Groovy

Casiano R. León [1]

21 de abril de 2010

[1]DEIOC Universidad de La Laguna

Índice general

Índice de figuras

Índice de cuadros

A Juana

For it is in teaching that we learn
And it is in understanding that we are understood

Agradecimientos/Acknowledgments

I'd like to thank Paul King

Dierk König
Guillaume Laforgue
Ubay Machín
Venkat Subramaniam
Ted Naleid

Special thanks to The Groovy Community

Capítulo 1

Introducción

1.1. Lecturas Recomendadas

- Véase La Guía de usuario (Groovy User's Guide): http://docs.codehaus.org/display/GROOVY/User+Guide
- Groovy in Action por Dierk König, [1]. Lugar en Manning: http://www.manning.com/koenig/.
  También en Safari: http://my.safaribooksonline.com/9781932394849.
- PLEAC - Programming Language Examples Alike Cookbook
- Rosetta Code is a programming chrestomathy site
- Scripting in Java por Dejan Bosanac http://www.scriptinginjava.net/
- http://grails.org.mx/codice/
- http://groovy.org.es/
- http://mrhaki.blogspot.com/search/label/Groovy
- http://www.davioth.com/
- http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=practically+groovy:
- http://docs.codehaus.org/display/GROOVY/Testing+Guide
- Practically Groovy: Unit test your Java code faster with Groovy http://www.ibm.com/developerworks/java/library/j-pg11094/
- http://www.vogella.de/articles/Eclipse/article.html

1.2. Instalación de `groovy`

Comenzando: Java

> *Groovy requires Java, so you need to have a version available (1.4 or greater is required). Here are the steps if you don't already have Java installed:*
>
> - *Get the latest Java distribution from the http://java.sun.com website.*
> - *Run the installer.*
> - *Set the* `JAVA_HOME` *environment variables. On Windows, follow these steps:*
>   - *Open the System control panel*
>   - *Click the Advanced tab*
>   - *Click the Environment Variables button*
>   - *Add a new System variable with the name* `JAVA_HOME` *and the value of the directory Java was installed.*
>   - *Optionally add* `%JAVA_HOME%\bin` *to your system path*

Puesta a punto del entorno

Download the Groovy installer or binaries from the downloads page and follow the installation instructions. (There is currently an issue where you cannot have spaces in the path where Groovy is installed under windows. So, instead of accepting the default installation path of `c:\Program Files\Groovy` *you will want to change the path to something like* `c:\Groovy`*)*

OR

- *Get a copy of the Groovy distribution from the website, and copy it to some place on your hard drive.*
- *Unzip the groovy archive to some logical place on your hard drive.*
- *Set the* `GROOVY_HOME` *environment variables.*
- *Optionally add* `%GROOVY_HOME%\bin` *to your system path*
- *Try opening* `groovyConsole.bat` *by double clicking on the icon in the bin directory of the Groovy distribution. If it doesn't work, open a command prompt, and change to the bin directory and run it from there to see what the error message is.*

Instalación de archivos jar opcionales

You may wish to obtain optional jar files, either corresponding to Groovy modules (see module documentation for details) or corresponding to other Java classes you wish to make use of from Groovy. Some possibilities are listed below:

Name	From	Description
jtds-version.jar	http://jtds.sourceforge.net	Database driver for SQL Server and/or Sybase
hsqldb-version.jar	http://www.hsqldb.org/	Database driver for HSQLDB, a 100% Java database

The recommended way for making Groovy be aware of your additional jar files is to place them in a predefined location. Your Groovy install should include a file called `groovy-starter.conf`*.*

Here is mine:

```
root@nereida:/usr/local/src/groovy/groovy-1.6.5/target/install/conf# ls -l
total 4
-rw-r--r-- 1 root root 893 2009-11-11 16:39 groovy-starter.conf
root@nereida:/usr/local/src/groovy/groovy-1.6.5/target/install/conf# cat -n groovy-starter.conf
     1	##############################################################################
     2	##                                                                          ##
     3	##  Groovy Classloading Configuration                                       ##
     4	##                                                                          ##
     5	##############################################################################
     6	
     7	##
     8	## $Revision: 9225 $ $Date: 2007-11-15 21:17:45 +0100 (Do, 15. Nov 2007) $
     9	##
    10	## Note: do not add classes from java.lang here. No rt.jar and on some
    11	##       platforms no tools.jar
    12	##
    13	## See http://groovy.codehaus.org/api/org/codehaus/groovy/tools/LoaderConfiguration.html
    14	## for the file format
    15	
    16	    # load required libraries
    17	    load !{groovy.home}/lib/*.jar
```

```
# load user specific libraries
load !{user.home}/.groovy/lib/*.jar

# tools.jar for ant tasks
load ${tools.jar}
```

Within that file, make sure a line such as

```
load ${user.home}/.groovy/lib/*
```

is not commented out. The `user.home` *system property is set by your operating system. Now simply place your jar files into the* `.groovy/lib` *directory.*

As an alternative, you can set up a `CLASSPATH` *variable and make sure it mentions all of your additional* `jar` *files, otherwise Groovy works fine with an empty or no* `CLASSPATH` *variable.*

Programa de prueba

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ cat hello.groovy
println "Hello World!"
```

Makefile

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ cat Makefile
run: hello.class
	java -classpath /usr/local/src/groovy/groovy-1.6.5/target/install/embeddable/groovy-all-1.6.5.jar:. hello
hello.class: hello.groovy
	groovyc hello.groovy
clean:
	rm -f hello.class
```

Variables de entorno

```
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk/
PATH=$PATH:/usr/local/src/groovy/groovy-1.6.5/target/install/bin/
GROOVY_HOME=/usr/local/src/groovy/groovy-1.6.5/target/install/
```

Ejecución

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ make clean
rm -f hello.class
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ make
groovyc hello.groovy
java -classpath /usr/local/src/groovy/groovy-1.6.5/target/install/embeddable/groovy-all-1.6.5.jar:. hello
Hello World!
```

1.3. `groovysh`

1.3.1. Introducción

Posiblemente la forma mas rápida de ejecutar `groovy` sea proveer el programa en la línea de comandos:

```
casiano@tonga:~$ groovy -e "println 'Hello ' + args[0]" World
Hello World
```

Sin embargo, la shell `groovysh` permite la experimentación con `groovy`. Comencemos:

```
casiano@tonga:~/src/groovy$ groovysh
Groovy Shell (, JVM: 1.6.0_0-b11)
Type 'help' or '\h' for help.
---------------------------------------------------------------------------------------------------------------------------------------------------
groovy:000> \h

For information about Groovy, visit:
    http://groovy.codehaus.org

Available commands:
  help     (\h) Display this help message
  ?        (\?) Alias to: help
  exit     (\x) Exit the shell
  quit     (\q) Alias to: exit
  import   (\i) Import a class into the namespace
  display  (\d) Display the current buffer
  clear    (\c) Clear the buffer
  show     (\S) Show variables, classes or imports
  inspect  (\n) Inspect a variable or the last result with the GUI object browser
  purge    (\p) Purge variables, classes, imports or preferences
  edit     (\e) Edit the current buffer
  load     (\l) Load a file or URL into the buffer
  .        (\.) Alias to: load
  save     (\s) Save the current buffer to a file
  record   (\r) Record the current session to a file
  history  (\H) Display, manage and recall edit-line history
  alias    (\a) Create an alias
  set      (\=) Set (or list) preferences

For help on a specific command type:
    help command

groovy:000>
```

Podemos pedir ayuda sobre un comando específico:

```
groovy:000> help show

usage: show [<variables|classes|imports|preferences|all>]

Show variables, classes, imports or preferences.
```

1.3.2. Expresiones

Es posible evaluar expresiones simples:

```
groovy:000> println "Hello"
Hello
===> null
```

```
groovy:000> 2+4
===> 6
```

El resultado de una evaluación queda en la variable especial _:

```
groovy:000> _
===> 6
```

```
class Foo {
    def bar() {
        println "baz"
    }
}
```

Es posible introducir expresiones complejas que ocupen varias lineas (como una definición de una clase). Cuando la shell detecta que la expresión ha sido completada pasa a compilarla y evaluarla:

```
groovy:000> class Foo {
groovy:001>     def bar() {
groovy:002>         println "baz"
groovy:003>     }
groovy:004> }
===> true
```

Ahora podemos usar la definición anterior:

```
groovy:000> foo = new Foo()
===> Foo@b0095d
groovy:000> foo.bar()
baz
===> null
groovy:000>
```

Las variables de la shell no tiene tipo. Esta asignación define una variable shell:

```
groovy:000> foo = "bar"
===> bar
groovy:000> foo
===> bar
```

Esta definición sin embargo evalúa una variable local que no será guardada en el entorno de la shell:

```
groovy:000> def tutu = "titi"
===> titi
groovy:000> tutu
ERROR groovy.lang.MissingPropertyException: No such property: tutu for class: groovysh_evaluate
```

Las variables así definidas tienen el ámbito de la línea:

```
groovy:000> def tutu = "titi"; println tutu
titi
===> null
```

Es posible definir funciones y usarlas luego:

```
groovy:000> def hello(name) {
groovy:001>   println("Hello $name")
groovy:002> }
===> true
groovy:000> hello('LHP')
Hello LHP
===> null
```

1.3.3. Comandos

La shell dispone de un cierto número de comandos que proporcionan acceso a su entorno. Cada comando tiene una forma abreviada (algo como `\h`). Los usuarios pueden crear sus propios alias.

help

Ya vimos el comando `help` que nos permite ver la lista de comandos:

```
groovy:000> help

For information about Groovy, visit:
    http://groovy.codehaus.org

Available commands:
  help     (\h) Display this help message
  ?        (\?) Alias to: help
  exit     (\x) Exit the shell
  quit     (\q) Alias to: exit
  import   (\i) Import a class into the namespace
  display  (\d) Display the current buffer
  clear    (\c) Clear the buffer
  show     (\S) Show variables, classes or imports
  inspect  (\n) Inspect a variable or the last result with the GUI object browser
  purge    (\p) Purge variables, classes, imports or preferences
  edit     (\e) Edit the current buffer
  load     (\l) Load a file or URL into the buffer
  .        (\.) Alias to: load
  save     (\s) Save the current buffer to a file
  record   (\r) Record the current session to a file
  history  (\H) Display, manage and recall edit-line history
  alias    (\a) Create an alias
  set      (\=) Set (or list) preferences

For help on a specific command type:
    help command
```

exit

Nos permite salir de la shell. Esta es la forma de salir de la shell. Es posible salir pulsando `ctrl-C` pero la shell se quejará:

```
groovy:000> exit
casiano@tonga:~/src/groovy$ groovysh
Groovy Shell (, JVM: 1.6.0_0-b11)
Type 'help' or '\h' for help.
--------------------------------------------
groovy:000> quit
casiano@tonga:~/src/groovy$ groovysh
Groovy Shell (, JVM: 1.6.0_0-b11)
Type 'help' or '\h' for help.
---------------------------------------------
groovy:000> ^C
WARNING: Abnormal JVM shutdown detected
casiano@tonga:~/src/groovy$
```

import
Para añadir un `import` que será incluido en las evaluaciones de la shell.

```
groovy:000> import java.util.regex.Pattern
groovy:000> p = Pattern.compile("a+b")
===> a+b
groovy:000> m = p.matcher("aaaaab")
===> java.util.regex.Matcher[pattern=a+b region=0,6 lastmatch=]
groovy:000> b = m.matches()
===> true
groovy:000> m = p.matcher("b")
===> java.util.regex.Matcher[pattern=a+b region=0,1 lastmatch=]
groovy:000> b = m.matches()
===> false
```

display
Muestra los contenidos del buffer actual.
Muestra únicamente el buffer de una expresión incompleta.

```
groovy:000> class Foo {
groovy:001>   def bar
groovy:002> def baz () {
groovy:003> display
 001> class Foo {
 002>   def bar
 003> def baz () {
groovy:003>   println "Hola"
groovy:004> }}
===> true
groovy:000> display
Buffer is empty
```

clear Limpia el buffer actual

```
groovy:000> class Foo {
groovy:001> display
 001> class Foo {
groovy:001> clear
groovy:000> display
Buffer is empty
```

show
Muestra variables, clases, preferencias o imports:

```
groovy:000> a=4
===> 4
groovy:000> show variables
Variables:
  a = 4
  _ = 4
groovy:000> class Foo {}
===> true
groovy:000> show classes
```

```
Classes:
  class Foo
groovy:000> import java.util.regex.Pattern
groovy:000> show imports
Custom imports:
  import java.util.regex.Pattern
groovy:000> set verbosity VERBOSE
groovy:000> show preferences
Preferences:
    editor=/Applications/TextEdit.app/Contents/MacOS/TextEdit
    editor="/usr/bin/vim"=true
    verbosity=VERBOSE
groovy:000> show all
Variables:
  a = 4
  _ = true
Classes:
  class Foo
Custom imports:
  import java.util.regex.Pattern
Preferences:
    editor=/Applications/TextEdit.app/Contents/MacOS/TextEdit
    editor="/usr/bin/vim"=true
    verbosity=VERBOSE
===> [variables, classes, imports, preferences, all]
```

inspect

Abre un browser gráfico que muestra el resultado de la última evaluación. Este browser es una interfaz de usario Swing que nos permite navegar en la API del objeto.

```
groovy:000> a = [ 1, 2, [3, 4], "a" ]
===> [1, 2, [3, 4], a]
groovy:000> inspect
groovy:000> h = [ a : 1, b : [3, 4], c : 5 ]
===> {a=1, b=[3, 4], c=5}
groovy:000> inspect
groovy:000> r = 1..5
===> 1..5
groovy:000> inspect
```

Podemos ver entonces los métodos del objeto y como usarlos:

```
groovy:000> h.size()
===> 3
groovy:000> h.each {k,v -> println "$k -> $v" }
a -> 1
b -> [3, 4]
c -> 5
===> {a=1, b=[3, 4], c=5}
groovy:000>
```

purge

Suprime objetos de la shell.

- `purge variables`

- purge classes
- purge imports
- purge preferences
- purge all

Véase un ejemplo:

```
groovy:000> show variables
Variables:
  b = 5
  a = 4
  _ = 5
groovy:000> purge variables
Custom variables purged
groovy:000> show variables
No variables defined
```

edit

Permite editar el buffer de trabajo con un editor externo. sólo funciona en Unix y debe establecerse la variable `EDITOR`.

```
casiano@tonga:~$ export EDITOR='gvim -f'
casiano@tonga:~$ groovysh
aGroovy Shell (, JVM: 1.6.0_0-b11)
Type 'help' or '\h' for help.
-----------------------------------
groovy:000> a =
groovy:001> display
 001> a =
groovy:001> edit
 001> a =
 002> a = "hello world"
===> hello world
groovy:000> a
 001> a
===> hello world
```

La opción `-f` de `gvim` hace que no se cree un proceso aparte.

load

Carga uno o mas ficheros en el buffer.

Consideremos el siguiente programa Java:

```
casiano@tonga:~/src/groovy$ cat Greetings.groovy
public class Greetings {
  public static void main(String[] args) {
    for(int i = 0; i < 3; i++) {
      System.out.print("jo! ");
    }
    System.out.println(".... ");
  }
}
```

Podemos ejecutarlo con `groovy` usando el comando `load` de la shell:

```
casiano@tonga:~/src/groovy$ groovysh
Groovy Shell (, JVM: 1.6.0_0-b11)
Type 'help' or '\h' for help.
-----------------------------------------
groovy:000> load Greetings. (pressed tab key)

Greetings.class     Greetings.groovy    Greetings.java
groovy:000> load Greetings.groovy
Loading: file:/home/casiano/src/groovy/Greetings.groovy
 001> public class Greetings {
 002>   public static void main(String[] args) {
 003>     for(int i = 0; i < 3; i++) {
 004>       System.out.print("jo! ");
 005>     }
 006>     System.out.println(".... ");
 007>   }
 008> }
===> true
groovy:000> Greetings.main()
 001> Greetings.main()
jo! jo! jo! ....
===> null
```

Ficheros asociados

- `$HOME/.groovy/groovysh.profile`: Este script, si existe es cargado cuando la shell comienza.
- `$HOME/.groovy/groovysh.rc`: Este script, si existe, es cargado cuando la shell entra en modo interactivo.
- `$HOME/.groovy/groovysh.history`: Contiene el histórico de las líneas de edición

1.3.4. Ejecución en línea de comandos

The shell supports several options to control verbosity, ANSI coloring and other features:

```
casiano@millo:~/src/groovy$ groovysh --help
usage: groovysh [options] [...]
  -C, --color[=FLAG]         Enable or disable use of ANSI colors
  -D, --define=NAME=VALUE    Define a system property
  -T, --terminal=TYPE        Specify the terminal TYPE to use
  -V, --version              Display the version
  -d, --debug                Enable debug output
  -h, --help                 Display this help message
  -q, --quiet                Suppress superfluous output
  -v, --verbose              Enable verbose output
```

In addition to options, commands or expressions can be given on the command-line which will invoke the shell in non-interactive mode. The commands or expressions will be evaluated and the shell will exit. If no additional arguments are given the shell will startup interactively.

Sigue un ejemplo de ejecución de un comando:

```
casiano@millo:~/src/groovy$ groovysh 'show preferences'
No preferences are set
```

Sigue un ejemplo de evaluación de una expresión:

```
casiano@tonga:~/src/groovy$ groovysh 'System.properties.each { k, v -> println("$k = $v") }'
java.runtime.name = OpenJDK  Runtime Environment
sun.boot.library.path = /usr/lib/jvm/java-6-openjdks/jre/lib/i386
java.vm.version = 1.6.0_0-b11
java.vm.vendor = Sun Microsystems Inc.
java.vendor.url = http://java.sun.com/
path.separator = :
java.vm.name = OpenJDK Client VM
file.encoding.pkg = sun.io
sun.java.launcher = SUN_STANDARD
user.country = US
sun.os.patch.level = unknown
program.name = groovysh
java.vm.specification.name = Java Virtual Machine Specification
user.dir = /home/casiano/src/groovy
java.runtime.version = 1.6.0_0-b11
java.awt.graphicsenv = sun.awt.X11GraphicsEnvironment
java.endorsed.dirs = /usr/lib/jvm/java-6-openjdks/jre/lib/endorsed
os.arch = i386
javax.accessibility.assistive_technologies = org.GNOME.Accessibility.JavaBridge
java.io.tmpdir = /tmp
line.separator =
.....
```

1.4. Práctica: groovysh

Utilizando `groovysh`:

- Escriba una función que calcule el área de un círculo y úsela (Class Math)

  ```
  ~$ groovysh
  Groovy Shell (1.6.5, JVM: 1.6.0_17)
  Type 'help' or '\h' for help.
  -------------------------
  groovy:000> Math.sqrt(16)
  ===> 4.0
  groovy:000> Math.PI
  ===> 3.141592653589793
  ```

- Utilice el comando `inspect` para averiguar que métodos tiene un objeto `String`. ¿Que hace el método `center`? ¿Cómo funciona?

- ¿Que hace el método `reverse`? ¿Cómo funciona? Escriba una función que devuelva `true` si, y sólo si su argumento es un palindromo. El operador `==` puede ser utilizado para comprobar que dos objetos son iguales.

- ¿Que hace el método `times` de un número? ¿Que argumentos recibe? Ejecute este código:

  ```
  groovy:000> 3.times { print "jo! " }; println ".... "
  ```

  ¿Cual es la salida? ¿Que es `times`: un método o una palabra reservada del lenguaje? Compare este código con el código Java equivalente:

```
public class Greetings {
  public static void main(String[] args) {
    for(int i = 0; i < 3; i++) {
      System.out.print("jo! ");
    }
    System.out.println(".... ");
  }
}
```

1.5. La Cónsola Swing Groovy

Introducción

Para arrancar la cónsola Groovy:

```
~/src/groovy$ groovyConsole &
```

1. La Cónsola tiene un área de entrada y un área de salida
2. Tecleamos nuestro código Groovy en el área de entrada
3. Cuando seleccionamos `"Run"` en el menu `"Scripts"`, la cónsola compila el script y lo ejecuta
4. Cualquier cosa que normalmente sería enviada a `System.out` es mostrada en el área de salida
5. Si el script retorna un valor no nulo también es impreso

El manejo por la Cónsola de los flujos es bastante elemental. Se redirige STDOUT pero no STDERR ni STDIN que siguen asociados con la terminal desde la que se arrancó `groovyConsole`. Evite usar `groovyConsole` con programas que lean desde `STDIN`.

Recetas

1. Use `ctrl-Enter` y `ctrl-R` como sinónimos de `"Run Script"`.
2. Si seleciona un texto del área de entrada y se pulsa `ctrl-Shift-Enter`, Groovy ejecutará ese texto
3. El resultado de un script es el valor de la última expresión evaluada
4. Podemos activar desactiva la captura de `System.out` en el menú `View`
5. Pulse `ctrl-W` para limpiar el área de salida

Edición de Ficheros

Podemos editar fichero siguiendo estos pasos:

1. Selecciona File -¿Open (atajo: `ctrl-O`) para abrir un fichero
2. Selecciona File -¿Save (atajo: `ctrl-S`) para guardar un fichero
3. Selecciona File -¿New File (atajo: `ctrl-N`) para empezar con un área de entrada en blanco

Historia y Resultados

1. Podemos arrancar un inspector gráfico sobre el último resultado en el menu `Script` (`ctrl-I`) en `"Inspect Last"` y `"Inspect Variables"` (`ctrl-J`)
2. La cónsola recuerda las últimas ejecuciones. Se puede navegar en la historia usando `ctrl-,` y `ctrl-.`
3. El último resultado (non-null) se guarda en la variable `'_'` (underscore).
4. El último resultado (null y non-null) para cada ejecución es almacenado en la variable lista `'__'` (dos underscores). El resultado e la última ejecución es `__[-1]`, el penúltimo es `___[-2]`, etc.

1.6. Compilación conjunta de código Groovy y Java

Código Java

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ cat -n AJavaClass.java
     1  // Some Java code
     2  public class AJavaClass
     3  {
     4    {
     5      System.out.println("Created Java Class");
     6    }
     7
     8    public void sayHello() {
     9      System.out.println("Hello");
    10    }
    11  }
```

Código Groovy

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ cat -n UseJavaClass.groovy
     1  new AJavaClass().sayHello()
```

Usándolos juntos

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ groovyc -j AJavaClass.java -Jsource 1.6
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ groovyc UseJavaClass.groovy
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ java -classpath /usr/local/src/groovy/groovy-1.6.5/target/install/embeddable/groovy-all-1.6.5.jar:. UseJavaClass
Created Java Class
Hello
```

1.7. Usando clases Java desde Groovy

El uso de clases Java por un programa Groovy es directo cuand las clases Java forman parte de JDK: se importan las clases o los paquetes de la misma forma que se hace en Java. De hecho el intérprete Groovy ya importa un cierto número de clases y paquetes:

```
* java.io.*
* java.lang.*
* java.math.BigDecimal
* java.math.BigInteger
* java.net.*
* java.util.*
* groovy.lang.*
* groovy.util.*
```

Veamos un ejemplo para clases que no están en JDK.

