert matcher.matchesPartially()

matcher = 'HA' =~ identification
assert matcher.matchesPartially()

matcher = 'A-431' =~ identification
assert !matcher.matchesPartially()

matcher = 'YK-901201' =~ identification
assert !matcher.matchesPartially()
```

Original post written on June 28, 2012

Transform String into Enum

After reading Groovy, State of the Union - Groovy Grails eXchange 2010 by Guillaume Laforge I discovered that in Groovy 1.7.6 we can transform a String into a Enum value. We can use type coersion or the `as` keyword to turn a String or GString into a corresponding Enum value (if possible).

```
enum Compass {
    NORTH, EAST, SOUTH, WEST
}

// Coersion with as keyword.
def north = 'NORTH' as Compass
assert north == Compass.NORTH

// Coersion by type.
Compass south = 'south'.toUpperCase()
assert south == Compass.SOUTH

def result = ['EA', 'WE'].collect {
    // Coersion of GString to Enum.
    "${it}ST" as Compass
}
assert result[0] == Compass.EAST
assert result[1] == Compass.WEST
```

Original post written on December 16, 2010

String Continuation

Groovy makes writing concise code easy,. We can use the continuation character (\) in a String to split up the definition over multiple lines.

```
def name ='mrhaki'

def s = "This is not a multiline\
 String, $name, but the continuation\
 character (\\) makes it more readable."

assert 'This is not a multiline String, mrhaki, but the continuation character (\\) makes it more read\
able.' == s
```

Original post written on November 29, 2010

Strip Leading Spaces from Lines

Multiline strings are very useful in Groovy. But sometimes they can mess up our code formatting especially if we want to use the multiline string's value literally. If our lines cannot start with spaces we must define our multiline string that way:

```
class Simple {

    String multi() {
        '''\
Multiline string
with simple 2 space
indentation.'''
    }

    // Now in Groovy 1.7.3:
    String multi173() {
        '''\
        Multiline string
        with simple 2 space
        indentation.'''.stripIndent()
    }

}
```

Since Groovy 1.7.3 we can strip leading spaces from such lines, so we can align the definition of our multiline string the way we want with the `stripIndent()` method. Groovy finds the line with the least spaces to determine how many spaces must be removed from the beginning of the line. Or we can tell the `stripIndent()` method how many characters must be removed from the beginning of the line.

```
def multi = '''\
  Multiline string
  with simple 2 space
  indentation.'''

assert '''\
Multiline string
with simple 2 space
indentation.''' == multi.stripIndent()

assert '''\
ine string
imple 2 space
ation.''' == multi.stripIndent(8)  // We can define the number of characters ourselves as well.
```

Original post written on June 14, 2010

Strip Leading Spaces from Lines with Margin

Since Groovy 1.7.3 we can use the `stripMargin()` method to strip characters up to and including a margin character from multiline strings. The default character is the pipe symbol (|), but we can pass a parameter to the method and use a custom character.

```
def s = '''\
        |Groovy
        |Grails
        |Griffon'''

assert '''\
Groovy
Grails
Griffon''' == s.stripMargin()

def s1 = '''\
   * Gradle
   * GPars
   * Spock'''

assert '''\
 Gradle
 GPars
 Spock''' == s1.stripMargin("* ")
```

Original post written on June 20, 2010

Formatted Strings with sprintf

Groovy adds the `sprintf()` method to the `Object` class. This means we can use the method in all of the classes, because it is defined at the top of the hierarchy. The `sprintf()` method uses the Java Formatter syntax to format values. We get a String as a result from the method.

```
assert 'Groovy is cool!' == sprintf( '%2$s %3$s %1$s', ['cool!', 'Groovy', 'is'])
assert '00042' == sprintf('%05d', 42)
```

Original post written on December 11, 2009

Expand or Unexpand Space or Tab Delimited Text

Groovy 1.7.3 adds new functionality to the `String` class. For example we can use the `expand()` method to expand tabs in a `String` to spaces with a default tab stop size of 8. We can use a parameter to use a different tab stop size. But we can also go the other way around.

So if we have a tabular text based on spaces we can convert the `String` to a tab separated `String`. Here the default tab stop size is also 8, but we can use the parameter to define a different tab stop size.