Estructura de la aplicación
El programa cliente es `UseGreetJava.groovy` y la clase Java está en `java/com/agiledeveloper/GreetJava.java`:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ tree
.
|-- UseGreetJava.groovy
|-- java
    |
    `-- com
        `-- agiledeveloper
            `-- GreetJava.class
```

La Clase Java
Estos son los contenidos de la clase Java:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ cat -n java/com/agiledeveloper/GreetJava.java
     1  package com.agiledeveloper;
     2
     3  public class GreetJava {
     4    public static void sayHello() {
     5      System.out.println("Hello from Java class GreetJava");
     6    }
     7  }
```

Compilación de la clase Java
Comenzamos compilandola con javac, lo que nos genera el correspondiente fichero `.class`:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ cd java/com/agiledeveloper/
generaciondecodigos@nereida:~/src/groovy/separatedcomp/java/com/agiledeveloper$ javac GreetJava.java
generaciondecodigos@nereida:~/src/groovy/separatedcomp/java/com/agiledeveloper$ ls -ltr
total 8
-rw-r--r-- 1 generaciondecodigos generaciondecodigos 153 2009-11-12 12:53 GreetJava.java
-rw-r--r-- 1 generaciondecodigos generaciondecodigos 441 2009-11-12 16:42 GreetJava.class
```

El cliente Groovy
Los contenidos del script Groovy se limitan a una llamada al método `sayHello`:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ cat -n UseGreetJava.groovy
     1  com.agiledeveloper.GreetJava.sayHello();
```

Ejecución
Para ejecutar el programa es necesario que el intérprete groovy encuentre la clase. Añadimos la opción `-classpath java/` para indicarle que incluya dicho directorio en el camino de búsqueda de librerías. Esto es sólo necesario si el fichero de la clase está en un directorio distinto que el del código Groovy.

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp$ groovy -classpath java/ UseGreetJava.groovy
Hello from Java class GreetJava
```

1.8. Práctica: Llamada a una Clase Java desde Groovy

- Escriba una clase Java que provee un método y úsela desde un script Groovy

1.9. LLamando scripts Groovy desde código Groovy

La clase GroovyShell provee las funcionalidades necesarias para ejecutar scripts controlando su entorno de ejecución.

Llamada simple
En este ejemplo, el script `Script2.groovy`

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat -n Script2.groovy
     1  println "In Script2"
     2  shell = new GroovyShell()
     3  shell.evaluate(new  File('Script1.groovy'))
```

llama al script `Script1.groovy`:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat -n Script1.groovy
     1  println "Hello from Script1"
```

Para ello primero se crea un objeto GroovyShell y después se llama al método `evaluate`: En el ejemplo `evaluate` recibe una cadena conteniendo el nombre de fichero, pero es posible obtener el script de diversas formas:

- `Object   evaluate(File file)`
  Evaluates some script against the current Binding and returns the result
- `Object         evaluate(GroovyCodeSource codeSource)`
  Evaluates some script against the current Binding and returns the result
- `Object         evaluate(Reader in)`
  Evaluates some script against the current Binding and returns the result
- `Object         evaluate(Reader in, String fileName)`
  Evaluates some script against the current Binding and returns the result
- `Object         evaluate(String scriptText)`
  Evaluates some script against the current Binding and returns the result
- `Object         evaluate(String scriptText, String fileName)`
  Evaluates some script against the current Binding and returns the result
- `Object         evaluate(String scriptText, String fileName, String codeBase)`
  Evaluates some script against the current Binding and returns the result.

Sigue el resultado de una ejecución:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ groovy Script2.groovy
In Script2
Hello from Script1
```

También es posible llamar simplemente a `evaluate`:

```
$ cat -n Script3.groovy
     1  println "In Script3"
     2  evaluate(new  File('Script1.groovy'))
$ groovy Script3.groovy
In Script3
Hello from Script1
```

LLamada con parámetros y retorno de resultados

Este script hace uso de una variable `$name` que definiremos en otro script:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat -n Script1a.groovy
     1  println "Hello $name"
     2  name = "Dan"
```

Aunque los atributos y variables locales deben ser declarados antes de su uso dentro de una clase, en los scripts es posible usar variables no declaradas. En ese caso se asume que las variables vienen del binding del script y que serán añadidas a este si aún no están allí.

> *In computer science, binding is the creation of a simple reference to something that is larger and more complicated and used frequently. The simple reference can be used instead of having to repeat the larger thing. A binding is such a reference*
>
> *The terms language binding and name binding refer to both the linking of libraries to application programs, and to the way symbols (variable names) are handled by compilers.*
>
> *In programming languages, these terms mean the references of an identifier to a value. In this context, binding is the act of associating a name or symbol with a machine address, and this association may occur either at compile time, in which case it is called 'static linking', or it may occur dynamically at runtime, which is called 'dynamic linking'. The terms binding and linking are used interchangeably in some contexts.*
>
> *Static binding has an additional characteristic: it prevents libraries from being updated (recompiled) independently of applications.*

El siguiente script crea un objeto `GroovyShell` pasándole como argumento el objeto Binding del script actual. Este objeto representa los binding (el mapa) de las variables en el script. Al pasarlo como argumento conseguimos que las variables pueden ser alteradas desde fuera del script y creadas fuera y pasadas al mismo. Citando la documentación:

> *The public class Binding represents the variable bindings of a script which can be altered from outside the script object or created outside of a script and passed into it.*

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat -n Script2a.groovy
     1  println "In Script2a"
     2  name = "Juana"
     3
     4  shell = new GroovyShell(binding)
     5  result = shell.evaluate(new File('Script1a.groovy'))
     6
     7  println "Script1a returned: $result"
     8  println "Hello $name"
```

En efecto:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ groovy Script2a.groovy
In Script2a
Hello Juana
Script1a returned: Dan
Hello Dan
```

El valor retornado por `evaluate` es el valor retornado por el script.

Protegiendo el Entorno del LLamador En el ejemplo anterior pasamos el binding del llamador al script llamado. Crearemos un nuevo binding si queremos aislar la ejecución del script llamado:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat Script5.groovy
Binding binding = new Binding();
binding.setVariable("foo", new Integer(2));
GroovyShell shell = new GroovyShell(binding);

Object value = shell.evaluate(new File('Script1b.groovy'))
println value
println binding.getVariable("x")

assert value.equals(new Integer(20));
assert binding.getVariable("x").equals(new Integer(123));
```

Estos son los contenidos del script `Script1b.groovy`:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat Script1b.groovy
println 'Hello World!'
x = 123
return foo * 10
```

Al ejecutar `Script5.groovy` vemos como la variable `foo` está en el binding de `Script1b.groovy` cuando éste se ejecuta:

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ groovy Script5.groovy
Hello World!
20
123
```

Además `Script5.groovy` obtiene e imprime el valor retornado por `Script1b.groovy` y accede a la variable `x` del mismo.

El Método `run`

Utilizaremos el método `run` en vez de `evaluate` si se quieren pasar los argumentos usando la línea de comandos:

- `Object run(File scriptFile, List list)`
  A helper method which runs the given script file with the given command line arguments
- `Object run(File scriptFile, String[] args)`
  Runs the given script file name with the given command line arguments
- `Object run(Reader in, String fileName, String[] args)`
  Runs the given script with command line arguments
- `Object run(String scriptText, String fileName, List list)`
  A helper method which runs the given cl script with the given command line arguments
- `Object run(String scriptText, String fileName, String[] args)`
  Runs the given script text with command line arguments

El siguiente ejemplo ilustra el modo de uso. Código del llamador:

```
~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat -n run.groovy
     1  binding = new Binding()
     2  binding.setVariable("name", "Juan")
     3
     4  shell = new GroovyShell(binding)
     5
     6  value = shell.run(new File('Script1c.groovy'), ['a', 'b'])
     7
     8  println "Script1c returned ${value}"
     9
    10  println "z = ${binding.getVariable("z")}"
```

Este es el código del script llamado:

```
~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat -n Script1c.groovy
     1  println "Script1c: args received ${args}"
     2  println "Script1c: Hello $name"
     3  name = "Dan"
     4  z = 4
     5  8
```

y este el resultado de la ejecución:

```
~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ groovy run.groovy
Script1c: args received [a, b]
Script1c: Hello Juan
Script1c returned 8
z = 4
```

El método `parse`

El método `parse` de los objetos GroovyShell retorna un objeto de la clase Script. Estos objetos disponen de un método `run`:

```
$ cat parseandrun.groovy
Binding binding = new Binding()
binding.setVariable("x", new Integer(2))
def shell = new GroovyShell(binding)

def script = shell.parse(
'''
  x++
  println x
'''
)

10.times {
  script.run()
}
```

Cuando se ejecuta el programa anterior produce el siguiente resultado:

```
$ groovy parseandrun.groovy
3
4
5
6
```

```
7
8
9
10
11
12
```

Ejemplo: Una Calculadora

El siguiente programa pone en práctica lo aprendido para proveer una calculadora. Las cadenas leídas en `stdin` son evaluadas dentro de un script Groovy y el resltado es mostrado por pantalla:

```
~/Lgroovy/separatedcomp/groovyscriptsfromgroovy$ cat -n calc.groovy
def stdin = new BufferedReader(new InputStreamReader(System.in))
Binding binding = new Binding()
def shell = new GroovyShell(binding)

int c = 1;
print "> "
stdin.eachLine{ input ->
  def script = shell.parse(input)
  def a = script.run()
  println a
  print "> "
}
```

sigue un ejemplo de ejecución:

```
generaciondecodigos@nereida:~/Lgroovy/separatedcomp/groovyscriptsfromgroovy$ groovy calc.groovy
> a =2*3
6
> a+2
8
> b = a-3
3
> Math.sqrt(b)
1.7320508075688772
> ^D
```

Hemos pulsado la combinación `CTRL-D` para producir la señal de final de entrada.

Véase

- http://groovy.codehaus.org/Embedding+Groovy
- La documentación de GroovyShell
- La documentación de Script

1.10. Práctica: Guión controlado

- Escriba un guión que Groovy que controle a otro guión Groovy (`hello world argumentos`) pasándole los argumentos en la línea de comandos

1.11. LLamando Clases Groovy desde Java

Makefile

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyfromjava$ cat -n Makefile
run: CrearAlumnoJava.class
        java -classpath /usr/local/src/groovy/groovy-1.6.5/target/install/embeddable/groovy-all-1.6.5.jar:. CrearAlumnoJava

CrearAlumnoJava.class: CrearAlumnoJava.java Alumno.java AlumnoGroovy.groovy CrearAlumnoGroovy.groovy
        javac Alumno.java
        groovyc AlumnoGroovy.groovy
        groovyc CrearAlumnoGroovy.groovy
        javac CrearAlumnoJava.java

clean:
        rm -f *.class
```

CrearAlumnoJava.java

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyfromjava$ cat -n CrearAlumnoJava.java
public class CrearAlumnoJava {
  public static void main (String agrs[]) {
    /**Con Java**/
    Alumno al = new Alumno();
    al.setNombre("Antonio");
    al.setCodigo(1);
    System.out.println("Alumno Creado con Java: "+ al.getNombre());

    /**Con Groovy**/
    AlumnoGroovy ag = new AlumnoGroovy();
    ag.setNombre("Jose");
    ag.setCodigo(2);
    // En .groovy no hay definido explicitamente un set: es creado automaticamente
    System.out.println("Alumno Creado con Groovy: "+ ag.getNombre());
  }
}
```

Alumno.java

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyfromjava$ cat -n Alumno.java
public class Alumno {
  private String nombre;
  private Integer codigo;

  public String getNombre () {
    return nombre;
  }
  public void setNombre (String nombre) {
    this.nombre= nombre;
  }
  public Integer getCodigo () {
    return codigo;
  }
  public void setCodigo (Integer codigo) {
```

```
    this.codigo=codigo;
  }
}
```

AlumnoGroovy.groovy

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyfromjava$ cat -n AlumnoGroovy.groovy
class AlumnoGroovy{
  String nombre
  Integer codigo
}
```

CrearAlumnoGroovy.groovy

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyfromjava$ cat -n CrearAlumnoGroovy.groovy
class CrearAlumnoGroovy {

  public static void main(String []agrs) {
/*con Java*/
    def al=new Alumno(nombre:"Antonio", codigo:1)
    println "Nombre del Alumno Java: " + al.nombre

/*con Groovy*/
    def ag=new AlumnoGroovy(nombre:"Jose", codigo:2)
    println "Nombre del Alumno Groovy: "+ag.nombre
  }
}
```

Ejecución

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyfromjava$ make
javac Alumno.java
groovyc AlumnoGroovy.groovy
groovyc CrearAlumnoGroovy.groovy
javac CrearAlumnoJava.java
java -classpath /usr/local/src/groovy/groovy-1.6.5/target/install/embeddable/groovy-all-1.6.5.jar:. CrearAlumnoJava
Alumno Creado con Java: Antonio
Alumno Creado con Groovy: Jose
```

Capítulo 2

Tipos de Datos Simples

2.1. El Sistema de tipos de Java

Tipos Primitivos y de Referencia

Java distingue entre tipos primitivos `int`, `double`, `char` y `byte`, y los tipos de *referencia* (tales como `Object` y `String`). Los tipos primitivos son fijos y son los únicos que tienen *semantica de valor*. No es posible crear tipos-valor en Java. No es posible llamar a un método en un valor de un tipo primitivo. Las variables con tipos *referencia* contienen una referencia a un objeto.

Tipos Envoltorio

Para facilitar el paso de tipos primitivos a tipos referencia, Java provee un tipo wrapper para cada tipo primitivo. Por ejemplo, el wrapper para el tipo `int` es `java.lang.Integer`. Tales tipos de datos se denominan *type objects*. La wikipedia dice lo siguiente sobre los type objects:

> *In computer science, an object type (a.k.a. wrapping object) is a datatype which is used in object-oriented programming to wrap a non-object type to make it look like a dynamic object.*
>
> *Some object-oriented programming languages make a distinction between reference and value types, often referred to as objects and non-objects on platforms where complex value types don't exist, for reasons such as runtime efficiency and syntax or semantic issues. For example, Java has primitive wrapper classes corresponding to each primitive type: Integer and int, Character and char, Float and float, etc. Languages like C++ have little or no notion of reference type; thus, the use of object type is of little interest.*

El Problema

Por otro lado, los operadores sobre los tipos primitivos no se soportan sobre los correspondientes tipos referencia. Esto daba lugar en las versiones de Java anteriores a J2SE 5.0 a códigos innecesariamente complejos, como éste en que se suman dos listas en una tercera:

```
~/Lgroovy/simpledatatypes$ cat -n Sum.java
// Java code
import java.io.*;
import java.util.*;
public class Sum {
    public static void main (String [] args) {
      ArrayList listOne = new ArrayList();
      ArrayList listTwo = new ArrayList();
      ArrayList results = new ArrayList();

      int s = (args.length >  0)? Integer.parseInt(args[0]) : 5;
      for (int i = 1; i < s; i++) {
```

```
     System.out.println("  - Storing (" + i + ")");
     listOne.add(new Integer(i));
     // listTwo = 2*listOne
     listTwo.add(new Integer(2 * ((Integer) listOne.get(i-1)).intValue()));
   }

   for(int i = 0; i < listOne.size(); i++) {
    Integer first = (Integer) listOne.get(i);
    Integer second = (Integer) listTwo.get(i);

    int sum = first.intValue()+second.intValue();
    results.add (new Integer(sum));
   }

   for (int i = 0; i < results.size(); i++) {
     System.out.println("  - results(" + i + ") = " + results.get(i));
   }
 }
}
```

Autoboxing en Groovy

> *Autoboxing is the term for treating a value type as a reference type without any extra source code. The compiler automatically supplies the extra code needed to perform the type conversion.*

En Groovy, todos los tipos son objetos. Las operaciones de boxing y unboxing son realizadas automáticamente. Este es un programa Groovy equivalente al anterior:

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ cat Sum.groovy
#!/usr/bin/env groovy
a = (1..10).toList()                # objeto IntRange -> ArrayList
b = a.collect { 2*it }              # objeto ArrayList
r = (0..9).collect { a[it]+b[it] }   # objeto IntRange -> ArrayList
println r
```

Damos permisos de ejecución al fichero y ejecutamos:

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ chmod a+x Sum.groovy
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ ./Sum.groovy
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

O usando `groovysh`:

```
groovy:000> (c, d) = [(1..10).toList(), (1..10).collect {2*it }]
===> [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]]
groovy:000> println (0..9).collect { c[it]+d[it] }
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Véanse las entradas Multiple Assignment y Collections en groovy.codeHaus.org asi como la documentación de la clase `Object` del Groovy JDK.

Groovy siempre usa la clase wrapper en vez de la clase primitiva, pero nos permite utilizar las operaciones sobre los tipos básicos. La conversión de un tipo primitivo en un tipo wrapper se conoce como *boxing*. La acción inversa se denomina *unboxing*. Groovy realiza estas operaciones donde quiera que sea necesario.

Autoboxing en Java

En las versiones posteriores de Java a J2SE 5.0 es posible simplificar la suma, ya que se soporta *autoboxing*:

```
for(int i = 0; i < listOne.size(); i++) {
   results.add ((Integer) listOne.get(i) + (Integer) listTwo.get(i));
 }
```

Tipos Genéricos en Java

La presencia de genericidad, bucles `for-each` y autoboxing hace que en las últimas versiones de Java se simplifiquen estos problemas. El siguiente programa calcula las frecuencias de las palabras que se pasan en la línea de comandos:

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ cat -n Frequency.java
import java.util.*;

// Prints a frequency table of the words on the command line
public class Frequency {
   public static void main(String[] args) {
      Map<String, Integer> m = new TreeMap<String, Integer>();
      for (String word : args) {
          Integer freq = m.get(word);
          m.put(word, (freq == null ? 1 : freq + 1));
      }
      System.out.println(m);
   }
}
```

sigue un ejemplo de ejecución:

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ javac Frequency.java
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ java Frequency en todos los y son todos las operaciones de boxing y boxing son y
{boxing=2, de=1, en=1, las=1, los=1, operaciones=1, son=2, todos=2, y=3}
```

El programa declara primero un `Map` de `String` a `Integer`, implementado según un `TreeMap`, el cual asociará el número de veces que una palabra ocurre en la línea de comandos.

La declaración:

```
Map<String, Integer> m = new TreeMap<String, Integer>()
```

es un ejemplo de uso de genericos. Los genéricos nos proveen con un mecanismo para comunicar el tipo de una colección al compilador, de manera que su correcto uso pueda ser comprobado en tiempo de compilación. Como el compilador conoce ahora el tipo de un elemento de una colección puede insertar los ahormados o casts y comprobar que el uso de la colección es consistente con la declaración.

El siguiente fragmento está tomado del libro The Java Language Specification, Third Edition:

> *A class is generic if it declares one or more type variables. These type variables are known as the type parameters of the class. The type parameter section follows the class name and is delimited by angle brackets. It defines one or more type variables that act as parameters. A generic class declaration defines a set of parameterized types, one for each possible invocation of the type parameter section. All of these parameterized types share the same class at runtime.*
>
> *For instance, executing the code*
>
> ```
> Vector<String> x = new Vector<String>();
> Vector<Integer> y = new Vector<Integer>();
> boolean b = x.getClass() == y.getClass();
> ```

will result in the variable `b` *holding the value* `true`.

Cuando veamos algo como `<Type>` deberemos leer `"of Type"`. La declaración anterior se lee un `MAP of String to Integer`. Esto es mejor que usar un ahormado o cast, porque un cast nos informa de algo que el programador cree que es cierto en cierto punto del código y que la JVM deberá comprobar en tiempo de ejecución.

Bucles for-each en Java

Después se itera sobre cada argumento que aparece en la línea de comandos usando el nuevo bucle `for-each`.

```
for (String word : args) {
    Integer freq = m.get(word);
    m.put(word, (freq == null ? 1 : freq + 1));
}
```

En la tabla puede verse las equivalencias del nuevo bucle, dependiendo que lo que se recorra sea un `Arrays` o un `Iterator`. En ambos casos se requiere una variable extra, un índice para el array y un iterador para la colección.

<table>
<tr><th>Uso</th><th>Definición</th></tr>
<tr><td><pre>for (type var : arr) {
    body-of-loop
}</pre></td><td><pre>for (int i = 0; i < arr.length; i++) {
    type var = arr[i];
    body-of-loop
}</pre></td></tr>
<tr><td><pre>for (type var : coll) {
    body-of-loop
}</pre></td><td><pre>for (Iterator<type> iter = coll.iterator();
     iter.hasNext();)
{
    type var = iter.next();
    body-of-loop
}</pre></td></tr>
</table>

Autoboxing: valores de un hash

Para cada palabra `word`, obtenemos la correspondiente entrada en el `Map` (`freq = m.get(word)`) y actualizamos la entrada (`m.put(word, (freq == null ? 1 : freq + 1))`). Esta actualización es un caso de autoboxing. No es posible poner un `int` en el `Map`, ya que la signature de `put` es:

```
Object     put(Object key, Object value)
         Associates the specified value with the specified key in this map (optional operation).
```

y tampoco es posible añadir uno a un `Integer`.

Contando la frecuencia de aparición de una palabra: Versión Groovy

Aún así el correspondiente programa Groovy es mas pequeño y legible:

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ cat Frequency.groovy
#!/usr/bin/env groovy
def m = [:]
args.each { v = m[it]; m[it] = (v ?  ++v : 1) }
println(m);
```

Sigue una ejecución:

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ ./Frequency.groovy en todos son y son en en
[en:3, todos:1, son:2, y:1]
```

Véase También

- Sun Java tutorial
- The Java Language Specification, Third Edition

2.2. Práctica: Suma de Dos vectores en Java

Reescriba el programa de suma de dos vectores usando genéricos, bucles for-each y autoboxing

2.3. Especificación de Tipos Opcional

Groovy nos ofrece la posibilidad de especificar los tipos. La palabra reservada `def` se usa para declarar un objeto sin especificar su tipo.

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ cat -n optionalTyping.groovy
import org.codehaus.groovy.runtime.typehandling.GroovyCastException

def v = 3 //variable v can hold any value
println v
v = 'helicopter'
println v
v = false
println v
v = new StringBuffer()
println "<$v>"
v = null
println v

int i = 15 //variable i can only hold integer values
println i
i = 'A'
println i
try {
  i = false // produce una GroovyCastException
}
catch(e) {
   println e.class
}
```

Observe el resultado de la ejecución:

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ groovy optionalTyping.groovy
3
helicopter
false
<>
null
15
65
class org.codehaus.groovy.runtime.typehandling.GroovyCastException
```

2.4. Conversiones y Coerciones

A diferencia de Perl no existen conversiones automáticas de tipo, en el caso de que este se haya determinado. Es conveniente hacer una conversión explícita:

```
groovy:000> String a = 1; a = 4; a = a+1
===> 41
groovy:000> String a = 1; a = 4; a = a*2
===> 44
groovy:000> String a = 1; a = 4; a = a*3
===> 444
groovy:000> String a = 2; b = a.toInteger(); b = b*2
===> 4
```

Si la cadena contiene un número y se hace coerción explícita también funciona. Pero si la cadena no contiene un número se produce una excepción:

```
groovy:000> b = "432"
===> 432
groovy:000> b.class
===> class java.lang.String
groovy:000> c = b as Integer
===> 432
groovy:000> c.class
===> class java.lang.Integer
groovy:000> d = "432b"
===> 432b
groovy:000> e = d as Integer
ERROR java.lang.NumberFormatException: For input string: "432b"
        at groovysh_evaluate.run (groovysh_evaluate:3)
        ...
```

Sin embargo, si que se produce coerción implícita entre tipos numéricos. Se llama coerción a la promoción de uno de los argumentos de un operador al tipo mas general:

```
groovy:000> a = 1+1.0
===> 2.0
groovy:000> a.class
===> class java.math.BigDecimal
```

Podemos hacer la coerción explícita:

```
groovy:000> b = 1 as BigDecimal
===> 1
groovy:000> b.class
===> class java.math.BigDecimal
```

También funciona de número a String:

```
groovy:000> d = 65
===> 65
groovy:000> d.class
===> class java.lang.Integer
groovy:000> c = d as String
===> 65
groovy:000> c.class
===> class java.lang.String
```

```
groovy:000> c
===> 65
groovy:000> c[1]
===> 5
groovy:000> e = 8.5
===> 8.5
groovy:000> f = e as String
===> 8.5
groovy:000> f.class
===> class java.lang.String
groovy:000> f[1]
===> .
groovy:000> f[2]
===> 5
```

En realidad, la conversión a `String` funcionará siempre que la clase disponga de un método `toString`. En el siguiente ejemplo usamos una clase escrita por el usuario en la que hemos implementado el método `toString`:

```
groovy:000> import Complex
===> [import Complex]
groovy:000> c = new Complex(4.5, 1.0)
===> 4.5 + i
groovy:000> g = c as String
===> 4.5 + i
groovy:000> g.class
===> class java.lang.String
groovy:000> g[2]
===> 5
```

2.5. Práctica: Factorial

Escriba un programa que calcule el factorial de un número. Obtenga el número en la línea de comandos.

2.6. Sobreescritura y Sobrecarga de Operadores

Definiciones

Esto es lo que dice la wikipedia sobre Method overriding:

> *Method overriding, in object oriented programming, is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its superclasses. The implementation in the subclass overrides (replaces) the implementation in the superclass.*
>
> *A subclass can give its own definition of methods which also happen to have the same signature as the method in its superclass. This means that the subclass's method has the same name and parameter list as the superclass's overridden method. Constraints on the similarity of return type vary from language to language, as some languages support covariance on return types.*
>
> *Method overriding is an important feature that facilitates polymorphism*[1] *in the design of object-oriented programs.*

[1]The term "polymorphismrefers to the ability of code (in particular, methods or classes) to act on values of multiple types, or to the ability of different instances of the same data-structure to contain elements of different types.

> *Some languages allow the programmer to prevent a method from being overridden, or disallow method overriding in certain core classes. This may or may not involve an inability to subclass from a given class.*
>
> *In many cases, abstract classes are designed — i.e. classes that exist only in order to have specialized subclasses derived from them. Such abstract classes have methods that do not perform any useful operations and are meant to be overridden by specific implementations in the subclasses. Thus, the abstract superclass defines a common interface which all the subclasses inherit.*

En el caso de los operadores, un término mas convencional que redefinición o sobreescritura (overriding) es overloading o sobrecarga de operadores [1], [2]. La diferencia es que sobrecarga sugiere que hay múltiples instancias de un método (y por tanto del correspondiente operador asociado) que sólo difieren en el tipo de sus parámetros.

Operadores que pueden Sobrecargarse

Citemos la documentación de groovy en http://groovy.codehaus.org/Operator+Overloading:

> *Groovy supports operator overloading which makes working with* `Numbers, Collections, Maps` *and various other data structures easier to use.*
>
> *Various operators in Groovy are mapped onto regular Java method calls on objects. This allows you the developer to provide your own Java or Groovy objects which can take advantage of operator overloading. The following table describes the operators supported in Groovy and the methods they map to.*

```
Operator                Method
a + b                    a.plus(b)
a - b                    a.minus(b)
a * b                    a.multiply(b)
a ** b                   a.power(b)
a / b                    a.div(b)
a % b                    a.mod(b)
a | b                    a.or(b)
a & b                    a.and(b)
a ^ b                    a.xor(b)
a++ or ++a               a.next()
a-- or --a               a.previous()
a[b]                     a.getAt(b)
a[b] = c                 a.putAt(b, c)
a << b                   a.leftShift(b)
a >> b                   a.rightShift(b)
switch(a) { case(b) : }  b.isCase(a)
~a                       a.bitwiseNegate()
-a                       a.negative()
+a                       a.positive()
```

Operadores de Comparación y Manejo de Valores Nulos

> *Note that all the following comparison operators handle nulls gracefully avoiding the throwing of* `java.lang.NullPointerException`

```
Operator        Method
a == b          a.equals(b) or a.compareTo(b) == 0 **
a != b          ! a.equals(b)
a <=> b         a.compareTo(b)
```

```
a > b           a.compareTo(b) > 0
a >= b          a.compareTo(b) >= 0
a < b           a.compareTo(b) < 0
a <= b          a.compareTo(b) <= 0
```

Also in Groovy comparison operators handle `null`*s gracefully. So that* `a == b` *will never throw a* `NullPointerException` *whether* `a` *or* `b` *or both are null.*

```
groovy:000> a = null
===> null
groovy:000> b = "foo"
===> foo
groovy:000> a != b
===> true
groovy:000> b != a
===> true
groovy:000> a == null
===> true
```

Coerción y Operadores de Comparación

In addition when comparing numbers of different types the type coercion rules apply to convert numbers to the largest numeric type before the comparison. So the following is valid in Groovy

```
~/src/groovy/overloading$ cat -n Numberscohercioncomparison.groovy
#!/usr/bin/env groovy
Byte a = 12
Double b = 256

println  "a($a) instanceof Byte: ${a instanceof Byte}"
println  "b($b) instanceof Double: ${b instanceof Double}"

if (b > a) {
  println "(b > a): Cohercion working. $b is greater than $a"
  println  "($b > $a) instanceof Boolean: ${(b > a) instanceof Boolean}"
}

if (a > (Byte) b) {
  println "Byte $a is greater than (byte) Double $b"
  println  "($a > (Byte) $b) instanceof Boolean: ${(a > (Byte) b) instanceof Boolean}"
}
```

Al ejecutar este ejemplo obtenemos la salida:

```
~/src/groovy/overloading$ ./Numberscohercioncomparison.groovy
a(12) instanceof Byte: true
b(256.0) instanceof Double: true
(b > a): Cohercion working. 256.0 is greater than 12
(256.0 > 12) instanceof Boolean: true
Byte 12 is greater than (byte) Double 256.0
(12 > (Byte) 256.0) instanceof Boolean: true
```

Sobreescritura de la asignación En Java:

> *The* `equals()` *method of* `java.lang.Object` *acts the same as the* `==` *operator; that is, it tests for object identity rather than object equality. The implicit* <u>*contract*</u>[2] *of the* `equals()` *method, however, is that it tests for equality rather than identity. Thus most classes will override* `equals()` *with a version that does field by field comparisons before deciding whether to return* `true` *or* `false`.

En Groovy:

> *The* `==` *operator doesn't always exactly match the* `.equals()` *method. You can think of them as equivalent in most situations. In situations where two objects might be thought* equal *via normal Groovy* coercion *mechanisms, the* `==` *operator will report them as equal; the* `.equals()` *method will not do so if doing so would break the normal rules Java has around the equals method. Expect further improvements to Groovy over time to provide clearer, more powerful and more consistent behavior in this area.*

Sobrecarga del Operador `in` via `isCase`

Sobreescribir `isCase` altera la conducta del operador `in`. El siguiente programa:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ cat -n isCase1.groovy
class A{
  boolean isCase(Object o){
    if(o == 'A') return true
    else return false
  }
}

def a= new A()

println "a.isCase('A')  = "+a.isCase('A')
```

[2] The central idea of contract is a metaphor on how elements of a software system collaborate with each other, on the basis of mutual obligations and benefits. The metaphor comes from business life, where a çlient.ªnd a ”supplier.ªgree on a çontract”which defines for example that:

- The supplier must provide a certain product (obligation) and is entitled to expect that the client has paid its fee (benefit).
- The client must pay the fee (obligation) and is entitled to get the product (benefit).
- Both parties must satisfy certain obligations, such as laws and regulations, applying to all contracts.

Similarly, if a routine from a class in object-oriented programming provides a certain functionality, it may :

- Expect a certain condition to be guaranteed on entry by any client module that calls it: the routine's precondition — an obligation for the client, and a benefit for the supplier (the routine itself), as it frees it from having to handle cases outside of the precondition.
- Guarantee a certain property on exit: the routine's postcondition — an obligation for the supplier, and obviously a benefit (the main benefit of calling the routine) for the client.
- Maintain a certain property, assumed on entry and guaranteed on exit: the class invariant.

The contract is the formalization of these obligations and benefits. One could summarize design by contract by the ”three questions”that the designer must repeatedly ask:

- What does it expect?
- What does it guarantee?
- What does it maintain?

Many languages have facilities to make assertions like these. However, Design by Contract oriented languages considers these contracts to be so crucial to software correctness that they should be part of the design process. In effect, Design by Contract advocates writing the assertions first.

```
println "('A' in a)     = "+('A' in a)         //more common, shortcut syntax for isCase()

println "(a.isCase('Z') = "+ (a.isCase('Z'))
println "('Z' in a)     = "+ ('Z' in a) //more common, shortcut syntax for isCase()
```

Produce la salida:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ groovy isCase1.groovy
a.isCase('A')  = true
('A' in a)     = true
(a.isCase('Z') = false
('Z' in a)     = false
```

He aqui otro ejemplo:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ cat -n isCase2.groovy
class MyList extends ArrayList {
    boolean isCase(Object val) {
      return val == 66
    }
}

def myList = new MyList()
myList << 55
println (55 in myList) // return false but myList.contains(55) returns true
println (66 in myList) // returns true but myList.contains(55) returns false
```

Sobrecarga del `Switch` via `isCase`

> *The* `switch` *statement inspects an expression and resumes execution from the first matching case-expression, ie, regex matched, list or set or range contained in, class an instance of, or object equal to.*

Veamos un ejemplo:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ cat -n Switch1.groovy
def values= [
    'abc': 'abc',
    'xyz': 'list',
       18: 'range',
       31: BigInteger,
  'dream': 'something beginning with dr',
     1.23: 'none',
]
values.each{
  def result
  switch( it.key ){
    case 'abc': //if switched expression matches case-expression, execute all
                //statements until 'break'
      result= 'abc'
      break
    case [4, 5, 6, 'xyz']:
      result= 'list'
      break
    case 'xyz': //this case is never chosen because 'xyz' is matched by
```

```
        //previous case, then 'break' executed
   result= 'xyz'
   break
  case 12..30:
   result= 'range'
   break
  case Integer:
   result= Integer //because this case doesn't have a 'break', result
           //overwritten by BigInteger in next line
  case BigInteger:
   result= BigInteger
   break
  case ~/dr.*/:
   result= 'something beginning with dr'
   break
  case {it instanceof Integer && it>30}: //use Closure
   result= 'result is > 30'
   break
  default:
   result= 'none'
 }
 println "it = (${it.key}, ${it.value}) => result == $result"
}
```

Este es el resultado de la ejecución:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ groovy Switch1.groovy
it = (abc, abc) => result == abc
it = (xyz, list) => result == list
it = (18, range) => result == range
it = (31, class java.math.BigInteger) => result == class java.math.BigInteger
it = (dream, something beginning with dr) => result == something beginning with dr
it = (1.23, none) => result == none
```

> *When we supply our own values in the case-expression, the* `isCase` *method is invoked to determine whether or not the switch-expression is matched. If there's no* `isCase` *method, the* `equals` *method is used to test for equality:*

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ cat -n isCase3.groovy
class A{
 boolean isCase(Object switchValue){ //'isCase' method used for case-expression
  if(switchValue == 'Hi') return true
  else return false
 }
}
switch( 'Hi' ){
 case new A():
  println "case A"
  break
 default:
  println "A was false"
}

```

```
class B{
  boolean equals(Object switchValue){ //'equals' method used for case-expression
    this.class == switchValue.getClass()
  }
}
switch( new B() ){
  case new B():
    println "case B"
    break
  default:
    println "B was false"
}
```

El resultado de la ejecución del programa anterior es:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ groovy isCase3.groovy
case A
case B
```

> *"because a Java* `switch/case` *does not work like a Groovy* `switch/case`*. In Java a* `case` *can take only* `int` *compatible constants, in Groovy it can take expressions. In Java all cases share a scope, in Groovy each* `case` *has its own scope. In Groovy we call the* `isCase` *method, in Java it has to be a* `number` *we switch with. If we for example use a closure as case, then this might cause side effects. There are cases where we can let them behave the same and usually when using the java version you won't see a difference in Groovy besides the placement and logic of default."*
>
> *So, while in Java the* `default` *can be placed anywhere in the* `switch/case`*, the default in Groovy is used more as an* `else` *than assigning a* `default` *case.*

Ejemplo: Operaciones con Divisas

El siguiente ejemplo muestra implementaciones de los operadores de igualdad `equals` y suma `plus` para una clase `Money`.

- La implementación de `equals` imita el estilo Groovy, evitando que se produzca una `java.lang.NullPointerException`. Recuérdese que `==` denota igualdad entre objetos/valores y no identidad.
- Si el otro objeto no es una `instanceof Money` retornamos `false`:

  ```
  if (!(other instanceof Money))   return false
  ```

  Una alternativa hubiera sido usar *double-dispatch*[3] y hacer que el otro operando nos promocione, delegando en su método `equals`. Quizá la otra clase conoce como compararse con objetos de la clase `Money`.

  ```
  if (!(other instanceof Money))  {
    other.equals(this)
  }
  ```

- También se sobreescribe el método `hashCode` de la clase Object.

  ```
      int hashCode() { [real, imag].hashCode() }
  ```

[3] In software engineering, double dispatch is a mechanism that dispatches a function call to different concrete functions depending on the runtime types of multiple objects involved in the call

Esto forma parte del contrato que Java impone en la función `hashCode` cuando se sobrecarga `equals`:

> `public int hashCode()`
>
> *Returns a hash code*[4] *value for the object. This method is supported for the benefit of hashtables such as those provided by* `java.util.Hashtable`*.*
>
> *The general contract of* `hashCode` *is:*
>
> - *Whenever it is invoked on the same object more than once during an execution of a Java application, the* `hashCode` *method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.*
> - *If two objects are equal according to the* `equals`*(Object) method, then calling the* `hashCode` *method on each of the two objects must produce the same integer result.*
> - *It is not required that if two objects are unequal according to the* `equals`*(Object) method, then calling the* `hashCode` *method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.*
> - *As much as is reasonably practical, the* `hashCode` *method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java programming language.)*

- También escribimos un operador `plus`:

```
Money plus(Money other) {
  if (other == null)               return null
  if (other.currency != currency)
    throw new IllegalArgumentException(
      "cannot add $other.currency to $currency"
    )
  return new Money(amount+other.amount, currency)
}

```

Es necesario asegurarse que los argumentos tienen el tipo apropiado para llevar a cabo la operación. En el ejemplo se ha optado por lanzar una excepción `IllegalArgumentException` cuando el otro objeto `Money` no es de la misma divisa.

No es del todo correcto decir que hay sobrecarga del `plus` en cuanto que no existe tal operador en la superclase Object. Sería mas preciso hablar de implementación del operador. Si que tendría sentido usar el término sobrecarga si añadimos un método para sumar objetos `Money` con objetos `Integer`:

```
Money plus(Integer other) {
  return new Money(amount+other, currency)
}
```

el mecanismo de dispatch de Groovy encontrará la implementación correcta en tiempo de ejecución.

[4] A hash function is any well-defined procedure or mathematical function that converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes

- Por último implementamos el método `toString()`.