```
// Simple ruler to display 0 up to 30
def ruler = (0..30).inject('\n') { result, c ->
    result += (c % 10)
}

def stringWithTabs = 'Groovy\tGrails\tGriffon'

println ruler
println stringWithTabs.expand()  // default tab stop is 8
println stringWithTabs.expand(10)  // tab stop is 10

// Output:
// 012345678901234567890123456789
// Groovy  Grails  Griffon
// /Groovy    Grails    Griffon

assert 'Groovy  Grails  Griffon' == stringWithTabs.expand()
assert 'Groovy    Grails    Griffon' == stringWithTabs.expand(10)


def stringWithSpaces = 'Hubert  Klein   Ikkink'
def stringWithSpaces10 = 'Hubert    Klein     Ikkink'
println ruler
println stringWithSpaces
println stringWithSpaces10

// Output:
// 012345678901234567890123456789
// Hubert  Klein   Ikkink
// Hubert    Klein     Ikkink

assert 'Hubert\tKlein\tIkkink' == stringWithSpaces.unexpand()
assert 'Hubert\tKlein\tIkkink' == stringWithSpaces10.unexpand(10)
```

Original post written on June 14, 2010

Normalize and Denormalize Linefeeds and Carriage Returns

Each platform where we can run Java and Groovy applications has different line separators. Groovy adds two methods to the `String` class to convert the specific platform line separator to linefeeds and vica versa.

```
def text = 'First line\r\nSecond line\r\n'
def textNormalized = text.normalize()
def platformLineSeparator = System.properties['line.separator']

assert 'First line\nSecond line\n' == textNormalized
assert "First line${platformLineSeparator}Second line${platformLineSeparator}" == textNormalized.denor\
malize()
```

Original post written on January 2, 2010

Base64 URL and Filename Safe Encoding

Groovy supported Base64 encoding for a long time. Since Groovy 2.5.0 we can also use Base64 URL and Filename Safe encoding to encode a byte array with the method `encodeBase64Url`. The result is a `Writable` object. We can invoke the `toString` method on the `Writable` object to get a `String` value. An encoded `String` value can be decoded using the same encoding with the method `decodeBase64Url` that is added to the `String` class.

In the following example Groovy code we encode and decode a byte array:

```
import static java.nio.charset.StandardCharsets.UTF_8

def message = 'Groovy rocks!'

// Get bytes array for String using UTF8.
def messageBytes = message.getBytes(UTF_8)

// Encode using Base64 URL and Filename encoding.
def messageBase64Url = messageBytes.encodeBase64Url().toString()

// Encode using Base64 URL and Filename encoding with padding.
def messageBase64UrlPad = messageBytes.encodeBase64Url(true).toString()

assert messageBase64Url == 'R3Jvb3Z5IHJvY2tzIQ'
assert messageBase64UrlPad == 'R3Jvb3Z5IHJvY2tzIQ=='

// Decode the String values.
assert new String(messageBase64Url.decodeBase64Url()) == 'Groovy rocks!'
assert new String(messageBase64UrlPad.decodeBase64Url()) == 'Groovy rocks!'
```

Written with Groovy 2.5.0.

Original post written on June 11, 2018

Calculate MD5 And SHA Hash Values

Groovy adds a lot of useful methods to the `String` class. Since Groovy 2.5.0 we can even calculate MD5 and SHA hash values using the methods `md5` and `digest`. The `md5` method create a hash value using the MD5 algorithm. The `digest` method accepts the name of the algorithm as value. These values are dependent on the available algorithms on our Java platform. For example the algorithms MD2, MD5, SHA-1, SHA-256, SHA-384 and SHA-512 are by default available.

In the next example we use the `md5` and `digest` methods on a `String` value:

```
def value = 'IamASecret'

def md5 = value.md5()

// We can provide hash algorithm with digest method.
def md2 = value.digest('MD2')
def sha1 = value.digest('SHA-1')
def sha256 = value.digest('SHA-256')

assert md5 == 'a5f3147c32785421718513f38a20ca44'
assert md2 == '832cbe3966e186194b1203c00ef47488'
assert sha1 == '52ebfed118e0a411e9d9cbd60636fc9dea718928'
assert sha256 == '4f5e3d486d1fd6c822a81aa0b93d884a2a44daf2eb69ac779a91bc76de512cbe'
```

Written with Groovy 2.5.0.

Original post written on June 12, 2018

Converting Byte Array to Hex String

To convert a `byte[]` array to a `String` we can simply use the `new String(byte[])` constructor. But if the array contains non-printable bytes we don't get a good representation. In Groovy we can use the method `encodeHex()` to transform a `byte[]` array to a hex `String` value. The `byte` elements are converted to their hexadecimal equivalents.