```
String toString() {
  "$amount $currency"
}
```

> *When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method* `valueOf()` *defined by* `String`*.*
>
> `valueOf()` *is overloaded for all the simple types and for type Object.*
>
> - *For the simple types,* `valueOf()` *returns a string that contains the human-readable equivalent of the value with which it is called.*
> - *For objects,* `valueOf()` *calls the* `toString()` *method on the object.*
>   *The* `toString()` *method is the means by which you can determine the string representation for objects of classes that you create.*
>
> *Every class implements* `toString()` *because it is defined by Object. However, the default implementation of* `toString()` *is seldom sufficient. For most important classes that you create, you will want to override* `toString()` *and provide your own string representations.*
>
> *By overriding* `toString()` *for classes that you create, you allow the resulting strings to be fully integrated into Java's programming environment. For example, they can be used in* `print()` *and* `println()` *statements and in concatenation expressions.*

Véase También

- Practically Groovy: Smooth operators

2.7. Precedencia de Operadores

> *The precedence hierarchy of the operators, some of which we haven't looked at yet, is, from highest to lowest:*

1. `$(scope escape)`
2. `new ()(parentheses)`
3. `[](subscripting) ()(method call) {}(closable block) [](list/map)`
4. `. ?. *. (dots)`
5. `~ ! $ ()(cast type)`
6. `**(power)`
7. `++(pre/post) --(pre/post) +(unary) -(unary)`
8. `* / %`
9. `+(binary) -(binary)`
10. `<< >> >>> .. ..<`
11. `< <= > >= instanceof in as`
12. `== != <=>`
13. `&`

14. `^`

15. `|`

16. `&&`

17. `||`

18. `?:`

19. `= **= *= /= %= += -= <<= >>= >>>= &= ^= |=`

Aún no hemos estudiado todos los operadores, como es el caso de `*.`. Veamos un ejemplo de uso:

```
generaciondecodigos@nereida:~/Lgroovy/collections$ cat -n spreadDotOperator.groovy
     1  class Language {
     2      String lang
     3      def speak() { "$lang speaks." }
     4  }
     5
     6  // Create a list with 3 objects. Each object has a lang
     7  // property and a speak() method.
     8  def list = [
     9      new Language(lang: 'Groovy'),
    10      new Language(lang: 'Java'),
    11      new Language(lang: 'Scala')
    12  ]
    13
    14  // Use the spread-dot operator to invoke the speak() method.
    15  println list*.speak()
    16  println list.collect{ it.speak() }
    17
    18  // We can also use the spread-dot operator to access
    19  // properties, but we don't need to, because Groovy allows
    20  // direct property access on list members.
    21  println list*.lang
    22  println list.lang
```

Al ejecutarse, este programa produce la siguiente salida:

```
generaciondecodigos@nereida:~/Lgroovy/collections$ groovy spreadDotOperator.groovy
[Groovy speaks., Java speaks., Scala speaks.]
[Groovy speaks., Java speaks., Scala speaks.]
[Groovy, Java, Scala]
[Groovy, Java, Scala]
```

2.8. Ejemplo de Sobrecarga: Números Complejos

La Clase `Complex`

Este ejemplo esta tomado de Rosetta Code (Complex Class in Groovy):

- *final*

```
class Complex {
    final Number real, imag

```

    - *The final keyword is used in several different contexts to define an entity which cannot later be changed.*
    - *A final variable can only be assigned once. This assignment does not grant the variable immutable status. If the variable is a field of a class, it must be assigned in the constructor of its class.*
      Ejemplo:
      ```
      groovy:000> import Complex
      ===> [import Complex]
      groovy:000> a = new Complex(7,9)
      ===> 7 + 9i
      groovy:000> a.real = 4
      ERROR groovy.lang.ReadOnlyPropertyException: Cannot set readonly property: real for class: Complex
              at Complex.setProperty (Complex.groovy)
              at groovysh_evaluate.run (groovysh_evaluate:3)
              ...
      ```
- *static*
  ```
  static final Complex I = [0,1] as Complex
  ```
    - *Static Attribute: Attributes that belong to a class rather than instances of a class*
    - *Implication: They can be used without instantiating an object*
- `Complex(Number real) { this(real, 0) }`
    - *From within a constructor, you can also use the* `this` *keyword to call another constructor in the same class. Doing so is called an explicit constructor invocation.*
    - *If present, the invocation of another constructor must be the first line in the constructor.*
- *En Groovy es posible explicitar coerción de una lista a una clase usando la palabra reservada* `as`*:*
  ```
  static final Complex I = [0,1] as Complex
  ```
  en este caso se llamará al constructor de `Complex` *con argumentos los elementos de la lista:* `[0, 1]`.
  ```
  groovy:000> b = new Complex(-2,-1)
  ===> -2 - i
  groovy:000> a = b
  ===> -2 - i
  groovy:000> a = [7,3] as Complex
  ===> 7 + 3i
  groovy:000> a = [8,5]
  ===> [8, 5]
  groovy:000> a = (Complex) [8,5]
  ===> 8 + 5i
  ```
- *También se sobreescribe el método* `hashCode` *de la clase Object.*
  ```
  int hashCode() { [real, imag].hashCode() }
  ```
  Esto forma parte del contrato que Java impone en la función `hashCode` *cuando se sobrecarga* `equals`*:*

  > `public int hashCode()`
  > *Returns a hash code[5] value for the object. This method is supported for the benefit of hashtables such as those provided by* `java.util.Hashtable`.

[5] A hash function is any well-defined procedure or mathematical function that converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes

The general contract of `hashCode` *is:*

- *Whenever it is invoked on the same object more than once during an execution of a Java application, the* `hashCode` *method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.*
- *If two objects are equal according to the* `equals`*(Object) method, then calling the* `hashCode` *method on each of the two objects must produce the same integer result.*
- *It is not required that if two objects are unequal according to the* `equals`*(Object) method, then calling the* `hashCode` *method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.*
- *As much as is reasonably practical, the* `hashCode` *method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java programming language.)*

- *También sobreescribimos* `toString`*:*

```
    String toString() {
        def realPart = "${real}"
        def imagPart = imag.abs() == 1 ? "i" : "${imag.abs()}i"
        real == 0 && imag == 0 \
                ? "0" \
                : real == 0 \
                        ? (imag > 0 ? '' : "-")  + imagPart \
                        : imag == 0 \
                                ? realPart \
                                : realPart + (imag > 0 ? " + " : " - ")  + imagPart
    }
}
```

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method `valueOf()` *defined by* `String`*.*

`valueOf()` *is overloaded for all the simple types and for type Object.*

- *For the simple types,* `valueOf()` *returns a string that contains the human-readable equivalent of the value with which it is called.*
- *For objects,* `valueOf()` *calls the* `toString()` *method on the object.*
  The `toString()` *method is the means by which you can determine the string representation for objects of classes that you create.*

Every class implements `toString()` *because it is defined by Object. However, the default implementation of* `toString()` *is seldom sufficient. For most important classes that you create, you will want to override* `toString()` *and provide your own string representations.*

By overriding `toString()` *for classes that you create, you allow the resulting strings to be fully integrated into Java's programming environment. For example, they can be used in* `print()` *and* `println()` *statements and in concatenation expressions.*

Programa Cliente

```
generaciondecodigos@nereida:~/src/groovy/overloading$ cat -n useComplex.groovy
#!/usr/bin/env groovy

def tol = 0.000000001  // tolerance: acceptable "wrongness" to account for rounding error

println 'Demo 1: functionality as requested'
def a = [5,3] as Complex
println 'a == ' + a
def b = [0.5,6] as Complex
println 'b == ' + b

println "a + b == (${a}) + (${b}) == " + (a + b)
println "a * b == (${a}) * (${b}) == " + (a * b)
assert a + (-a) == 0
println "-a == -(${a}) == " + (-a)
assert (a * a.recip() - 1).abs() < tol
println "1/a == (${a}).recip() == " + (a.recip())
println()

println 'Demo 2: other functionality not requested, but important for completeness'
println "a - b == (${a}) - (${b}) == " + (a - b)
println "a / b == (${a}) / (${b}) == " + (a / b)
println "a ** b == (${a}) ** (${b}) == " + (a ** b)
println 'a.real == ' + a.real
println 'a.imag == ' + a.imag
println 'a.rho == ' + a.rho
println 'a.theta == ' + a.theta
println 'a*~a == ' + a*~a
println 'sqrt(a*~a) == ' + (a*~a).power(0.5)
println '|a| == ' + a.abs()
println 'a_bar == ' + ~a

def rho = 10
def piOverTheta = 3
def theta = Math.PI / piOverTheta
def fromPolar1 = Complex.fromPolar(rho, theta)          // direct polar-to-cartesian conversion
def fromPolar2 = Complex.exp(Complex.I * theta) * rho     // Euler's equation
println "rho*cos(theta) + rho*i*sin(theta) == ${rho}*cos(pi/${piOverTheta}) +  ${rho}*i*sin(pi/${piOverTheta}) == " + fromPolar1
println "rho * exp(i * theta) == ${rho} * exp(i * pi/${piOverTheta}) == " + fromPolar2
assert (fromPolar1 - fromPolar2).abs() < tol
println()
```

Ejecución

```
generaciondecodigos@nereida:~/src/groovy/overloading$ ./useComplex.groovy | cat -n
Demo 1: functionality as requested
a == 5 + 3i
b == 0.5 + 6i
a + b == (5 + 3i) + (0.5 + 6i) == 5.5 + 9i
a * b == (5 + 3i) * (0.5 + 6i) == -15.5 + 31.5i
-a == -(5 + 3i) == -5 - 3i
1/a == (5 + 3i).recip() == 0.1470588235 - 0.0882352941i
```

```

Demo 2: other functionality not requested, but important for completeness
a - b == (5 + 3i) - (0.5 + 6i) == 4.5 - 3i
a / b == (5 + 3i) / (0.5 + 6i) == 0.56551724145 - 0.78620689665i
a ** b == (5 + 3i) ** (0.5 + 6i) == -0.013750112198456855 - 0.09332524760169053i
a.real == 5
a.imag == 3
a.rho == 5.830951894845301
a.theta == 0.5404195002705842
a*~a == 34
sqrt(a*~a) == 5.830951894845301
|a| == 5.830951894845301
a_bar == 5 - 3i
rho*cos(theta) + rho*i*sin(theta) == 10*cos(pi/3) +  10*i*sin(pi/3) == 5.000000000000001 + 8.660254037844386i
rho * exp(i * theta) == 10 * exp(i * pi/3) == 5.000000000000001 + 8.660254037844386i
```

2.9. Duck Typing

Introducción

> *In computer programming with object-oriented programming languages,* duck typing *is a style of dynamic typing in which an object's current set of methods and properties determines the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface. The name of the concept refers to the duck test, attributed to James Whitcomb Riley, which may be phrased as follows:*
>
> **when I see a bird that `walks` like a duck**
and `swims` like a duck
and `quacks` like a duck,
I call that bird a duck
>
> *In duck typing, one is concerned with just those aspects of an object that are used, rather than with the type of the object itself.*
>
> *. . .*
>
> *In a duck-typed language, the equivalent function would take an object of any type and call that object's* `walk` *and* `quack` *methods. If the object does not have the methods that are called then the function signals a run-time error. It is this action of any object having the correct* `walk` *and* `quack` *methods being accepted by the function that evokes the quotation and hence the name of this form of typing.*
>
> `Duck` *typing is aided by habitually not testing for the type of arguments in method and function bodies, relying on*
>
> - *documentation,*
> - *clear code,*
> - *and testing*
>
> *to ensure correct use. Users of statically typed languages new to dynamically typed languages are usually tempted to add such static (before run-time) type checks, defeating the benefits and flexibility of duck typing, and constraining the language's dynamism.*

Ejemplo:

```
generaciondecodigos@nereida:~/src/groovy/objects$ cat -n duckTyping.groovy
class Dog {
    void speak() { print "guaguau! " }
    void eat()   { println "bones, bone, bon, bo, b" }
```

```
}

class Pig {
    void speak() { print "oink, oink! " }
    void eat() { println "fodder, fodde, fodd, fod, fo, f" }
}

def behave(animal) {
  3.times {
    animal.speak()
    animal.eat()
  }
}

(Fly, Babe) = [ new Dog(), new Pig() ]
behave(Fly)
behave(Babe)
```

Ejecución:

```
generaciondecodigos@nereida:~/src/groovy/objects$ groovy duckTyping.groovy
guaguau! bones, bone, bon, bo, b
guaguau! bones, bone, bon, bo, b
guaguau! bones, bone, bon, bo, b
oink, oink! fodder, fodde, fodd, fod, fo, f
oink, oink! fodder, fodde, fodd, fod, fo, f
oink, oink! fodder, fodde, fodd, fod, fo, f
```

No se comprueba de que clase es el parámetro `animal` pasado a la función `behaves`. Si puede llamar a `speak` y a `eat` todo irá bien. De lo contrario obtendremos un error.

Ejemplo de Polimorfismo con Duck-Typing

```
generaciondecodigos@nereida:~/src/groovy/objects$ cat -n wikipduck.groovy
def calculate(a, b, c) { return (a+b)*c }

example1 = calculate (1, 2, 3)
example2 = calculate ([1, 2, 3], [4, 5, 6], 2)
example3 = calculate ('apples ', 'and oranges, ', 3)

println example1
println example2
println example3

```

> *Thus, duck typing allows polymorphism without inheritance. The only requirement that function calculate needs in its variables is having the* `+` *and the* `*` *methods.*

```
generaciondecodigos@nereida:~/src/groovy/objects$ groovy wikipduck.groovy
9
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
apples and oranges, apples and oranges, apples and oranges,
```

Críticas

> *One issue with duck typing is that it forces the programmer to have a much wider understanding of the code he or she is working with at any given time. In a strongly and statically typed language that uses type hierarchies and parameter type checking, it's much harder to supply an unexpected object type to a class.*
>
> *For instance, in Python, you could easily create a class called* `Wine`*, which expects a class implementing the ”*`press`*.ªttribute. However, a class called* `Trousers` *might also implement the* `press`*() method. With Duck Typing, in order to prevent strange, hard-to-detect errors, the developer needs to be aware of each potential use of the method ”*`press`*”, even when it's conceptually unrelated to what he or she is working on.*

2.10. Práctica: Números Fraccionarios

Escriba una Clase Groovy que implemente los números racionales sobrecargando las operaciones usuales. El lenguaje Ruby provee una clase Rational que puede servirle de inspiración.

2.11. Multiple Dispatching

Introducción

> *Multiple dispatch or multimethods is the feature of some object-oriented programming languages in which a function or method can be dynamically dispatched based on the run time (dynamic) type of more than one of its arguments. This is an extension of single dispatch polymorphism where a method call is dynamically dispatched based on the actual derived type of the object. Multiple dispatch generalizes the dynamic dispatching to work with a combination of two or more objects.*

Java

> *Java-style method overloading isn't multiple dispatch. In multiple dispatch, the 'switch' is done on the dynamic (run-time) type tag of the object – but Java's (and C++'s) overloading only considers the static (compile-time) type of the object reference.*

> *Distinguishing multiple and single dispatch may be made clearer by an example. Imagine a game which has, among its (user-visible) objects, spaceships and asteroids. When two objects collide, the program may need to do different things according to what has just hit what.*

```
generaciondecodigos@nereida:~/Lgroovy/objects/dispatch$ cat -n DoubleDispatchTest.java
class SpaceShip {}
class GiantSpaceShip extends SpaceShip {}

class Asteroid {

  public void collideWith(SpaceShip sp) {
    System.out.println("Asteroid hit a SpaceShip");
  }
  public void collideWith(GiantSpaceShip gsp) {
    System.out.println("Asteroid hit a GiantSpaceShip");
  }
}

```

```
class ExplodingAsteroid extends Asteroid {

  public void collideWith(SpaceShip sp) {
    System.out.println("ExplodingAsteroid hit a SpaceShip");
  }
  public void collideWith(GiantSpaceShip gsp) {
    System.out.println("ExplodingAsteroid hit a GiantSpaceShip");
  }
}

public class DoubleDispatchTest {
  public static void main(String args[]) {
    Asteroid ast = new Asteroid();
    Asteroid ast1 = new ExplodingAsteroid();
    SpaceShip sp = new SpaceShip();
    SpaceShip sp1  = new GiantSpaceShip();
    ast.collideWith(sp);
    ast.collideWith(sp1);
    ast1.collideWith(sp);
    ast1.collideWith(sp1);
  }
}
```

Cuando se ejecuta se produce la siguiente salida:

```
generaciondecodigos@nereida:~/Lgroovy/objects/dispatch$ javac DoubleDispatchTest.java
generaciondecodigos@nereida:~/Lgroovy/objects/dispatch$ time java DoubleDispatchTest
Asteroid hit a SpaceShip
Asteroid hit a SpaceShip
ExplodingAsteroid hit a SpaceShip
ExplodingAsteroid hit a SpaceShip

real    0m0.106s
user    0m0.052s
sys     0m0.012s
```

> *This is due to the fact that, though Java can recognize the runtime type of the Asteroid, it ignores the runtime type of the SpaceShip which is sent as an argument.*

El problema no aparece si declaramos los objetos `sp1` y `ast1` como `GiantSpaceShip` y `ExplodingAsteroid` respectivamente:

```
~/Lgroovy/objects/dispatch$ cat -n DoubleDispatchTest1.java
class SpaceShip {}
class GiantSpaceShip extends SpaceShip {}

class Asteroid {

  public void collideWith(SpaceShip sp) {
    System.out.println("Asteroid hit a SpaceShip");
  }
  public void collideWith(GiantSpaceShip gsp) {
    System.out.println("Asteroid hit a GiantSpaceShip");
  }
}
```

```

class ExplodingAsteroid extends Asteroid {

  public void collideWith(SpaceShip sp) {
    System.out.println("ExplodingAsteroid hit a SpaceShip");
  }
  public void collideWith(GiantSpaceShip gsp) {
    System.out.println("ExplodingAsteroid hit a GiantSpaceShip");
  }
}

public class DoubleDispatchTest1 {
  public static void main(String args[]) {
    Asteroid ast = new Asteroid();
    ExplodingAsteroid ast1 = new ExplodingAsteroid();
    SpaceShip sp = new SpaceShip();
    GiantSpaceShip sp1  = new GiantSpaceShip();
    ast.collideWith(sp);
    ast.collideWith(sp1);
    ast1.collideWith(sp);
    ast1.collideWith(sp1);
  }
}
```

Ahora la salida es la esperada:

```
~/Lgroovy/objects/dispatch$ javac DoubleDispatchTest1.java
~/Lgroovy/objects/dispatch$ java DoubleDispatchTest1
Asteroid hit a SpaceShip
Asteroid hit a GiantSpaceShip
ExplodingAsteroid hit a SpaceShip
ExplodingAsteroid hit a GiantSpaceShip
```

Groovy

En Groovy se consulta el tipo del objeto en tiempo de ejecución. La consecuencia es que aún cuando los objetos hayan sido declarados de la clase mas general, el método invocado es elegido en términos de los tipos actuales de los parámetros.

```
generaciondecodigos@nereida:~/Lgroovy/objects/dispatch$ cat -n DoubleDispatchTest.groovy
class SpaceShip {}
class GiantSpaceShip extends SpaceShip {}

class Asteroid {

  public void collideWith(SpaceShip sp) {
    System.out.println("Asteroid hit a SpaceShip");
  }
  public void collideWith(GiantSpaceShip gsp) {
    System.out.println("Asteroid hit a GiantSpaceShip");
  }
}

class ExplodingAsteroid extends Asteroid {

```

```
  public void collideWith(SpaceShip sp) {
    System.out.println("ExplodingAsteroid hit a SpaceShip");
  }
  public void collideWith(GiantSpaceShip gsp) {
    System.out.println("ExplodingAsteroid hit a GiantSpaceShip");
  }
}

Asteroid ast = new Asteroid();
Asteroid ast1 = new ExplodingAsteroid();
SpaceShip sp = new SpaceShip();
SpaceShip sp1  = new GiantSpaceShip();
ast.collideWith(sp);
ast.collideWith(sp1);
ast1.collideWith(sp);
ast1.collideWith(sp1);
```

Cuando se ejecuta se produce la siguiente salida:

```
$ groovyc DoubleDispatchTest.groovy
$ time java -classpath \
    /usr/local/src/groovy/groovy-1.6.5/target/install/embeddable/groovy-all-1.6.5.jar:. \
    DoubleDispatchTest
Asteroid hit a SpaceShip
Asteroid hit a GiantSpaceShip
ExplodingAsteroid hit a SpaceShip
ExplodingAsteroid hit a GiantSpaceShip

real    0m0.737s
user    0m1.028s
sys     0m0.088s
```

Cuando es Útil Tener Multiple Dispatching

Damian Conway dice al respecto:

> *Typical situations where multiple dispatch is needed include:*
>
> - *Processing events in an graphical user interface, where the correct response to an event depends not just on the graphical object that receives it, but also on the type of event, and the current mode of the interface (i.e. whether it's active or not, what types of events are enabled, and so on).*
> - *Performing image-processing operations between heterogeneous images, such as a blend between two images that may be in different formats. Using multiple dispatch, the common case where the two images are in the same format can be handled by one (optimized) subroutine, and cases where conversions are required can be delegated to a more general (but probably less efficient) method.*
> - *Handling binary operations on different numerical types (integer, rational, arbitrary-precision, etc.) Often the return type of such an operation will depend on the types of both operands: integer + integer gives integer, integer + rational gives rational, arbitrary-precision + rational gives arbitrary-precision) Multiple dispatch lets you supply a separate method for each combination of operands, and then enables the program to automatically find the right one each time.*
> - *Implementing simulations in which a diversity of objects interact. For example, in a physical simulation, the interaction of two objects colliding will depend on the nature*

of both (hard/hard, hard/brittle, soft/hard, brittle/sticky, and so on). Using multiple dispatch, handlers for each type of object-object interaction can be coded separately, and the correct handler selected automatically, based on which types of objects are involved.

Generally speaking, multiple dispatch is needed whenever two or more objects belonging to different class hierarchies need to interact, and you find yourself doing different things depending on which objects are being combined.

Note that multiple dispatch isn't the same thing as overloaded functions (in C++ or Java). In those languages, you can define two or more methods with the same name but different parameter lists, and the compiler works out which one to call based on the nominal types of the arguments you specify. In other words, the compiler analyzes the argument type information, selects the corresponding target method, and hardcodes a call to it. That means that if you then call the overloaded method with a set of arguments belonging to derived classes, you still invoke the method that handles the original base class arguments.

In multiple dispatch, on the other hand, the method is always chosen polymorphically, by examining the actual run-time types of the objects you passed as arguments, not the compile-time types of the pointers or references through which those arguments were passed. Hence, if you pass derived objects as arguments, you get the method that handles derived objects.

2.12. Práctica: Máximo

Escriba una subrutina que calcule el máximo de los elementos de una lista. Aplíquela a listas de números, cadenas, números fraccionarios y números complejos.

Para sobrecargar los operadores de comparación deberá especificar que la clase `Rational` implementa la interface `Comparable` y proveer el método `compareTo`:

```
public class Rational implements Comparable {
   private int num;
   private int dem;

   Rational(int a, b) {
      assert (b != 0);
      if (b < 0) {
        b = -b
        a = -a
      }
      int x = mcd(a, b);
      num = a/x;
      dem = b/x;
   }

   Rational(int a) { num = a; dem = 1 }
   ................................

   int mcd(int a, b) {
      return (b == 0) ? a : mcd(b, a - (b * (int) Math.floor(a/b)));
   }

   ................................

   public int compareTo(Object obj) {
     obj.dem*this.num <=> obj.num*this.dem
```

```
    }

}
```

> *An interface in the Java programming language is an abstract type that is used to specify an interface (in the generic sense of the term) that classes must implement. Interfaces are declared using the* `interface` *keyword, and may only contain method signatures and constant declarations (variable declarations that are declared to be both* `static` *and* `final`*). An interface may never contain method definitions.*
>
> *As interfaces are implicitly* `abstract`*, they cannot be directly instantiated except when instantiated by a class that* `implements` *the said interface. The class must implement all of the methods described in the interface, or be an abstract class.*
>
> *Object references in Java may be specified to be of an interface type; in which case, they must either be* `null`*, or be bound to an object that implements the interface.*
>
> *One benefit of using interfaces is that they simulate multiple inheritance. All classes in Java (other than* `java.lang.Object`*, the root class of the Java type system) must have exactly one base class; multiple inheritance of classes is not allowed.*
>
> *Furthermore, a Java class may implement, and an interface may extend, any number of interfaces; however an interface may not implement an interface.*

2.13. Repaso

- ¿Que opción puedo usar para indicarle al intérprete Groovy que el programa va en la línea de comandos?
- ¿Que hace cada uno de estos comandos de `groovysh`?
  - `help`
  - `(\h)`
  - `?`
  - `(\?)`
  - `exit`
  - `(\x)`
  - `quit`
  - `(\q)`
  - `import`
  - `(\i)`
  - `display`
  - `(\d)`
  - `clear`
  - `(\c)`
  - `show`
  - `(\S)`
  - `inspect`
  - `(\n)`
  - `purge`
  - `(\p)`
  - `edit`

    - `(\e)`
    - `load`
    - `(\l)`
    - `.`
    - `(\.)`
    - `save`
    - `(\s)`
    - `record`
    - `(\r)`
    - `history`
    - `(\H)`
    - `alias`
    - `(\a)`
    - `set`
    - `(\=)`

- ¿Que funcionalidades provee la clase GroovyShell?
- ¿que hace el siguiente fragmento de código?

```
generaciondecodigos@nereida:~/src/groovy/separatedcomp/groovyscriptsfromgroovy$ cat -n Script2.groovy
println "In Script2"
shell = new GroovyShell()
shell.evaluate(new  File('Script1.groovy'))
```

- ¿Que retorna `shell.evaluate`?
- ¿Que diferencias hay entre los métodos `run` y `evaluate`?
- ¿Que hace una llamada al método `parse`?
- ¿Que representa un objeto de la clase Binding?
- ¿Que hace una llamada al método `binding.getVariable`?
- ¿Que hace una llamada al método `binding.setVariable`?
- Explique el siguiente programa:

```
~/Lgroovy/separatedcomp/groovyscriptsfromgroovy$ cat -n calc.groovy
def stdin = new BufferedReader(new InputStreamReader(System.in))
Binding binding = new Binding()
def shell = new GroovyShell(binding)

int c = 1;
print "> "
stdin.eachLine{ input ->
  def script = shell.parse(input)
  def a = script.run()
  println a
  print "> "
}
```

Indique cual es la salida para cada una de las siguientes entradas:

  - `a =2*3`
  - `a+2`
  - `b = a-3`
  - `Math.sqrt(b)`
  - `^D`

- ¿Que es un tipo *wrapper*?
- ¿Que es *boxing*?
- ¿Que es *unboxing*?
- ¿Que es *autoboxing*?
- ¿Qué es una clausura?
- Explique el siguiente código:

```
a = (1..10).toList()
b = a.collect { 2*it }
r = (0..9).collect { a[it]+b[it] }
println r
```

  - ¿Que significa `it`?
  - ¿Que hace el método `collect`?
  - ¿Que clase tiene `1..10`?

- Dado el código Java:

```
generaciondecodigos@nereida:~/src/groovy/simpledatatypes$ cat -n Frequency.java
import java.util.*;

// Prints a frequency table of the words on the command line
public class Frequency {
   public static void main(String[] args) {
      Map<String, Integer> m = new TreeMap<String, Integer>();
      for (String word : args) {
          Integer freq = m.get(word);
          m.put(word, (freq == null ? 1 : freq + 1));
      }
      System.out.println(m);
   }
}
```

  - ¿Que quedará en `args`?
  - ¿Que significa `public`?
  - ¿Que significa `static`?
  - ¿De que clase es el objeto `m`?
  - ¿En que lugares está ocurriendo autoboxing?
  - ¿Cuál es la semántica de un bucle `for` como el del ejemplo?

- ¿Cuál es el comando para compilar un programa Java?
- ¿Cuál es el comando para interpretar el byte code generado por el compilador de Java?
- ¿Que queda en `b`?

```
Vector<String> x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
boolean b = x.getClass() == y.getClass();
```

- ¿Cuales seran las salidas en la siguiente sesión `groovysh`?

```
groovy:000> String a = 1; a = 4; a = a+1

groovy:000> String a = 1; a = 4; a = a*2

groovy:000> String a = 1; a = 4; a = a*3

groovy:000> String a = 2; b = a.toInteger(); b = b*2
```

- ¿Cuales seran las salidas en la siguiente sesión `groovysh`?

```
groovy:000> b = "432"

groovy:000> b.class

groovy:000> c = b as Integer

groovy:000> c.class

groovy:000> d = "432b"

groovy:000> e = d as Integer
```

- ¿Cuales seran las salidas en la siguiente sesión `groovysh`?

```
groovy:000> a = 1+1.0

groovy:000> a.class
```

- ¿Cuales seran las salidas en la siguiente sesión `groovysh`?

```
groovy:000> b = 1 as BigDecimal

groovy:000> b.class
```

- ¿Cuales seran las salidas en la siguiente sesión `groovysh`?

```
groovy:000> d = 65

groovy:000> d.class
```

```
groovy:000> c = d as String

groovy:000> c.class

groovy:000> c

groovy:000> c[1]

groovy:000> e = 8.5

groovy:000> f = e as String

groovy:000> f.class

groovy:000> f[1]

groovy:000> f[2]
```

- Para cada uno de estos operadores diga como se llama el método que hay que escribir para sobrecargarlo:

```
a + b
a - b
a * b
a ** b
a / b
a % b
a | b
a & b
a ^ b
a++ or ++a
a-- or --a
a[b]
a[b] = c
a << b
a >> b
switch(a) { case(b) : }
~a
-a
+a
```

- ¿Cuales seran las salidas en la siguiente sesión `groovysh`?

```
groovy:000> a = null

groovy:000> b = "foo"

groovy:000> a != b

groovy:000> b != a
```

```
groovy:000> a == null
```

- Indique la salida del siguiente programa:

```
~/src/groovy/overloading$ cat -n Numberscohercioncomparison.groovy
#!/usr/bin/env groovy
Byte a = 12
Double b = 256

println  "a($a) instanceof Byte: ${a instanceof Byte}"
println  "b($b) instanceof Double: ${b instanceof Double}"

if (b > a) {
  println "(b > a): Cohercion working. $b is greater than $a"
  println  "($b > $a) instanceof Boolean: ${(b > a) instanceof Boolean}"
}

if (a > (Byte) b) {
  println "Byte $a is greater than (byte) Double $b"
  println  "($a > (Byte) $b) instanceof Boolean: ${(a > (Byte) b) instanceof Boolean}"
}
```

- Indique la salida del siguiente programa:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ cat -n isCase1.groovy
class A{
  boolean isCase(Object o){
    if(o == ’A’) return true
    else return false
  }
}

def a= new A()

println "a.isCase(’A’)  = "+a.isCase(’A’)
println "(’A’ in a)     = "+(’A’ in a)

println "(a.isCase(’Z’) = "+ (a.isCase(’Z’))
println "(’Z’ in a)     = "+ (’Z’ in a)
```

- Indique la salida del siguiente programa:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ cat -n isCase2.groovy
class MyList extends ArrayList {
    boolean isCase(Object val) {
      return val == 66
    }
}

def myList = new MyList()
myList << 55
println (55 in myList)
println (66 in myList)
```

- Indique la salida del siguiente programa:

```
generaciondecodigos@nereida:~/Lgroovy/overloading$ cat -n Switch1.groovy
def values= [
    ’abc’: ’abc’,
    ’xyz’: ’list’,
       18: ’range’,
       31: BigInteger,
  ’dream’: ’something beginning with dr’,
     1.23: ’none’,
]
values.each{
  def result
  switch( it.key ){
    case ’abc’: //if switched expression matches case-expression, execute all
                //statements until ’break’
      result= ’abc’
      break
    case [4, 5, 6, ’xyz’]:
      result= ’list’
      break
    case ’xyz’: //this case is never chosen because ’xyz’ is matched by
                //previous case, then ’break’ executed
      result= ’xyz’
      break
    case 12..30:
      result= ’range’
      break
    case Integer:
      result= Integer //because this case doesn’t have a ’break’, result
                      //overwritten by BigInteger in next line
    case BigInteger:
      result= BigInteger
      break
    case ~/dr.*/:
      result= ’something beginning with dr’
      break
    case {it instanceof Integer && it>30}: //use Closure
      result= ’result is > 30’
      break
    default:
      result= ’none’
  }
  println "it = (${it.key}, ${it.value}) => result == $result"
}
```

- ¿Que ocurre en un `switch` si la clase de la expresión de un `case` no dispone de un método `isCase`?

Capítulo 3

Entrada/Salida

3.1. Lectura

La clase `File` provee un conjunto de métodos para la entrada/salida.

> *Groovy provides a number of helper methods for working with I/O. All of these work with standard Java Reader*[1]*/Writer*[2] *and InputStream*[3]*/OutputStream*[4] *and File and URL classes.*

Lectura de un fichero

```
generaciondecodigos@nereida:~/src/groovy/files$ cat -n readfile.groovy
     1  #!/usr/bin/env groovy
     2  /**
     3  Reading a text file
     4  */
     5  file = args[0]
     6  new File(file).eachLine{ line ->
     7    println line
     8  }
```

> *If for whatever reason the* `println()` *method were to throw an exception, the* `eachLine()` *method ensures that the file resource is correctly closed. Similarly if an exception occurs while reading, the resource will be closed too.*

Ejecución:

```
generaciondecodigos@nereida:~/src/groovy/files$ ./readfile.groovy readfile.groovy
```

[1] Abstract class for reading character streams. The only methods that a subclass must implement are `read(char[], int, int)` and `close()`.
`read(char[], int, int)` read characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached. The meaning of the parameters is: Destination buffer, Offset at which to start storing characters, Maximum number of characters to read. It returns the number of characters read, or -1 if the end of the stream has been reached. It throwsi an `IOException` if an I/O error occurs.
Most subclasses, however, will override some of the methods defined in order to provide higher efficiency, additional functionality, or both.

[2] Abstract class for writing to character streams. The only methods that a subclass must implement are `write(char[], int, int)`, `flush()`, and `close()`. Most subclasses, however, will override some of the methods defined in order to provide higher efficiency, additional functionality, or both.

[3] This abstract class is the superclass of all classes representing an input stream of bytes. Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.

[4] This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink. Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.

```
#!/usr/bin/env groovy
/**
Reading a text file
*/
file = args[0]
new File(file).eachLine{ line ->
  println line
}
```

Lectura de un Fichero en una sola Operación

El acceso `new File(fileName).text` se traduce en una llamada al método `getText()`

```
public String getText()
```

El cual lee los contenidos del `File` y retorna la `String`.

```
generaciondecodigos@nereida:~/src/groovy/io$ cat -n slurpFile.groovy
#!/usr/bin/env groovy
// retrieve the content of the file
if (!args) return
fileName = args[0]
content = new File(fileName).text
println content
```

Existe otra forma de llamar a `getText`:

```
  String  getText(String)
```

que lee los contenidos del `File` usando la codificación especificada como argumento. En ese caso no es posible acceder via la sintáxis de atributo (`.text`):

```
generaciondecodigos@nereida:~/src/groovy/io$ cat -n slurpFile2.groovy
#!/usr/bin/env groovy
// you can specify the encoding of the file
// note that since getText() has a parameter,
// you cannot call it with something like text("UTF-8")
if (!args) return
fileName = args[0]
content = new File(fileName).getText("UTF-8")

println content
```

Lectura desde `STDIN`

> *An* `InputStreamReader` *is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.*
>
> *Each invocation of one of an* `InputStreamReader`*'s* `read()` *methods may cause one or more bytes to be read from the underlying byte-input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation.*
>
> *For top efficiency, consider wrapping an* `InputStreamReader` *within a* `BufferedReader`*.*

```
~/Lgroovy/files$ cat -n stdin.groovy
#!/usr/bin/env groovy
prompt = '\n> '
print 'Enter text including a digit:' + prompt
new BufferedReader(new InputStreamReader(System.in)).eachLine{ line ->
                                              // line is read from System.in
    if (line =~ '\\d') println "Read: $line"   // normal output to System.out
    else System.err.println 'No digit found.'  // this message to System.err
}
```

El atributo `in` de la clase `System` se refiere al flujo de entrada estandard. Es un objeto de la clase `java.io.BufferedInputStream`. Del mismo modo existen los atributos `out` y `err` para referirse a la salida estandard y al flujo e errores.

> *The* `eachLine` *method iterates through the stream, passing each line to the given closure.*

Ejecución:

```
~/Lgroovy/files$ ./stdin.groovy
Enter text including a digit:
> a3b
Read: a3b
hjk
No digit found.
42
Read: 42
^D
```

En realidad `System.in` pertenece a la clase `java.io.BufferedInputStream` y por tanto es ya un flujo dotado de un buffer. El anterior programa puede reescribirse:

```
~/Lgroovy/files$ cat -n readline.groovy
#!/usr/bin/env groovy
prompt = '\n> '
print 'Enter text including a digit:' + prompt
System.in.eachLine{ line ->
                                              // line is read from System.in
    if (line =~ '\\d') println "Read: $line"   // normal output to System.out
    else System.err.println 'No digit found.'  // this message to System.err
}
```

Matching y Verdad

El operador `a =~ 'regexp'` comprueba si la cadena `a` casa/matches con `regexp`.

```
groovy:000> line = "hello world"
===> hello world
groovy:000>  m = line =~ '[aeiou]'
===> java.util.regex.Matcher[pattern=[aeiou] region=0,11 lastmatch=]
groovy:000> m[0]
===> e
groovy:000> m[1]
===> o
groovy:000> m.size()
===> 3
```

en un contexto booleano es evaluado como cierto si la cadena `a` pertenece al lenguaje descrito por la `regexp`.

> *The following table summarizes the rules and the order in which they are applied to evaluate truth in a condition:*
>
> | Type | *Evaluation criteria* |
> |---|---|
> | `Boolean` | *Value is equal to* `true` |
> | `Matcher` | *There is at least one match* |
> | `Collection` | *The collection is not empty* |
> | `Map` | *The map is not empty* |
> | `String, GString` | *The string is not empty* |
> | `Number, Character` | *The value is not 0 (zero)* |
> | `others` | *Non* `null` *reference* |
>
> *Remember that the type is determined in runtime, not at compile time.*

Lectura y Filtrado de un Fichero

El siguiente programa muestra la primera ocurrencia de un enlace `href` en cada línea de un fichero HTML:

```
~/Lgroovy/files$ cat -n links.groovy
     1  #!/usr/bin/env groovy
     2  if (args.length == 0 ) return
     3  testfile = new File(args[0])
     4  testfile.eachLine { line ->
     5     line.find(/(?ix) href \s* = \s* "( [^"]* )"/) {
     6        delta, link -> println "delta = '$delta' link='$link'"
     7     }
     8  }
```

> *The* `String` *method* `public String find(Pattern pattern, Closure closure)` *finds the first occurrence of a compiled regular expression* `Pattern` *within a* `String`.
>
> *If the pattern doesn't match, the* `closure` *will not be called and* `find` *will return* `null`.
>
> *If it does match and we don't have any capture groups in our regex, there is a single parameter on the* `closure` *that the match gets passed to.*
>
> *If we have capture groups in our expression, our* `closure` *has one parameter for the match, followed by one for each of the capture groups.*
>
> *The funtion returns a* `String` *containing the result of the* `closure`, *or* `null` *if the regex* `pattern` *doesn't match*

Cuando lo ejecutamos obtenemos una salida similar a esta:

```
~/Lgroovy/files$ ./links.groovy node17.html | head -4
delta = 'HREF  =   "mystyle.css"' link='mystyle.css'
delta = 'HREF="node18.html"' link='node18.html'
delta = 'HREF="node16.html"' link='node16.html'
delta = 'HREF="node6.html"' link='node6.html'
```

Lectura de un Fichero en una Lista

El método `readLines` nos permite leer un fichero completo y guardarlo en una lista:

```
groovy:000> f = new File("input.txt")
===> input.txt
groovy:000> lines = f.readLines()
===> [an, input, file]
groovy:000> lines.class
===> class java.util.ArrayList
```

Usando `readLines` podemos contar las líneas en un fichero:

```
groovy:000> new File("input.txt").readLines().size()
===> 3
```

Una vez que los contenidos del fichero se tienen en forma de lista es posible aplicar cualquiera de los métodos de manipulación de listas.

```
groovy:000> new File('input.txt').readLines().reverse()
===> [file, input, an]
groovy:000> new File('input.txt').readLines().sort()
===> [an, file, input]
```

Analizando un Fichero de Texto

En el siguiente ejemplo contaremos el número de párrafos, líneas y apariciones de la palabra `captain` en un fichero de texto:

```
groovy:000> f = new File("texto")
===> texto
groovy:000> f.text
===> Captain my Captain! our fearful trip is done;
The ship has weathered every rack, the prize we sought is won;
The port is near, the bells I hear, the people all exulting,
While follow eyes the steady keel, the vessel grim and daring:

    But O heart! heart! heart!
    O the bleeding drops of red,
    Where on the deck my Captain lies,
    Fallen cold and dead.

O Captain! my Captain! rise up and hear the bells;
Rise up|for you the flag is flung|for you the bugle trills;
For you bouquets and ribbon'd wreaths|for you the shores a-crowding;
For you they call, the swaying mass, their eager faces turning;

    Here Captain! dear father!
    This arm beneath your head;
    It is some dream that on the deck,

        You've fallen cold and dead.

My Captain does not answer, his lips are pale and still;
My father does not feel my arm, he has no pulse nor will;
The ship is anchored safe and sound, its voyage closed and done;
From fearful trip, the victor ship, comes in with object won;

    Exult, O shores, and ring, O bells!
    But I, with mournful tread,
    Walk the deck my Captain lies,

        Fallen cold and dead.
```

Un párrafo se define como texto delimitado entre líneas en blanco:

```
groovy:000> m = f.text.findAll(’\\n\\n+’).size()
===> 8
```

El número de líneas es el número de retornos de carro:

```
groovy:000> m = f.text.findAll(’\\n’).size()
===> 35
```

Contemos el número de veces que aparece la palabra `Captain`:

```
m = f.text.findAll(’(?i)\\bCaptain\\b’).size()
===> 8
```

3.2. Repaso

- ¿Cual es la salida de este programa Java? ¿Porqué? ¿Cual es la salida del correspondiente programa en Groovy?

```java
class SpaceShip {}
class GiantSpaceShip extends SpaceShip {}

class Asteroid {

  public void collideWith(SpaceShip sp) {
    System.out.println("Asteroid hit a SpaceShip");
  }
  public void collideWith(GiantSpaceShip gsp) {
    System.out.println("Asteroid hit a GiantSpaceShip");
  }
}

class ExplodingAsteroid extends Asteroid {

  public void collideWith(SpaceShip sp) {
    System.out.println("ExplodingAsteroid hit a SpaceShip");
  }
  public void collideWith(GiantSpaceShip gsp) {
    System.out.println("ExplodingAsteroid hit a GiantSpaceShip");
  }
}

public class DoubleDispatchTest {
  public static void main(String args[]) {
    Asteroid ast = new Asteroid();
    Asteroid ast1 = new ExplodingAsteroid();
    SpaceShip sp = new SpaceShip();
    SpaceShip sp1  = new GiantSpaceShip();
    ast.collideWith(sp);
    ast.collideWith(sp1);
    ast1.collideWith(sp);
    ast1.collideWith(sp1);
  }
}
```

- ¿Qué hace el método `eachline` de un objeto `File`? ¿Que argumentos recibe?
- ¿Qué hace el método `getText` de un objeto `File`?
- ¿Es lo mismo un byte que un carácter? ¿Cuál es la diferencia?
- Dado el código:

```
groovy:000> line = "hello world"
===> hello world
groovy:000>  m = line =~ '[aeiou]'
===> java.util.regex.Matcher[pattern=[aeiou] region=0,11 lastmatch=]
```

  ¿Que retorna `m[0]`? ¿y `m[1]`? ¿y `m[2]`?

- ¿Cómo se evalúa `line =~ '[aeiou]'` en un contexto booleano?
- ¿Que es un grupo de captura o paréntesis de captura en una expresión regular?
- ¿Que hace el método `find` de la clase `String`? ¿Que argumentos recibe?
- En la llamada:

```
line.find(/(?ix) href \s* = \s* "( [^"]* )"/) { ... }
```

  - ¿Que argumentos se le pasan a la clausura?
  - ¿Que significa `(?ix)`?
  - ¿Que significa `\s`?
  - ¿Que significa `[^"]`?
  - ¿Que retorna `find`?

3.3. Escritura

El Método `write`

Los objetos `File` tienen dos formas del método `write`:

```
void    write(String)
void    write(String, String)
```

ambas escriben su primer argumento al fichero. El segundo argumento especifica la codificación a usar.