```
final byte[] printable = [109, 114, 104, 97, 107, 105]

// array with non-printable bytes 6, 27 (ACK, ESC)
final byte[] nonprintable = [109, 114, 6, 27, 104, 97, 107, 105]


assert new String(printable) == 'mrhaki'
assert new String(nonprintable) != 'mr  haki'


// encodeHex() returns a Writable
final Writable printableHex = printable.encodeHex()
assert printableHex.toString() == '6d7268616b69'
final nonprintableHex = nonprintable.encodeHex().toString()
assert nonprintableHex == '6d72061b68616b69'


// Convert back
assert nonprintableHex.decodeHex() == nonprintable
```

Code written with Groovy 2.2.1

Original post written on April 3, 2014

GString as Writable

The Groovy API has the interface Writable. Classes that implement this interface are capable of writing their selves to a `java.io.Writer` object. The interface has one method `writeTo()` where the code is that writes the contents to a given `Writer` instance. Most implementations will also use the implementation of the `writeTo()` method in their `toString()` implementation.

The GString implementation in Groovy also implements the `Writable` interface. This means we can redirect the GString contents to some `Writer` instance if we want to. In the following code we use a `FileWriter` to write the contents of a GString to a file:

```
def data = [
    new Expando(id: 1, user: 'mrhaki', country: 'The Netherlands'),
    new Expando(id: 2, user: 'hubert', country: 'The Netherlands'),
]

data.each { userData ->
    new File("${userData.id}.txt").withWriter('UTF-8') { fileWriter ->
        // Use writeTo method on GString to save
        // result in a file.
        "User $userData.user lives in $userData.country".writeTo(fileWriter)
    }
}


assert new File('1.txt').text == 'User mrhaki lives in The Netherlands'
assert new File('2.txt').text == 'User hubert lives in The Netherlands'
```

Code written with Groovy 2.2.2

Original post written on April 4, 2014

Closure as Writable

In a previous post we learned about the `Writable` interface and how the GString implementation implements this interface. In Groovy we can also use a closure as an implementation of the `Writable` interface. The `Closure` class has the method `asWritable()` that will return a version of the closure with an implementation of the `writeTo()` method. The `Writer` object that is used as an argument for the `writeTo()` method will be passed as argument to the closure. The `asWritable()` method also adds a `toString()` implementation for the closure to return the result of a closure as a `String`.

In the following code we write a sample `make()` method. The `make()` method return a `Writable` closure. The closure is only executed when the `writeTo()` or `toString()` method is invoked.

```
Writable make(Map binding = [:], Closure template) {
    // Use asWritable() to make the closure
    // implement the Writable interface.
    def writableTemplate = template.asWritable()

    // Assing binding map as delegate so we can access
    // the keys of the maps as properties in the
    // closure context.
    writableTemplate.delegate = binding

    // Return closure as Writable.
    writableTemplate
}

// Use toString() of Writable closure.
assert make { Writer out -> out <<  "Hello world!" }.toString() == 'Hello world!'

// Provide data for the binding.
// The closure is not executed when the
// make method is finished.
final writable = make(user:'mrhaki', { out ->
    out.println "Welcome ${user},"
    out.print "Today on ${new Date(year: 114, month: 3, date: 4).format('dd-MM-yyyy')}, "
    out.println "we have a Groovy party!"
})

// We invoke toString() and now the closure
// is executed.
final result = writable.toString()

assert result == '''Welcome mrhaki,
Today on 04-04-2014, we have a Groovy party!
'''

// Append contents to a file.
// NOTE: The leftShift (<<) operator on File is implemented
// in Groovy to use the File.append() method.
// The append() method creates a new Writer and
// invokes the write() method which
// is re-implemented in Groovy if the argument
// is a Writable object. Then the writeTo() method
// is invoked:
// Writer.write(Writable) becomes Writable.writeTo(Writer).
// So a lot of Groovy magic allows us to use the following one-liner
// and still the writeTo() method is used on Writable.
new File('welcome.txt') << writable

assert new File('welcome.txt').text == '''Welcome mrhaki,
Today on 04-04-2014, we have a Groovy party!
'''
```

Code written with Groovy 2.2.2

Original post written on April 4, 2014

Maps

Check if Maps are Equal

With Groovy 1.8 the `equals()` method is added to `Map`. This means we can check if maps are equals. They are equals if both maps have the same size, and keys and values are the same.