```
generaciondecodigos@nereida:~/src/groovy/files$ cat -n writeFile.groovy
     1  #!/usr/bin/env groovy
     2  // write the text in the file
     3  new File("myFile.txt").write("Hello world")
```

El Método `write` y la Codificación

Veamos otro ejemplo:

```
Hello worldgeneraciondecodigos@nereida:~/src/groovy/files$ cat -n writeEuro.groovy
     1  #!/usr/bin/env groovy
     2  // you may specify the encoding
     3  new File("myFile.txt").write("\u20AC is the symbol for the Euro currency", 'UTF-8')
```

Sobreescritura

Sucesivas llamadas al método `write` sobreescriben el fichero:

```
groovy:000> f = new File("fileWrite.txt")
===> fileWrite.txt
groovy:000> f.write("Hello 1\n")
===> null
groovy:000> "cat fileWrite.txt".execute().text
===> Hello 1

groovy:000> f.write("Hello 2\n")
===> null
groovy:000> "cat fileWrite.txt".execute().text
===> Hello 2
```

El método `execute()` arranca un proceso que ejecuta la cadena como comando. Retorna un objeto `java.lang.UNIXProcess` que tiene atributos `errorStream`, `inputStream` y `outputStream`.

El Método `append`

Usaremos `append` si se quieren concatenar las salidas:

```
groovy:000> f = new File("Chum.txt")
===> Chum.txt
groovy:000> f.append("Hello World!\n")
===> null
groovy:000> f.append("... and Goodbye!\n")
===> null
groovy:000> "cat -n Chum.txt".execute().text
===>      1     Hello World!
     2  ... and Goodbye!
```

Copia de un Fichero

El siguiente programa copia los contenidos de un fichero en otro:

```
generaciondecodigos@nereida:~/src/groovy/files$ cat -n copy.groovy
     1  #!/usr/bin/env groovy
     2  if (args.length != 2) {
     3    println "Usage:\n\tcopy file.input file.output"
     4    System.exit 1
     5  }
     6
     7  source = new File(args[0])
     8  if (!source.canRead()) {
     9      println "Can't find file $source!"
    10      System.exit 2
    11  }
    12
    13  target = new File(args[1])
    14
    15  try {
    16    target.withWriter { file ->
    17        source.eachLine { line ->
    18            file.writeLine(line)
    19        }
```

```
  }
} catch(e) {
  println "Can't create copy of $source in $target"
  System.exit 3
}
```

Veamos el resultado de algunas ejecuciones:

```
generaciondecodigos@nereida:~/src/groovy/files$ ./copy.groovy input.txt output.txt
generaciondecodigos@nereida:~/src/groovy/files$ diff input.txt output.txt
generaciondecodigos@nereida:~/src/groovy/files$
generaciondecodigos@nereida:~/src/groovy/files$ echo $?
0
generaciondecodigos@nereida:~/src/groovy/files$ ./copy.groovy doesnotexist.txt output.txt
Can't find file doesnotexist.txt!
generaciondecodigos@nereida:~/src/groovy/files$ echo $?
2
generaciondecodigos@nereida:~/src/groovy/files$ ./copy.groovy input.txt /root/output.txt
Can't create copy of input.txt in /root/output.txt
generaciondecodigos@nereida:~/src/groovy/files$ echo $?
3
generaciondecodigos@nereida:~/src/groovy/files$ ./copy.groovy input.txt a b
Usage:
        copy file.input file.output
generaciondecodigos@nereida:~/src/groovy/files$ echo $?
1
```

El Método `withWriter`, la Clase `BufferedWriter` y el Método `writeLine`

El método `withWriter`

```
public Object withWriter(Closure)
```

crea un nuevo objeto `BufferedWriter` para este fichero. Dicho objeto se le pasa a la clausura que constituye el argumento, garantizándose que cuando la clausura termina el buffer ha sido completamente volcado (flushed) y cerrado.

Los objetos `BufferedWriter` han sido concebidos para escribir texto a un flujo de entrada/salida usando buffering de manera que la escritura sea eficiente. Los objetos `BufferedWriter` disponen del método

```
void    writeLine(String)
```

el cual escribe el texto añadiendo al final una nueva línea:

```
  target.withWriter { file ->
      source.eachLine { line ->
          file.writeLine(line)
      }
  }
```

El Método `eachLine` de los Objetos `File`

La clausura que recibe como último argumento `eachLine` puede ser llamada también con dos parámetros. En ese caso el segundo argumento es el número de línea. Podemos modificar entonces las líneas de copia en el anterior ejemplo de esta forma:

```
 target.withWriter { file ->
      source.eachLine { line, num ->
          file.writeLine("$num: $line")
      }
  }
```

y obtendremos un programa que numera las líneas copiadas:

```
generaciondecodigos@nereida:~/src/groovy/files$ ./copywithnum.groovy input.txt output.txt
generaciondecodigos@nereida:~/src/groovy/files$ diff input.txt output.txt
1,3c1,3
< an
< input
< file
---
> 1: an
> 2: input
> 3: file
```

El método `eachLine` retorna el valor retornado por la última ejecución de la clausura.

Es posible pasarle como parámetros la codificación y el número de línea inicial:

```
public Object eachLine(String, int, Closure)
```

Parámetros:

1. *charset - Abre el fichero con el charset indicado*
2. *firstLine - Indica la primera línea (por defecto es 1)*
3. *closure - La clausura recibe dos argumentos. El primero es la línea, el segundo el número de línea*

Copia: Otra forma de hacerla

Esta es una solución alternativa en la que se usa el método `append` del fichero `target` prescindiendo de `withWriter`. En el caso de que el fichero `target` ya exista lo vaciamos:

```
generaciondecodigos@nereida:~/Lgroovy/files$ cat -n copy3.groovy
     1  #!/usr/bin/env groovy
     2  if (args.length < 2) return
     3
     4  source = new File(args[0])
     5  if (source.canRead()) {
     6    target = new File(args[1])
     7    if (target.exists()) {
     8      target.delete()
     9    }
    10    try {
    11          source.eachLine { line ->
    12              target.append("$line\n")
    13          }
    14    }
    15    catch(e) {
    16      println "Can't write in $target!"
    17    }
    18  }
    19  else {
    20    println "File $source not found!"
    21  }
```

3.4. Funciones para la Manipulación de Ficheros

La clase `java.io.File` implementa un buen número de métodos para la manipulación de ficheros.

Análisis del Nombre de un Fichero Existen múltiples métodos para analizar el nombre de un fichero.

> *The method* `getPath` `public String getPath()` *converts this abstract pathname into a pathname string. The resulting string uses the default name-separator character to separate the names in the name sequence.*

```
groovy:000> new File('io/cliargs.groovy').getPath()
===> io/cliargs.groovy
```

> *The method* `getAbsoluteFile` *with signature* `public File getAbsoluteFile()` *returns the absolute form of this abstract pathname.*

He aquí algunos ejemplos de uso de algunas funcionesr relacionadas:

```
groovy:000> new File('io/cliargs.groovy').getAbsoluteFile()
===> /Users/casianorodriguezleon/src/groovy/io/cliargs.groovy
groovy:000> new File('io/cliargs.groovy').getCanonicalPath()
===> /Users/casianorodriguezleon/src/groovy/files/cliargs.groovy
groovy:000> new File('io/cliargs.groovy').getCanonicalFile()
===> /Users/casianorodriguezleon/src/groovy/files/cliargs.groovy
groovy:000> new File('io/cliargs.groovy').toURL()
===> file:/Users/casianorodriguezleon/src/groovy/io/cliargs.groovy
groovy:000> new File('io/cliargs.groovy').getParent()
===> io
groovy:000> new File('io/cliargs.groovy').getParentFile()
===> io
groovy:000> new File('io/cliargs.groovy').getName()
===> cliargs.groovy
```

Estado de un Fichero

La clase `java.io.File` provee un buen número de métodos para consultar el estado de un fichero.

- `public boolean canRead()`

  > *Tests whether the application can read the file denoted by this abstract pathname.*
  > *Returns: true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise*

  ```
  groovy:000> new File('io/cliargs.groovy').canRead()
  ===> true
  ```

- `public boolean canWrite()`

  > *Tests whether the application can modify to the file denoted by this abstract pathname.*
  > *Returns: true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.*

  ```
  groovy:000> new File('io/cliargs.groovy').canWrite()
  ===> true
  ```

- `public boolean exists()`

  Tests whether the file or directory denoted by this abstract pathname exists.

  Returns: true if and only if the file or directory denoted by this abstract pathname exists; false otherwise

- `public boolean isDirectory()`

  Tests whether the file denoted by this abstract pathname is a directory.

  Returns: true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise

```
groovy:000> new File('io/cliargs.groovy').isDirectory()
===> false
groovy:000> new File('io').isDirectory()
===> true
```

- `public boolean isFile()`

  Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file.

  Returns: true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise

- `public boolean isHidden()`

  Tests whether the file named by this abstract pathname is a hidden file. The exact definition of hidden is system-dependent. On UNIX systems, a file is considered to be hidden if its name begins with a period character ('.'). On Microsoft Windows systems, a file is considered to be hidden if it has been marked as such in the filesystem.

  Returns: true if and only if the file denoted by this abstract pathname is hidden according to the conventions of the underlying platform Throws:

```
groovy:000> new File('io/cliargs.groovy').isHidden()
===> false
groovy:000> new File('.svn').isHidden()
===> true
```

- `public boolean isFile()`

  Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file.

  Returns: true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise

```
groovy:000> new File('io/cliargs.groovy').isFile()
===> true
```

Fechas Asociadas con un Fichero

> *The method* `public long lastModified()` *returns the time that the file denoted by this abstract pathname was last modified.*
>
> *Returns: A long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs*

```
groovy:000> a = new File('io/cliargs.groovy').lastModified()
===> 1267353623000
groovy:000> new Date(a)
===> Sun Feb 28 10:40:23 GMT 2010
```

Longitud de un Fichero

> *The method* `public long length()` *returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.*
>
> *Returns: The length, in bytes, of the file denoted by this abstract pathname, or 0L if the file does not exist*

```
groovy:000> new File('io/cliargs.groovy').length()
===> 3318
groovy:000> 'ls -l io/cliargs.groovy'.execute().text
===> -rw-r--r--  1 casianorodriguezleon  staff  3318 28 feb 10:40 io/cliargs.groovy
```

3.5. Procesado de la Línea de Comandos

La clase `CliBuilder`
La clase `CliBuilder` proporciona un conjunto de métodos para el procesado de la línea de comandos.

> *Most of the time we pass arguments to the script we invoke. These arguments are available as a* `String[]` *array in our script. For example to get the first argument we can use the following code* `s = args[0]`*. To have real argument handling in our script we use Groovy's* `CliBuilder` *class. This class uses Jakarta Commons CLI under the hood. With* `CliBuilder` *we can define the argument options and parse the arguments.*

Ejemplo Simple
El siguiente ejemplo muestra la forma de uso:

```
generaciondecodigos@nereida:~/Lgroovy/files$ cat -n cli.groovy
#!/usr/bin/env groovy
//http://groovy.codehaus.org/gapi/groovy/util/CliBuilder.html
import groovy.util.CliBuilder

def cli = new CliBuilder(usage:'cli.groovy')

cli.a(longOpt:'all',  'display all files')
cli.l(longOpt:'long', 'use a long listing format')
cli.t(longOpt:'time', 'sort by modification time')
cli.t(longOpt:'time', 'sort by modification time')
cli.h(longOpt: 'help', 'Show usage information')

def options = cli.parse(args)
```

```

println options

println options.arguments()

println "a = "+options.a
println "l = "+options.l
println "long = "+options.long
println "t = "+options.t
println "h = "+options.h

if (options.h) cli.usage()
```

Cuando lo ejecutamos obtenemos estas salidas:

```
generaciondecodigos@nereida:~/Lgroovy/files$ ./cli.groovy -h
groovy.util.OptionAccessor@396cbd97
[]
a = false
l = false
long = false
t = false
h = true
usage: cli.groovy
 -a,--all    display all files
 -h,--help   Show usage information
 -l,--long   use a long listing format
 -t,--time   sort by modification time
generaciondecodigos@nereida:~/Lgroovy/files$ ./cli.groovy -l s*.groovy
groovy.util.OptionAccessor@121321f5
[slurpFile2.groovy, slurpFile.groovy, socket.groovy, stdin2.groovy, stdin.groovy]
a = false
l = true
long = true
t = false
h = false
generaciondecodigos@nereida:~/Lgroovy/files$ ./cli.groovy --long s*.groovy
groovy.util.OptionAccessor@121321f5
[slurpFile2.groovy, slurpFile.groovy, socket.groovy, stdin2.groovy, stdin.groovy]
a = false
l = true
long = true
t = false
h = false
generaciondecodigos@nereida:~/Lgroovy/files$ ./cli.groovy -lat s*.groovy
groovy.util.OptionAccessor@693e4a5a
[slurpFile2.groovy, slurpFile.groovy, socket.groovy, stdin2.groovy, stdin.groovy]
a = true
l = true
long = true
t = true
h = false
```

Un Ejemplo más Complejo

El siguiente programa copia un fichero en otro filtrando las líneas según un comando especificado en línea de argumentos. Sigue un ejemplo de ejecución:

```
~/Lgroovy/files$ cat sample.txt
1
2
3
4
~/Lgroovy/files$ ./cf -vo t.txt -f 'line.replaceFirst(/1/, "One")' sample.txt
Using filter 'line.replaceFirst(/1/, "One")' on file 'sample.txt' to produce output in 't.txt'
~/Lgroovy/files$ cat t.txt
One
2
3
4
```

El programa admite opciones como `-v` (verbose), `-h` (ayuda), `-o` (salida) y `-f` (filtro).

```
~/Lgroovy/files$ cat -n cf
#!/usr/bin/env groovy
//http://groovy.codehaus.org/gapi/groovy/util/CliBuilder.html
import groovy.util.CliBuilder

def cli = new CliBuilder(usage: 'cf -[hof] source')

// Create the list of options.
cli.with {
    h longOpt: 'help', 'Show usage information'
    v longOpt: 'verbose', 'verbose output'
    o longOpt: 'output', args: 1, argName: 'output', 'Output file'
    f longOpt: 'filter', args: 1, argName: 'filter', 'Groovy Code for line'
}
```

El Método `with`

> *Groovy has a* `with` *method we can use to group method calls and property access to an object. The* `with` *method accepts a closure and every method call or property access in the closure applies to the object if applicable.*

El siguiente ejemplo muestra el uso del método `with`:

```
~/Lgroovy/learning$ cat -n with2.groovy
#!/usr/bin/env groovy
class Sample {
    String username
    String email
    List<String> labels = []
    def speakUp() { "I am $username" }
    def addLabel(value) { labels << value }
}

def sample = new Sample()
sample.with {
    username = 'fulanito'
    email = 'email@host.com'
```

```
    println speakUp()
    addLabel 'Groovy'
    addLabel 'Java'
    addLabel 'Perl'
}

sample.with {
    println labels.size()
    println labels[0]
    println labels[1]
    println username
    println email
}

```

Sigue una ejecución:

```
~/Lgroovy/learning$ ./with2.groovy
I am fulanito
3
Groovy
Java
fulanito
email@host.com
```

Análisis de los Argumentos en Línea de Comandos: `parse`

El método `parse` de La clase `CliBuilder`:

```
OptionAccessor parse(def args)
```

nos retorna un objeto `OptionAccessor` que nos permite acceder a las opciones especificadas en línea de comandos. Retorna un `null` cuando la línea de comandos esta mal formada.

Véase un ejemplo de uso en la `groovyShell`

```
groovy:000> cli = new CliBuilder()
===> groovy.util.CliBuilder@da3b359
groovy:000> cli.v(longOpt: 'verbose', 'verbose mode')
===> null
groovy:000> cli.D(longOpt: 'Debug', 'display debug info')
===> null
groovy:000> cli.o(longOpt: 'output', args:1, 'use/specify output file')
===> null
groovy:000> options = cli.parse(['-v', '-D', '-o mifichero'])
===> groovy.util.OptionAccessor@1952853d
groovy:000> options.v
===> true
groovy:000> options.D
===> true
groovy:000> options.o
===>  mifichero
groovy:000> options.output
===>  mifichero
groovy:000> options.verbose
===> true
```

Los métodos `v`, `D` y `o` del objeto `cli` son creados dinámicamente. Del mismo modo, el objeto `options` tiene atributos `v`, `D` y `o` que han sido creados dinámicamente. Obsérvese que ademas existen alias con los nombres largos para dichos atributos.

Comprobando la Definición de cada Argumento

Asi pues, nuestro programa continua con el análisis de la línea de comandos. Si la opción es `-h` se muestra la ayuda:

```
def options = cli.parse(args)

if (options.h) {
    cli.usage()
    println '''
Examples:
   $ ./cf -o t.txt -f 'line.replaceFirst(/1/, "One")' sample.txt
   $ ./cf -vo t.txt -f 'line.replaceFirst(/1/, "One")' sample.txt
'''
    return 0
}
```

Si no se especifica un filtro se le da un valor por defecto:

```
def filter = (options.f)? options.f : 'println line'
```

A continuación se analizan los restantes argumentos: los argumentos que no están asociados con una opción. En nuestro caso se trata del nombre del fichero fuente:

```
def extraArguments = options.arguments()
if (extraArguments) {
   source = new File(extraArguments[0])
}
else {
   println "Don't forget to specify the source file"
   return -3
}
```

El nombre del fichero de salida sigue a la opción `-o`:

```
def target = options.o ? new File(options.o) : new File('salida.tmp')

if (options.v) println "Using filter '$filter' on file '$source' to produce output in '$target'"
```

Copia de un Fichero en Otro con Filtrado Previo

Nuestro programa copia cada línea del fichero fuente `source` en el fichero `target` aplicándole previamente el código especificado en `filter`. Para ello usamos la clase GroovyShell. Primero compilamos el texto usando `shell.parse` y después ejecutamos el script resultante usando `script.run`:

```
def shell = new GroovyShell(binding)
try {
  script = shell.parse(filter)
}
catch(e) {
  println "ERROR compiling '$filter'"
  System.exit(-2)
}
```

```
target.withWriter { file ->
    source.eachLine {
        line = it
        try {
          result = script.run()
        }
        catch (e) {
          println "ERROR evaluating '$filter'"
          System.exit(-1)
        }
        result == null ? file.writeLine(line) : file.writeLine(result as String)
    }
}
```

Código Completo

```
~/Lgroovy/files$ cat -n cf
#!/usr/bin/env groovy
//http://groovy.codehaus.org/gapi/groovy/util/CliBuilder.html
import groovy.util.CliBuilder

def cli = new CliBuilder(usage: 'cf -[hof] source')

// Create the list of options.
cli.with {
    h longOpt: 'help', 'Show usage information'
    v longOpt: 'verbose', 'verbose output'
    o longOpt: 'output', args: 1, argName: 'output', 'Output file'
    f longOpt: 'filter', args: 1, argName: 'filter', 'Groovy Code for line'
}

def options = cli.parse(args)

if (options.h) {
    cli.usage()
    println '''
Examples:
   $ ./cf -o t.txt -f 'line.replaceFirst(/1/, "One")' sample.txt
   $ ./cf -vo t.txt -f 'line.replaceFirst(/1/, "One")' sample.txt
'''
    return 0
}

def filter = (options.f)? options.f : 'println line'

def extraArguments = options.arguments()
if (extraArguments) {
   source = new File(extraArguments[0])
}
else {
   println "Don't forget to specify the source file"
   return -3
}

```

```
def target = options.o ? new File(options.o) : new File('salida.tmp')

if (options.v) println "Using filter '$filter' on file '$source' to produce output in '$target'"

def shell = new GroovyShell(binding)
try {
  script = shell.parse(filter)
}
catch(e) {
  println "ERROR compiling '$filter'"
  System.exit(-2)
}

target.withWriter { file ->
    source.eachLine {
        line = it
        try {
          result = script.run()
        }
        catch (e) {
          println "ERROR evaluating '$filter'"
          System.exit(-1)
        }
        result == null ? file.writeLine(line) : file.writeLine(result as String)
    }
}
```

3.6. Práctica: Procesando la línea de Comandos

Escriba un programa similar a `cat`. Recibe como argumento un nombre de fichero e imprime las líneas entre los números indicados por sus argumentos `-b n` y `-e m`. La opción `-h` proporcionará la ayuda. La opción `-n` mostrará las líneas enumeradas.

3.7. Groovy One liners

```
~$ groovy -h
usage: groovy [options] [args]
options:
  -a,--autosplit <splitPattern>    split lines using splitPattern (default '\s')
                                   using implicit 'split' variable
  -c,--encoding <charset>          specify the encoding of the files
  -D,--define <name=value>         define a system property
  -d,--debug                       debug mode will print out full stack traces
  -e <script>                      specify a command line script
  -h,--help                        usage information
  -i <extension>                   modify files in place; create backup if
                                   extension is given (e.g. '.bak')
  -l <port>                        listen on a port and process inbound lines
  -n                               process files line by line using implicit
                                   'line' variable
  -p                               process files line by line and print result
                                   (see also -n)
  -v,--version                     display the Groovy and JVM versions
```

Because looping through `STDIN` *or input files tends to be a common thing to do, groovy (and ruby, perl etc) provide shortcuts for this.*

- `-n` *will loop through each line of the input, and provide it to your script in the line variable.*

```
~/LGCbook$ grep -i ^groov /usr/share/dict/words | groovy -n -e 'println line.toUpperCase()'
GROOVE
GROOVELESS
GROOVELIKE
GROOVER
GROOVERHEAD
GROOVINESS
GROOVING
GROOVY
```

- *If we definitely want to print the output of each line we can use* `-p` *and shorten it to*

```
~/LGCbook$ grep -i ^groov /usr/share/dict/words | groovy -p -e 'line.toUpperCase()'
GROOVE
GROOVELESS
GROOVELIKE
GROOVER
GROOVERHEAD
GROOVINESS
GROOVING
GROOVY
```

- *We can use the looping constructs along with* `-i`*, which writes the output back to the original files (and creates a backup copy with the given extension). And wreak havoc on our local file system, with wide-scale search and replace.*