```
def map1 = [user: 'mrhaki', likes: 'Groovy', age: 37]
def map2 = [age: 37.0, likes: 'Groovy', user: 'mrhaki']
def map3 = [user: 'Hubert Klein Ikkink', likes: 'Groovy']

assert map1.equals(map2)
assert map1 == map2
assert !map1.equals(map3)
assert map2 != map3
```

Original post written on April 27, 2011

Sorting a Map

Maps don't have an order for the elements, but we may want to sort the entries in the map. Since Groovy 1.7.2 we can use the `sort()` method which uses the natural ordering of the keys to sort the entries. Or we can pass a Comparator to the `sort()` method to define our own sorting algorithm for the keys.

```
def m = [sort: 'asc', name: 'test', paginate: true, max: 100]

def expectedKeys = ['max', 'name', 'paginate', 'sort']
assert expectedKeys == m.sort()*.key  // Since 1.7.2
assert expectedKeys == m.sort( { k1, k2 -> k1 <=> k2 } as Comparator )*.key // Since 1.7.2

// Sorting before Groovy 1.7.2
assert expectedKeys == new TreeMap(m)*.key
assert expectedKeys == m.sort { e1, e2 -> e1.key <=> e2.key }*.key  // Sort by closure.
```

Original post written on April 20, 2010

Turn a List into a Map

With Groovy we can use the values of an `Object` array and transform them to a map with the `toSpreadMap()` method. The array must have an even number of elements, because the odd elements are the keys for the new map and the even numbers are the values for the keys. The `SpreadMap` object, which now contains the keys and values, is an immutable map, so we cannot change the contents once we have created the map.

```
def list = ['key', 'value', 'name', 'mrhaki'] as Object[]
def map = list.toSpreadMap()

assert 2 == map.size()
assert 'value' == map.key
assert 'mrhaki' == map['name']
```

Original post written on January 4, 2010

Complex Keys in Maps

In Groovy we can use non-string keys for maps. We only have to place parenthesis around the key to make it work. This way we can use variables and types like Date and Boolean as keys for our map. When we use parenthesis around the key when using the . notation the key is converted to a String, otherwise the key is not converted and keeps it type.

```
def key = 100  // Variable to be used a key.

def m = [
    (new Date(109, 11, 1)): 'date key',
    (-42): 'negative number key',
    (false): 'boolean key',
    (key): 'variable key'
]
m.(true) = 'boolean key'  // Key is converted to String.
m.(2 + 2) = 'number key'
m[(key + 1)] = 'number key'  // Key keeps to be Integer.

assert 'date key' == m[new Date(109, 11, 1)]
assert 'negative number key' == m.get(-42)
assert 'boolean key' == m[(false)]
assert 'variable key' == m[100]
assert 'variable key' == m.getAt(key)
assert 'boolean key' == m['true']  // Key is String so we can use it to get the value.
assert 'number key' == m.'4'
assert 'number key' == m.get(101)
```

Original post written on November 7, 2009

Use inject Method on a Map

The `inject()` method is since Groovy 1.8.1 also available for Map objects. The closure arguments accepts two or three arguments. With the three-argument variant we get the key and value separately as arguments. Otherwise we get a map entry as closure argument.

```
// 3-argument closure with key, value.
def m = [user: 'mrhaki', likes: 'Groovy']
def sentence = m.inject('Message: ') { s, k, v ->
    s += "${k == 'likes' ? 'loves' : k} $v "
}

assert sentence.trim() == 'Message: user mrhaki loves Groovy'

// 2-argument closure with entry.
def map = [sort: 'name', order: 'desc']
def equalSizeKeyValue = map.inject([]) { list, entry ->
    list << (entry.key.size() == entry.value.size())
}

assert equalSizeKeyValue == [true, false]
```

Original post written on September 27, 2011

Intersect Maps

Since Groovy 1.7.4 we can intersect two maps and get a resulting map with only the entries found in both maps.

```
def m1 = [a: 'Groovy', b: 'rocks', c: '!']
def m2 = [a: 'Groovy', b: 'rocks', c: '?', d: 'Yes!']

assert [a: 'Groovy', b: 'rocks'] == m1.intersect(m2)

assert [1: 1.0, 2: 2] == [1: 1.0, 2: 2].intersect([1: 1, 2: 2.0])
```

Original post written on August 9, 2010

Subtracting Map Entries

Groovy 1.7.4 adds the `minus()` method to the `Map` class. The result is a new map with the entries of the map minus the same entries from the second map.

```
def m1 = [user: 'mrhaki', age: 37]
def m2 = [user: 'mrhaki', name: 'Hubert']
def m3 = [user: 'Hubert', age: 37]

assert [age: 37] == m1 - m2
assert [user: 'mrhaki'] == m1 - m3
```

Original post written on August 9, 2010

Process Map Entries in Reverse

Since Groovy 1.7.2 we can loop through a `Map` in reverse with the `reverseEach()`. The order in which the content is processed is not guaranteed with a `Map`. If we use a `TreeMap` the natural ordering of the keys of the map is used.

```
def reversed = [:]
[a: 1, c: 3, b: 2].reverseEach { key, value ->
    reversed[key] = value ** 2
}

assert [b: 4, c: 9, a: 1] == reversed

// TreeMap uses natural ordering of keys, so
// reverseEach starts with key 'c'.
def tree = [a: 10, c: 30, b: 20] as TreeMap
def reversedMap = [:]
tree.reverseEach {
    reversedMap[it.key] = it.value * 2
}
assert [c: 60, b: 40, a: 20] == reversedMap
```

Original post written on August 24, 2010

Getting a Submap from a Map

To get only a subset of a map we can use the `subMap()` method. We provide a list of keys as parameter to define which elements from the map we want returned.

```
def map = [name: 'mrhaki', country: 'The Netherlands', blog: true, languages: ['Groovy', 'Java']]

def keys = ['name', 'blog']
assert [name: 'mrhaki', blog: true] == map.subMap(keys)

def booleanKeys = map.findAll { it.value instanceof Boolean }.collect { it.key }
assert [blog: true] == map.subMap(booleanKeys)

def words = ['a': 'Apple', 'j': 'Java', 'g': 'Groovy', 'c': 'Cool']
def range = 'c'..'h'  // Range is also a list and can be used here.
def rangeWords = words.subMap(range).findAll{ it.value }
// words.subMap(range) returns [c:Cool, d:null, e:null, f:null, g:Groovy, h:null]
// so we use the findAll method to filter out all null values.
assert ['c': 'Cool', 'g': 'Groovy'] == rangeWords
```

Original post written on October 29, 2009

Grouping Map Elements

In a previous Groovy Goodness post we learned how to use the `groupBy` method on collections. The `Map` class has an extra method: `groupEntriesBy`. We must provide a closure

for this method to define how we want the elements of the map to be grouped. The result is a new `Map` with keys and a list of `Map$Entry` objects for each key. This is different from the result of the `groupBy` method. Because then we get a `Map` with keys and a `Map` for each key.

```
// A simple map.
def m = [q1: 'Groovy', sort: 'desc', q2: 'Grails']

// Closure we use to define the grouping.
// We want all keys starting with 'q' grouped together
// with the key 'params', all other keys are not grouped.
def groupIt = { key, value ->
    if (key.startsWith('q')) {
        'params'
    } else {
        key
    }
}

// Use groupEntriesBy.
def groupEntries = m.groupEntriesBy(groupIt)
assert 2 == groupEntries.size()
assert groupEntries.params & groupEntries.sort
assert 'desc' == groupEntries.sort[0].value  // Key for a list of Map$Entry objects.
assert 2 == groupEntries.params.size()
assert 'Groovy' == groupEntries.params[0].value
assert 'q1' == groupEntries.params[0].key
assert 'Grails' == groupEntries.params.find { it.key == 'q2' }.value
assert groupEntries.params instanceof ArrayList
assert groupEntries.params[0] instanceof Map$Entry

// Use groupBy.
def group = m.groupBy(groupIt)
assert 2 == group.size()
assert group.params & group.sort
assert 'desc' == group.sort.sort  // Key for Map with key/value pairs.
assert 2 == group.params.size()
assert 'Groovy' == group.params.q1
assert 'q1' == group.params.keySet().toArray()[0]
assert 'Grails' == group.params.q2
assert group.params instanceof Map
assert group.params.q1 instanceof String
```

Original post written on October 14, 2009

Get Value from Map or a Default Value

The `get()` method in the Groovy enhanced `Map` interface accepts two parameters. The first parameter is the name of the key we want to get a value for. And the second parameter is the default value if there is no value for the key.