```
~/LGCbook$ grep hola  gcexamples/*.html
gcexamples/node65.html:    14  ph = p("hola ")
gcexamples/node65.html:hola mundo
gcexamples/node79.html:groovy:000&gt; SomeClass.classField = 'hola'
gcexamples/node79.html:===&gt; hola
gcexamples/node79.html:groovy:000&gt; a.typedField = 'hola'
gcexamples/node79.html:===&gt; hola
gcexamples/node80.html:  String attr = 'hola'
gcexamples/node80.html:hola
~/LGCbook$ groovy -p -i .bak -e '(line =~ "hola").replaceAll("hola CocaCola")' gcexamples/*.html
~/LGCbook$ grep hola  gcexamples/*.html
gcexamples/node65.html:    14  ph = p("hola CocaCola ")
gcexamples/node65.html:hola CocaCola mundo
gcexamples/node79.html:groovy:000&gt; SomeClass.classField = 'hola CocaCola'
gcexamples/node79.html:===&gt; hola CocaCola
gcexamples/node79.html:groovy:000&gt; a.typedField = 'hola CocaCola'
gcexamples/node79.html:===&gt; hola CocaCola
gcexamples/node80.html:  String attr = 'hola CocaCola'
gcexamples/node80.html:hola CocaCola
```

  TIP: Never ever use the option `-i` *without a backup extension.*

- *Or to really get into groovy (literally)*

```
~/Lgroovy/files$ find . -name 'r*.groovy'
./readfile.groovy
./readline.groovy
~/Lgroovy/files$ find . -name 'r*.groovy' | xargs cat -n
     1  #!/usr/bin/env groovy
     2  // Reading a text file
     3  file = args[0]
     4  new File(file).eachLine{ line ->
     5    println line
     6  }
     1  #!/usr/bin/env groovy
     2  prompt = '\n> '
     3  print 'Enter text including a digit:' + prompt
     4  System.in.eachLine{ line ->
     5                                              // line is read from System.in
     6      if (line =~ '\\d') println "Read: $line"   // normal output to System.out
     7      else System.err.println 'No digit found.'  // this message to System.err
     8  }
~/Lgroovy/files$ find . -name 'r*.groovy' | xargs groovy -i .bak -pe '(line =~ "//.*").replaceAll("")'
~/Lgroovy/files$ ls -l r*.groovy.bak
-rw-r--r--  1 casianorodriguezleon  staff  111 15 abr 06:17 readfile.groovy.bak
-rw-r--r--  1 casianorodriguezleon  staff  348 15 abr 06:16 readline.groovy.bak
~/Lgroovy/files$ diff ./readline.groovy ./readline.groovy.bak
5,7c5,7
<
<     if (line =~ '\\d') println "Read: $line"
<     else System.err.println 'No digit found.'
---
>                                             // line is read from System.in
>     if (line =~ '\\d') println "Read: $line"   // normal output to System.out
>     else System.err.println 'No digit found.'  // this message to System.err
```

- *Additionally you have access to the line number in the current file you are reading via the variable* `count`*. This can be used for a number of convenient groovy one-liners.*
  Let us assume you want to prefix every line in a file with the line number. Doing this requires next to no work in Groovy (we additionally create a copy of the original file with the extension `.bak`*).*

```
~/Lgroovy/files$ cat prueba
one
two
three
four
~/Lgroovy/files$ groovy -pi .bak -e "count + ': ' + line" prueba
~/Lgroovy/files$ cat prueba
1: one
2: two
3: three
4: four
~/Lgroovy/files$ cat prueba.bak
one
two
three
four
```

- *Or let us create a grep-like command that prints the line number where it found matching strings for a regular expression.*

```
~/Lgroovy/files$ groovy -p -e "if (line =~ /groovy/) count + ': ' + line" *.groovy
4:     def cli = new CliBuilder(usage: 'showdate.groovy -[chflms] [date] [prefix]')
24:         // usage: showdate.groovy -[chflms] [date] [prefix]
1: #!/usr/bin/env groovy
2: //http://groovy.codehaus.org/gapi/groovy/util/CliBuilder.html
3: import groovy.util.CliBuilder
1: #!/usr/bin/env groovy
1: #!/usr/bin/env groovy
1: #!/usr/bin/env groovy
1: #!/usr/bin/env groovy
1: #!/usr/bin/env groovy
1: #!/usr/bin/env groovy
```

- *Print the first 3 lines of all files:*

```
~/Lgroovy/files$ groovy -pe "if (count <=3) count+':'+line" tail.groovy ls.groovy
1:#!/usr/bin/env groovy
2:import java.lang.management.*
3:
1:#!/usr/bin/env groovy
2:if (args.length > 1) {
3:  regexp = args[0]
```

- *Another very convenient option is* `-a`*, which splits the current input line into the array* `split`*. By default the split pattern is* `" "` *(one space).*
  Print processes owned by root:

```
lhp@nereida:~$ ps aux | grep lhp
root     16086  0.0  0.0  78748  3372 ?        S<s  05:39   0:00 sshd: lhp [priv]
lhp      16094  0.0  0.0  78852  1944 ?        S<   05:39   0:00 sshd: lhp@pts/15
lhp      16170  0.0  0.1  22364  5192 pts/15   S<s  05:39   0:00 -bash
root     16337  0.0  0.0  78748  3372 ?        S<s  05:39   0:00 sshd: lhp [priv]
lhp      16346  0.0  0.0  78748  1868 ?        S<   05:39   0:00 sshd: lhp@pts/16
lhp      16425  0.0  0.1  22364  5156 pts/16   S<s  05:39   0:00 -bash
lhp      22082  0.0  0.0  10444  1928 pts/15   S<+  06:31   0:00 man paste
lhp      22092  0.0  0.0   9672  1032 pts/15   S<+  06:31   0:00 pager -s
lhp      26079  0.0  0.0  15172  1136 pts/16   R<+  07:13   0:00 ps aux
lhp      26080  0.0  0.0   7544   924 pts/16   S<+  07:13   0:00 grep lhp
lhp@nereida:~$ ps aux | groovy -a -ne "if (split[0] =~ 'lhp') println split[-1]"
lhp@pts/15
-bash
lhp@pts/16
-bash
paste
-s
aux
split[-1]
```

- *The option* `-a` *optionally takes another split pattern which is then used instead.*
  Print all logins from `/etc/passwd` *that are not commented:*

```
lhp@nereida:~$ groovy -a':' -ne 'if (split[0] =~ /^lh/) println split[0]' /etc/passwd
lhp
```

- *Add the first and the penultimate column of a file:*

  ```
  ~/Lgroovy/files$ cat accounts.txt
  1 2 3 4 5
  1 2 3 4 5
  1 2 3 4 5
  1 2 3 4 5
  ~/Lgroovy/files$ groovy -a -pe "split[0].toInteger()+split[-2].toInteger()" accounts.txt
  5
  5
  5
  5
  ```

- *For more examples or inspiration browse through the search results for Perl One Liners listen mode*
- *Another groovy command line option is the ability to startup groovy in listen mode, which will attach groovy to a TCP port on your machine (*`-l <port>` *with a default port of 1960).*

  For each connection that is made to this port, groovy executes the supplied script on a line by line basis.

  This oneliner will reverse every line that is thrown at it, try `telnet` *to your machine on port 1960 to interact with this script.*

`lhp@nereida:~$ groovy -l -e "println line.reverse()"`
`groovy is listening on port 1960`
`^Clhp@nereida:~$`
	`lhp@nereida:~/LGROOVYBook$ telnet localhost 1960`
	`Trying 127.0.0.1...`
	`Connected to localhost.`
	`Escape character is '^]'.`
	`hola`
	`aloh`
	`dabalearrozalazorraelabad`
	`dabalearrozalazorraelabad`
	`Connection closed by foreign host.`
	`lhp@nereida:~/LGROOVYBook$`

- *You can combine the* `-p` *option from earlier, to automatically print the result of your script. The following one liner is equivalent to the one liner immediately above.*

  ```
  groovy -l -p -e "line.reverse()"
  ```

3.8. Guillaume Laforgue sobre E/S

Véase Heads-up on File and Stream groovy methods por Guillaume Laforgue:

> *In the same spirit, I decided to add some* `append()` *methods to append text at the end of the File.*
>
> ```
> f = new File("myFile.txt").append("Hello World\n")
> f.append("Hello again, that's me!")
> ```
>
> *You can also specify the encoding. Of course, don't write in the same file with different encodings, no editors would be able to read it ;-)*

```
f = new File("myFile.txt").append("Hello World\n", "UTF-16BE")
f.append("Hello again, that's me!", "UTF-16BE")
```

- newWrite(): File.newWriter(encoding)
- File.newWriter(encoding, append)
- File.newWriter(append)
- File.newPrintWriter(encoding)

Missing newWriter() *methods have been added. It's now possible to specify the encoding used to write files, and optionally to specify wether we're in append mode.*

```
wrter = new File("myText.txt").newWriter("UTF-8", true)
writer.write("\u20AC: euro symbol\n")
writer.write("$: dollar symbol\n")
writer.close()
```

- File.newReader(encoding)

Previously, when creating a new Reader *from a file, the encoding was automatically guessed, thanks to CharsetToolkit I had implemented which smartly guessed the charset used to encode the file. Now, it is possible to overide this mechanism:*

```
reader = new File("myText.txt").newReader("UTF-8")
```

- withWriter(): File.withWriter(encoding, closure)
- File.withWriterAppend(encoding, closure)
- OutputStream.withWriter(encoding, closure)

I extended the withWriter methods, so that we may specify an encoding, and also specify the append more:

```
// no need of course to close the writer,
// since it's gracefully taken care of by the method
newFile("myText.txt").withWriterAppend("UTF-8") { writer |
    writer.write("\u20AC: euro symbol\n")
    writer.write("$: dollar symbol\n")
}
```

- BufferedWriter.writeLine(text)

In the example above, you may also use the new writeLine() *method on BufferedReader, which is cleaner than appened* "\n" *after each call to* write()*, so the example becomes:*

```
newFile("myText.txt").withWriterAppend("UTF-8") { writer |
    writer.writeLine("\u20AC: euro symbol") // cleaner than "\n" obviously
    writer.writeLine("$: dollar symbol")
}
```

Note that it appends a platform dependant new line.

- File.readBytes()

I've finally changed the readBytes() *method to return an array of bytes instead of a List of Bytes which was quite ineficient:*

```
bytes = new File("myText.txt").readBytes()
```

With all these methods, I thing we pretty can do anything we want! We should certainly cover all the scope of IO methods, especially of text files with optional specification of the encoding used.

3.9. Directorios

El Método `public void eachFile(Closure)`

El siguiente programa muestra los contenidos de un directorio filtrados según una regexp. Siguen varias ejecuciones. Primero sin argumentos:

```
generaciondecodigos@nereida:~/src/groovy/files$ ./ls.groovy
slurpFile.groovy
writeFile.groovy
env.groovy
.readfile.groovy.swp
ls.groovy
readfile.groovy
.renamefiles.groovy.swp
.svn
writeEuro.groovy
myFile.txt
```

Ahora pasamos como argumento una expresión regular:

```
generaciondecodigos@nereida:~/src/groovy/files$ ./ls.groovy 'groovy$'
slurpFile.groovy
writeFile.groovy
env.groovy
ls.groovy
readfile.groovy
writeEuro.groovy
```

También podemos pasar un argumento adicional referenciando un directorio. En este caso se listarán los ficheros en dicho directorio que casan con la expresión regular:

```
generaciondecodigos@nereida:~/src/groovy/files$ ./ls.groovy 'jar$' /tmp/
groovy-groosh-0.3.5.jar
```

> *The method* `public void eachFile(Closure)` *invokes the closure for each 'child' file in this 'parent' folder/directory. Both regular files and subfolders/subdirectories are processed.*

```
generaciondecodigos@nereida:~/src/groovy/files$ cat -n ls.groovy
#!/usr/bin/env groovy
if (args.length > 1) {
  regexp = args[0]
  dirName = args[1]
}
else if (args.length > 0) {
  regexp = args[0]
  dirName = '.'
}
else {
  regexp = '.*'
  dirName = '.'
}

new File(dirName).eachFile() { file ->
    if (file.getName() =~ regexp) {
```

```
        println file.getName()
      }
  }
```

La clausura recibe un objeto `File`.

> *The method* `public String getName()` *returns the name of the file or directory denoted by this abstract pathname. This is just the last name in the pathname's name sequence. If the pathname's name sequence is empty, then the empty string is returned.*

El Método `public void eachFileMatch(Object, Closure)`
El ejemplo anterior puede reescribirse usando el método `public void eachFileMatch(Object, Closure)`

> *Invokes the closure for each file whose name (*`file.name`*) matches the given* `nameFilter` *in the given directory - calling the* `Object#isCase` *method to determine if a match occurs. This method can be used with different kinds of filters like regular expressions, classes, ranges etc. Both regular files and subdirectories are matched.*

```
groovy:000> x = new File('.').eachFileMatch(~/s.*\.groovy/) { println it }
./slurpFile.groovy
./slurpFile2.groovy
./socket.groovy
./stdin.groovy
===> null
```

El operador `~` aplicado a una cadena produce el objeto `Pattern` que representa el reconocedor del lenguaje definido por la expresión regular especificada en la cadena.

El Método `eachFileRecurse`

> *The method* `public void eachFileRecurse(Closure)` *invokes the closure for each descendant file in this directory. Sub-directories are recursively searched in a depth-first fashion. Both regular files and subdirectories are passed to the closure.*

Ejemplo:

```
groovy:000> x = new File('.').eachFileRecurse() { if (it =~ /[ao]n\.groovy$/) println it }
./objects/DynamicMethodCreation.groovy
./objects/dynamicmethodnamecreation.groovy
./overloading/Numberscohercioncomparison.groovy
./strings/scan.groovy
./strings/StringToBoolean.groovy
===> null
```

3.10. Práctica: Ficheros Grandes y Viejos

Escriba un programa que recibe como argumentos un tamaño `-t tam`, un número de días `-d days` y una lista de nombres de ficheros y directorios y muestra la lista de ficheros cuyo tamaño (método `public long length()`) es superior a `tam` y que no hayan sido modificado en los últimos `days` días (método `public long lastModified()`). Deberá también admitir en la línea de comandos la opción (`-h`) para la ayuda.

3.11. Acceso Directo a un Fichero

El siguiente programa funciona de manera parecida a `tail -f file.log` el cual muestra las últimas líneas del fichero `file.log` actualizándolas conforme este crece.

```
generaciondecodigos@nereida:~/src/groovy/files$ cat -n tail.groovy
#!/usr/bin/env groovy
import java.lang.management.*

if (args.length != 1) {
  println "Usage:\n\ttail.groovy file"
  System.exit 1
}

// something like: 1466@machine
name = ManagementFactory.getRuntimeMXBean().getName();
pid = Integer.parseInt(name[0..name.indexOf("@")-1])
println "Stop with 'kill -9 $pid or CTRL-C'"

fileName = args[0]

if (!(fileName =~ /\.log$/)) fileName += '.log'

logfile = new File(fileName)

sampleInterval = 2000 // 2000 millis = 2 secs

// Create if it does not exists
logfile.createNewFile(); //  Atomically creates a new, empty file if a file with this name does not yet exist.

file = new RandomAccessFile( logfile, "r" )
filePointer = 0 // set to logfile.size() to begin tailing from the end of the file
while( true ) {
    // Compare the length of the file to the file pointer
    long fileLength = logfile.size()
    if( fileLength < filePointer ) {
        // Log file must have been rotated or deleted;
        System.err.println "${new Date()}: Reopening $logfile"
        logfile.createNewFile(); // create if deleted
        file = new RandomAccessFile( logfile, "r" )
        filePointer = 0
    }
    if( fileLength > filePointer ) {
        // There is data to read
        file.seek( filePointer )
        while( (line = file.readLine()) != null ) {
            println '##' + line
        }
        filePointer = file.filePointer
    }
    // Sleep for the specified interval
    Thread.sleep( sampleInterval )
}
```

Al ejecutar el programa se queda esperando por cambios en el fichero vigilado:

```
$ rm my.log
$ ./tail.groovy my
Stop with 'kill -9 11542 or CTRL-C'
```

Tan pronto como en otra consola añadimos algo al fichero, queda reflejado en la salida del programa:

`$ echo "Hola" >> my.log`
`$ echo "Mas Saludos" >> my.log`
	`$ ./tail.groovy my`
	`Stop with 'kill -9 11542 or CTRL-C'`
	`##Hola`
	`##Mas Saludos`

- Las líneas 9-11 calculan el PID del proceso usando esta metodología:

  ```
  groovy:000> import java.lang.management.*
  ===> [import java.lang.management.*]
  groovy:000> name = ManagementFactory.getRuntimeMXBean().getName()
  ===> 30596@nereida
  groovy:000> name.indexOf("@")
  ===> 5
  groovy:000> name.indexOf("3")
  ===> 0
  groovy:000> name.indexOf("5")
  ===> 2
  groovy:000> name[0..name.indexOf("@")-1]
  ===> 30596
  groovy:000> pid = Integer.parseInt(name[0..name.indexOf("@")-1])
  ===> 30596
  groovy:000> pid.class
  ===> class java.lang.Integer
  ```

- `getRuntimeMXBean()` : Returns the managed bean for the runtime system of the Java virtual machine.

  > *Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (e. g. printers) and service oriented networks. Those resources are represented by objects called MBeans (for Managed Bean).*

  ```
  groovy:000> x = ManagementFactory.getRuntimeMXBean()
  ===> sun.management.RuntimeImpl@1ec278b5
  ```

  > *Java Implementation class for the runtime subsystem.*

- Véase la documentación de la clase RandomAccessFile

- > `seek` *: Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.*

- > `getFilePointer()` *: Returns the current offset in this file.*

3.12. Execute

`public Process execute()`

Véase la documentación de `String` sobre `execute()`:

> *Executes the given string as a command line process. For more control over the process mechanism in JDK 1.5 you can use java.lang.*`ProcessBuilder`*.*
>
> *Returns: the* `Process` *which has just started for this command line string*

Sigue un ejemplo de uso:

```
generaciondecodigos@nereida:~/src/groovy/processes$ cat -n retcode.groovy
     1  #!/usr/bin/env groovy
     2  sarg = args.join(' ')
     3  def command = """
     4  ls $sarg
     5  """// Create the String
     6  def proc = command.execute()
     7  proc.waitFor()
     8
     9  println "return code: ${proc.exitValue()}"
    10  println "stderr: ${proc.err.text}"
    11  println "stdout: ${proc.in.text}"
```

Veamos los resultados de dos ejecuciones:

```
generaciondecodigos@nereida:~/src/groovy/processes$ ./retcode.groovy -l *.groovy
return code: 0
stderr:
stdout: -rwxr-xr-x 1 generaciondecodigos generaciondecodigos 177 2009-11-18 11:00 cat2.groovy
-rwxr-xr-x 1 generaciondecodigos generaciondecodigos 172 2009-11-18 11:14 cat.groovy
-rwxr-xr-x 1 generaciondecodigos generaciondecodigos 351 2009-11-18 10:52 grooshEx2.groovy
-rwxr-xr-x 1 generaciondecodigos generaciondecodigos 145 2009-11-18 10:36 grooshEx.groovy
-rwxr-xr-x 1 generaciondecodigos generaciondecodigos 274 2010-03-01 11:19 retcode.groovy

generaciondecodigos@nereida:~/src/groovy/processes$ ./retcode.groovy doesnotexist
return code: 2
stderr: ls: no se puede acceder a doesnotexist: No existe el fichero ó directorio

stdout:
```

Argumentos que Contienen Blancos

> *Take care if you wish to pass a quoted argument that contains white space – it will be split into multiple arguments, e.g.:*
>
> ```
> generaciondecodigos@nereida:~/src/groovy/processes$ ls -l 'a file'
> -rw-r--r-- 1 generaciondecodigos generaciondecodigos 3 2010-03-01 11:23 a file
> generaciondecodigos@nereida:~/src/groovy/processes$ ./retcode.groovy 'a file'
> return code: 2
> stderr: ls: no se puede acceder a a: No existe el fichero ó directorio
> ls: no se puede acceder a file: No existe el fichero ó directorio
>
> stdout:
> ```

The call `"""executable "first with space" second""".execute()` *invokes* `executable` *with the following arguments:*

```
 * arg1 = "first
 * arg2 = with
 * arg3 = space"
 * arg4 = second
```

In such a case, you may prefer to use one of the array or list of String variations, e.g.:

```
["executable", "first with space", "second"].execute()
```

Veamos un ejemplo:

```
~/Lgroovy/files$ ls -l 'a file'
-rw-r--r--  1 casianorodriguezleon  staff  28 16 abr 06:10 a file
~/Lgroovy/files$ groovysh
Groovy Shell (1.6.5, JVM: 1.6.0_17)
Type 'help' or '\h' for help.
-----------------------------------------------------------------------------------------------------------------------------------------------------
groovy:000> r = ['ls', 'a file'].execute()
===> java.lang.UNIXProcess@50a69b6b
groovy:000> r.waitFor()
===> 0
groovy:000> r.exitValue()
===> 0
groovy:000> r.err.text
===>
groovy:000> r.in.text
===> a file
```

3.13. Práctica: Controlando a otros Procesos

Modifique el programa `tail.groovy` visto en la sección 3.11 para que admita una opción `--pid=PID` que haga que el programa termine cuando el proceso con identificador PID muera. Para ello ejecute `ps aux` y analice la salida para comprobar la presencia o ausencia del proceso.

3.14. Groosh

Véase

- Groosh
- Groovy Groosh pages
- download

El siguiente ejemplo muestra como usar Groosh:

```
generaciondecodigos@nereida:~/src/groovy/processes$ cat -n grooshEx.groovy
#!/usr/bin/env groovy
def gsh = new groosh.Groosh();


if (args) {
```

```
  gsh.cat(args[0]).toStdOut();
}
else {
  println 'Provide a file name'
}
generaciondecodigos@nereida:~/src/groovy/processes$ ./grooshEx.groovy file.txt
Hello!
```

3.15. Descargado Ficheros desde una URL

La clase URL permite la descarga de ficheros remotos.