```
// Simple map.
def m = [name: 'mrhaki', language: 'Groovy']

assert 'mrhaki' == m.getAt('name')
assert 'mrhaki' == m['name']
assert 'Groovy' == m.language
assert 'mrhaki' == m."name"
assert 'mrhaki' == m.get('name')  // We can omit the default value if we know the key exists.
assert 'Groovy' == m.get('language', 'Java')
assert null == m.get('expression')  // Non-existing key in map.
assert 'rocks' == m.get('expression', 'rocks')  // Use default value, this also creates the key/value \
pair in the map.
assert 'rocks' ==  m.get('expression')
assert [name: 'mrhaki', language: 'Groovy', expression: 'rocks'] == m
```

Run this script in Groovy web console.

Original post written on November 3, 2009

Map with Default Values

In Groovy we can create a map and use the `withDefault()` method with a closure to define default values for keys that are not yet in the map. The value for the key is then added to the map, so next time we can get the value from the map.

```
def m = [start: 'one'].withDefault { key ->
    key.isNumber() ? 42 : 'Groovy rocks!'
}

assert 'one' == m.start
assert 42 == m['1']
assert 'Groovy rocks!' == m['I say']
assert 3 == m.size()

// We can still assign our own values to keys of course:
m['mrhaki'] = 'Hubert Klein Ikkink'
assert 'Hubert Klein Ikkink' == m.mrhaki
assert 4 == m.size()
```

Original post written on July 14, 2010

Determine Min and Max Entries in a Map

Since Groovy 1.7.6 we can use the `min()` and `max()` methods on a `Map`. We use a closure to define the condition for a minimum or maximum value. If we use two parameters in the closure we must do a classic comparison. We return a negative value if the first parameters is less than the second, zero if they are both equal, or a positive value if the first parameter is greater than the second parameter. If we use a single parameter we can return a value that is used as `Comparable` for determining the maximum or minimum entry in the `Map`.

```
def money = [cents: 5, dime: 2, quarter: 3]

// Determine max entry.
assert money.max { it.value }.value == 5
assert money.max { it.key }.key == 'quarter'  // Use String comparison for key.
assert money.max { a, b ->
    a.key.size() <=> b.key.size()
}.key == 'quarter'  // Use Comparator and compare key size.

// Determine min entry.
assert money.min { it.value }.value == 2
assert money.min { it.key }.key == 'cents'  // Use String comparison for key.
assert money.min { a, b ->
    a.key.size() <=> b.key.size()
}.key == 'dime'  // Use Comparator and compare key size.
```

Original post written on December 16, 2010

Represent Map As String

Groovy adds to `Map` objects the `toMapString` method. With this method we can have a String representation of our `Map`. We can specify an argument for the maximum width of the generated `String`. Groovy will make sure at least the key/value pairs are added as a pair, before adding three dots (. . .) if the maximum size is exceeded.

```
def course = [
    name: 'Groovy 101',
    teacher: 'mrhaki',
    location: 'The Netherlands']

assert course.toMapString(15) == '[name:Groovy 101, ...]'
assert course.toMapString(25) == '[name:Groovy 101, teacher:mrhaki, ...]'
```

As mentioned in a previous post we can use the `toListString` method to represent a `List` as a `String`:

```
def names = ['mrhaki', 'hubert']

assert names.toListString(5) == '[mrhaki, ...]'
```

Written with Groovy 2.4.7.

Original post written on June 21, 2016

Turn A Map Or List As String To Map Or List

In a previous post we learned how to use the `toListString` or `toMapString` methods. With these methods we create a String representation of a `List` or `Map` object. With a bit of Groovy code we can take such a `String` object and turn it into a `List` or `Map` again.

In the following code snippet we turn the `String` value `[abc, 123, Groovy rocks!]` to a `List` with three items:

```
// Original List with three items.
def original = ['abc', 123, 'Groovy rocks!']

// Create a String representation:
// [abc, 123, Groovy rocks!]
def listAsString = original.toListString()

// Take the String value between
// the [ and ] brackets, then
// split on , to create a List
// with values.
def list = listAsString[1..-2].split(', ')

assert list.size() == 3
assert list[0] == 'abc'
assert list[1] == '123' // String value
assert list[2] == 'Groovy rocks!'
```

We can do something similar for a `String` value representing a map structure:

```
// Original Map structure.
def original = [name: 'mrhaki', age: 42]

// Turn map into String representation:
// [name:mrhaki, age:42]
def mapAsString = original.toMapString()

def map =
    // Take the String value between
    // the [ and ] brackets.
    mapAsString[1..-2]
        // Split on , to get a List.
        .split(', ')
        // Each list item is transformed
        // to a Map entry with key/value.
        .collectEntries { entry ->
            def pair = entry.split(':')
            [(pair.first()): pair.last()]
        }


assert map.size() == 2
assert map.name == 'mrhaki'
assert map.age == '42'
```

Written with Groovy 2.4.7.

Original post written on June 22, 2016