```
groovy:000> u = new URL('http://nereida.deioc.ull.es/~generaciondecodigos/gcexamples/node26.html')
===> http://nereida.deioc.ull.es/~generaciondecodigos/gcexamples/node26.html
groovy:000> u.text[0..100]
===> <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

O también:

```
groovy:000> u = 'http://nereida.deioc.ull.es/~generaciondecodigos/gcexamples/node26.html'.toURL()
===> http://nereida.deioc.ull.es/~generaciondecodigos/gcexamples/node26.html
groovy:000> result = []
===> []
groovy:000> u.eachLine { if (it =~ /href/) result << it }
===> null
groovy:000> result.size()
===> 2
groovy:000> result[0]
===>  <B>Err:</B> <a href="perl_errata_form.html" target="_blank">Si hallas una errata ...</a>
```

Otra forma:

```
groovy:000> u = 'http://nereida.deioc.ull.es/~generaciondecodigos/gcexamples/node26.html'.toURL()
===> http://nereida.deioc.ull.es/~generaciondecodigos/gcexamples/node26.html
groovy:000> u.eachLine { if (it =~ /(?i)href/)println it }
<LINK REL="STYLESHEET" HREF="mystyle.css">
<LINK REL="next" HREF="node27.html">
<LINK REL="previous" HREF="node25.html">
<LINK REL="up" HREF="node22.html">
<LINK REL="next" HREF="node27.html">
..............................................
  HREF="node22.html">Tipos de Datos Simples</A>
  HREF="node25.html">Especificación de Tipos Opcional</A>
 <B>Err:</B> <a href="perl_errata_form.html" target="_blank">Si hallas una errata ...</a>
===> null
groovy:000>
```

El siguiente programa usa la clase para descargar los contenidos de una URL:

```
generaciondecodigos@nereida:~/src/groovy/files$ cat -n wget.groovy
def download(address)
{
    def file = new FileOutputStream(address.tokenize("/")[-1])
    def out = new BufferedOutputStream(file)
    out << new URL(address).openStream()
    out.close()
```

```
}

if (args.length) {
  download(args[0])
}
```

Ejecución:

```
~/src/groovy/files$ groovy wget.groovy 'http://nereida.deioc.ull.es/~generaciondecodigos/gcexamples/node17.html'
~/src/groovy/files$ ls -l node17.html
-rw-r--r-- 1 generaciondecodigos generaciondecodigos 9110 2010-02-10 14:06 node17.html
```

El método `tokenize`

La función `tokenize` descompone la cadena según la cadena delimitadora en una lista de cadenas:

```
groovy:000> "This,,should,have,five,items".tokenize(',')
===> [This, should, have, five, items]
groovy:000> "This,,should,have,five,items".tokenize(',').class
===> class java.util.ArrayList
groovy:000> "This, ,should,have,six,items".tokenize(',')
===> [This,  , should, have, six, items]
groovy:000> "This, ,should,have,six,items".tokenize(',').size()
===> 6
```

Su comportamiento es similar a `split` salvo por el tratamiento de las cadenas vacías, que son ignoradas por `tokenize`:

```
groovy:000> "This,,should,have,five,items".split(',')
===> [Ljava.lang.String;@268c858a
groovy:000> "This,,should,have,five,items".split(',').size()
===> 6
groovy:000> println "This,,should,have,five,items".split(',')
[This, , should, have, five, items]
===> null
```

El Método `openStream`

El método Java `openStream` de la clase `URL` tiene la siguiente declaración:

```
public final InputStream openStream() throws IOException
```

> *Opens a connection to this URL and returns an InputStream for reading from that connection. This method is a shorthand for:*
>
> `openConnection().getInputStream()`

Returns: an input stream for reading from the URL connection. Throws: IOException - if an I/O exception occurs.

3.16. Práctica: Descarga Anidada de Páginas

Añada una opción `-r` al programa `wget.groovy` introducido en la sección 3.15 y que haga que se descarguen las páginas que son enlazadas mediante `href` dentro del fichero especificado en la URL que se pasa como argumento. (Véase también el programa `links.groovy` 3.1).

3.17. El Programa tee Reescrito en Groovy

> *Not all systems support the classic* `tee` *program for splitting output pipes to multiple destinations. This command sends the output from someprog to* `/tmp/output` *and to the mail pipe beyond.*
>
> ```
> % someprog | tee /tmp/output | Mail -s 'check this' user@host.org
> ```
>
> *This program helps not only users who aren't on Unix systems and don't have a regular* `tee`*. It also helps those who are, because it offers features not found on other version of tee.*
>
> *The four flag arguments are* `-i` *to ignore interrupts,* `-a` *to append to output files,* `-u` *for unbuffered output, and* `-n` *to omit copying the output on to standard out.*

Veamos algunas ejecuciones:

```
lhp@nereida:~/Lgroovy/files$ echo 'hola LHP' |./tctee.groovy file1 file2
hola LHP
lhp@nereida:~/Lgroovy/files$ cat file1 file2
hola LHP
hola LHP
lhp@nereida:~/Lgroovy/files$ echo 'hola mundo' |./tctee.groovy -n '| cat -n' file2
     1  hola mundo
lhp@nereida:~/Lgroovy/files$ cat file2
hola mundo
```

```
generaciondecodigos@nereida:~/src/groovy/files$ cat -n tctee.groovy
#!/usr/bin/env groovy
class MultiStream {
    private targets
    private ignoreErrors
    MultiStream(List targets, ignore) {
        this.targets = targets
        ignoreErrors = ignore
    }
    def println(String content) {
        targets.each{
            try {
                it?.write(content.bytes)
            } catch (Exception ex) {
                if (!ignoreErrors) throw ex
                targets -= it
                it?.close()
            }
        }
    }
    def close() { targets.each{ it?.close() } }
}

class TeeTarget {
    private filename
    private stream
    private p

    TeeTarget(String name, append, buffered, ignore) {
```

```
        if (name.startsWith('>>')) {
            createFileStream(name[2..-1],true,buffered,ignore)
        } else if (name.startsWith('|')) {
            createProcessReader(name[1..-1])
        } else {
            createFileStream(name,append,buffered,ignore)
        }
    }

    TeeTarget(OutputStream stream) { this.stream = stream }

    def write(bytes) { stream?.write(bytes) }
    def close() { stream?.close() }

    private createFileStream(name, append, buffered, ignore) {
        filename = name
        def fos
        try {
            fos = new FileOutputStream(name, append)
        } catch (Exception ex) {
            if (ignore) return
        }
        if (!buffered) stream = fos
        else stream = new BufferedOutputStream(fos)
    }
    private createWriter(os) {new PrintWriter(new BufferedOutputStream(os))}
    private createReader(is) {new BufferedReader(new InputStreamReader(is))}
    private createPiperThread(br, pw) {
        Thread.start{
            def next
            while((next = br.readLine())!=null) {
                pw.println(next)
            }
            pw.flush(); pw.close()
        }
    }
    private createProcessReader(name) {
        def readFromStream = new PipedInputStream()
        def r1 = createReader(readFromStream)
        stream = new BufferedOutputStream(new PipedOutputStream(readFromStream))
        p = Runtime.runtime.exec(name)
        def w1 = createWriter(p.outputStream)
        createPiperThread(r1, w1)
        def w2 = createWriter(System.out)
        def r2 = createReader(p.inputStream)
        createPiperThread(r2, w2)
    }
}

targets = []
append = false; ignore = false; includeStdout = true; buffer = true
(0..<args.size()).each{
    arg = args[it]
```

```
    if (arg.startsWith('-')) {
        switch (arg) {
            case '-a': append = true; break
            case '-i': ignore = true; break
            case '-n': includeStdout = false; break
            case '-u': buffer = false; break
            default:
                println "usage: tee [-ainu] [filenames] ..."
                System.exit(1)
        }
    } else targets += arg
}
targets = targets.collect{ new TeeTarget(it, append, buffer, ignore) }
if (includeStdout) targets += new TeeTarget(System.out)
def tee = new MultiStream(targets, ignore)
def stdin = new BufferedReader(new InputStreamReader(System.in))

stdin.eachLine { line ->
    tee.println("$line\n")
}
tee.close()
```

Capítulo 4

Cadenas y Expresiones Regulares

4.1. Cadenas

4.1.1. Introducción

Las cadenas de comillas simples son objetos `String`. Las cadenas de comillas simples no se expanden. Las variables de la clase `String` son inmutables.

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n literals.groovy
     1  str = 'hello'
     2  println str[2]
     3
     4  try {
     5    str[4] = '!'
     6  }
     7  catch (Exception ex) {
     8    println ex
     9  }generaciondecodigos@nereida:~/src/groovy/strings$ groovy literals.groovy
l
groovy.lang.MissingMethodException: No signature of method:
java.lang.String.putAt() is applicable for argument types:
(java.lang.Integer, java.lang.String) values: [4, !]
```

Siguen algunos ejemplos tomados de http://pleac.sourceforge.net/pleac_groovy/strings.html. Las cadenas de comillas dobles pertencence a la clase GString (Véase la entrada de la wikipedia para gstring) y permiten interpolar expresiones. Es posible obtener una cadena que abarca múltiples líneas repitiendo el delimitador tres veces:

```
//----------------------------------------------------------------------------------
string = '\\n'                    // two characters, \ and an n
assert string.size() == 2
string = "\n"                     // a "newline" character
string = '\n'                     // a "newline" character

string = "Jon 'Maddog' Orwant"    // literal single quote inside double quotes
string = 'Jon \'Maddog\' Orwant'  // escaped single quotes

string = 'Jon "Maddog" Orwant'    // literal double quotes inside single quotes
string = "Jon \"Maddog\" Orwant"  // escaped double quotes

string = '''
This is a multiline string declaration
```

```
using single quotes (you can use double quotes)
'''
//------------------------------------------------------------------------------
```

Los ’documentos aqui’ interpolables (véase la entrada Here document de la Wikipedia) - que es como se denominan a las cadenas multilínea - pueden ser muy útiles para crear templates:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n CreateXML.groovy
     1  #!/usr/bin/env groovy
     2
     3  def xmllang(language, author) {
     4    fragment = """
     5    <language name = "${language}">
     6      <author>${author}</author>
     7    </language>
     8  """
     9  }
    10  langs = [ 'C++': 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy' ]
    11  content = '<languages>'
    12  langs.each{language, author ->
    13    content += xmllang(language, author)
    14  }
    15  content += '</languages>'
    16  println  content
generaciondecodigos@nereida:~/src/groovy/strings$ ./CreateXML.groovy
<languages>
  <language name = "C++">
    <author>Stroustrup</author>
  </language>

  <language name = "Java">
    <author>Gosling</author>
  </language>

  <language name = "Lisp">
    <author>McCarthy</author>
  </language>
</languages>
generaciondecodigos@nereida:~/src/groovy/strings$
```

Si una línea termina en escape se une a la siguiente:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n ./MultilineStringsEscape.groovy
     1  #!/usr/bin/env groovy
     2  def name = "James"
     3
     4  def text = """\
     5  hello there ${name} \
     6  how are you today?
     7  """
     8
     9  println(text)
generaciondecodigos@nereida:~/src/groovy/strings$ ./MultilineStringsEscape.groovy
hello there James how are you today?
```

4.1.2. Operadores y Cadenas

Algunos operadores aritméticos han sido extendidos para trabajar con cadenas:

```
generaciondecodigos@nereida:~/src/groovy/strings$ groovysh
Groovy Shell (1.6.5, JVM: 1.6.0_0)
Type 'help' or '\h' for help.
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
groovy:000> a = "Hello"
===> Hello
groovy:000> b = a - 'o'
===> Hell
groovy:000> b = a*2
===> HelloHello
groovy:000> b = a + ' World!'
===> Hello World!
```

Veamos algunos ejemplos tomados de http://pleac.sourceforge.net/pleac_groovy/strings.html

```
//----------------------------------------------------------------------------------
// accessing substrings
string = 'hippopotamus'
start = 5; end = 7; endplus1 = 8
assert string.substring(start, endplus1) == 'pot'
assert string[start..end] == 'pot'

assert string.substring(start) == 'potamus'
assert string[start..-1] == 'potamus'

// String is immutable but new strings can be created in various ways
assert string - 'hippo' - 'mus' + 'to' == 'potato'
assert string.replace('ppopotam','bisc') == 'hibiscus'
assert string.substring(0, 2) + 'bisc' + string[-2..-1] == 'hibiscus'
// StringBuffer is mutable
sb = new StringBuffer(string)
sb[2..-3] = 'bisc'
assert sb.toString() == 'hibiscus'
//----------------------------------------------------------------------------------
```

El método `split` permite obtener una colección a partir de una cadena:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n split.groovy
     1  aaa = '"bread","apple","egg"'
     2  items = aaa.split(',')
     3  assert items[1] == '"apple"'
     4  items.each{ println "item: $it" }
generaciondecodigos@nereida:~/src/groovy/strings$ groovy split.groovy
item: "bread"
item: "apple"
item: "egg"
```

La asignación de cadenas producen una copia de la cadena como muestra el siguiente ejemplo:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n assignments.groovy
     1  println "-------------maps----------"
     2  st = ["status":"test"]
```

```
sn = st
println "Before sn=$sn st=$st"
st.status = "tset"
println "After sn=$sn st=$st"


println "-------------GStrings------------"
st = "test"
sn = st
println "Before sn=$sn st=$st"
st = "tset"
println "After sn=$sn st=$st"

println "-------------Strings------------"
st = 'test'
sn = st
println "Before sn=$sn st=$st"
st = 'tset'
println "After sn=$sn st=$st"

generaciondecodigos@nereida:~/src/groovy/strings$ groovy assignments.groovy
-------------maps----------
Before sn=[status:test] st=[status:test]
After sn=[status:tset] st=[status:tset]
-------------GStrings------------
Before sn=test st=test
After sn=test st=tset
-------------Strings------------
Before sn=test st=test
After sn=test st=tset
```

Here, `sn` *and* `st` *point at the very same map object in memory in the first example, while in the second snippet, at the end,* `st` *points at a different place in memory where there's the new immutable string.*

4.1.3. Conversión a Boolean

toBoolean

public Boolean toBoolean()

Converts the given string into a Boolean object. If the trimmed string is "true", z.ºr "1"(ignoring case) then the result is true othewrwise it is false.

Returns: The Boolean value

Since: 1.0

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n ./StringToBoolean.groovy
#!/usr/bin/env groovy
assert "y".toBoolean()
assert 'TRUE'.toBoolean()
assert '  trUe  '.toBoolean()
assert " y".toBoolean()
assert "1".toBoolean()

assert ! 'other'.toBoolean()
assert ! '0'.toBoolean()
```

```
    10  assert ! 'no'.toBoolean()
    11  assert ! '  FalSe'.toBoolean()
```

4.1.4. GStrings

> *Strings that are declared inside double-quotes (i.e. either single double-quotes or triple double-quotes for multi-line strings) can contain arbitrary expressions inside them as shown above using the* `${expression}` *syntax in a similar way to JSP EL, Velocity and Jexl. Any valid Groovy expression can be enclosed in the* `${...}` *including method calls etc. GStrings are defined the same way as normal* `String`*s would be created in Java. Here is a simple example involving a simple variable and an expression:*
>
> ```
> foxtype = 'quick'
> foxcolor = ['b', 'r', 'o', 'w', 'n']
> println "The $foxtype ${foxcolor.join()} fox"
> // => The quick brown fox
> ```
>
> *What actually happens is whenever a string expression contains a* `${...}` *expression then rather than a normal java.lang.*`String` *instance, a GString object is created which contains the text and values used inside the String.*

En una GString las interpolaciones se hacen mediante una estructura de template en la que en los puntos de interpolación se introducen las referencias al manejador de la sustitución.

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n Gstring.groovy
     1  #!/usr/bin/env groovy
     2  what = new StringBuffer('fence')
     3  text = "The cow jumped over the $what"
     4  println text
     5
     6  what.replace(0,5, "moon")
     7  println text
generaciondecodigos@nereida:~/src/groovy/strings$ ./Gstring.groovy
The cow jumped over the fence
The cow jumped over the moon
```

Este ejemplo funciona porque no hemos cambiado la referencia sino el referente (lo referenciado por `$what`).

Hemos usado el método `replace` de la clase `StringBuffer`:

```
replace

public StringBuffer replace(int start,
                            int end,
                            String str)

Replaces the characters in a substring of this StringBuffer with
characters in the specified String. The substring begins at the
specified start and extends to the character at index end - 1 or
to the end of the StringBuffer if no such character exists. First
the characters in the substring are removed and then the specified
String is inserted at start. (The StringBuffer will be lengthened
to accommodate the specified String if necessary.)

    Parameters:
        start - The beginning index, inclusive.
```

```
        end - The ending index, exclusive.
        str - String that will replace previous contents.
    Returns:
        This string buffer.
    Throws:
        StringIndexOutOfBoundsException - if start is negative, greater than length(), or greater than end.
    Since:
        1.2
```

Veamos que ocurre si cambiamos las variables referenciadas en la GString:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n lazyeval.groovy
#!/usr/bin/env groovy

price = 568.23
company = 'Google'
quote = "Today $company stock closed at $price"
println quote

stocks = [ Apple : 130.01, Microsoft : 35.95 ]
stocks.each { key, val ->
  company = key; val = price;
  println quote
}
```

Lo que ocurre es que `quote` no cambia:

```
generaciondecodigos@nereida:~/src/groovy/strings$ ./lazyeval.groovy
Today Google stock closed at 568.23
Today Google stock closed at 568.23
Today Google stock closed at 568.23
```

La siguiente solución al problema sustituye las referencias a variables por referencias a clausuras que se ejecutan en el momento de la interpolación:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n lazyeval2.groovy
      #!/usr/bin/env groovy

pricec = { it.write("$price") }
companyc =  { it.write(company) }

price = 568.23
company = 'Google'

quote = "Today ${companyc} stock closed at ${pricec}"
println quote

stocks = [ Apple : 130.01, Microsoft : 35.95 ]
stocks.each { key, value ->
  company = key
  price = value
  println quote
}
```

Sigue una ejecución:

```
generaciondecodigos@nereida:~/src/groovy/strings$ ./lazyeval2.groovy
Today Google stock closed at 568.23
Today Apple stock closed at 130.01
Today Microsoft stock closed at 35.95
```

La clausura `pricec` recibe en `it` un objeto de la clase `java.io.StringWriter`. La clase `java.io.StringWriter` provee un flujo de caracteres que recolecta su salida en una cadena. Así las líneas:

```
     3  pricec = { it.write("$price") }
     4  companyc =  { it.write(company) }
```

completan la formación de la cadena `quote`.

El siguiente ejemplo ilustra el uso de la clase `java.io.StringWriter`:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n useStringBuffer.groovy
     1  #!/usr/bin/env groovy
     2  // Create a StringWriter object.
     3  StringWriter writer = new StringWriter();
     4
     5  // Write some data to the StringWriter object.
     6  // Start with the 7th character and write to the end.
     7  String s = "Hello World";
     8  writer.write("Hello World", 6, s.length() - 6);
     9
    10  // Print out the contents of the StringWriter buffer.
    11  println "The StringWriter buffer contains: ${writer.toString()}";
generaciondecodigos@nereida:~/src/groovy/strings$ ./useStringBuffer.groovy
The StringWriter buffer contains: World
```

Esto es lo que dice el manual de Groovy al respecto

> *GString can involve lazy evaluation so it's not until the* `toString()` *method is invoked that the GString is evaluated. This lazy evaluation is useful for things like logging as it allows the calculation of the string, the calls to* `toString()` *on the values, and the concatenation of the different strings to be done lazily if at all. Here is an example that illustrates lazy evaluation:*
>
> ```
> println new Date()
> x = "It is currently ${ new Date() }"
> assert x.values[0] instanceof Date
> y = "It is currently ${ writer -> writer << new Date() }"
> assert y.values[0] instanceof Closure
> sleep 5000
> println x
> println y
> ```
>
> *which outputs the following:*
>
> ```
> Thu Apr 17 23:18:17 EST 2008
> It is currently Thu Apr 17 23:18:17 EST 2008
> It is currently Thu Apr 17 23:18:22 EST 2008
> ```
>
> *To explain this output, we need to understand additional rules about GStrings. Each value in the GString (where* `$` *appears) is the result of an expression. Most often, the expression is evaluated resulting in an Object, as in all the examples above except for the*

case of the variable `y` *in which the expression is a Closure A Groovy Closure is like a çode block.ºr a method pointer. It is a piece of code that is defined and then executed at a later point. It has some special properties like implicit variables, support for currying and support for free variables.*

Only when the GString is converted into a `String` *(as for println) is the expression's result converted and incorporated into the result* `String` *Each of the values to be substituted into the GStringlang/String. Each of the values to be substituted into the GString.html is obtained by applying one of the following methods:*

- *General* `Object` *instances are converted by the* `toString()` *method; there is special handling for types like array, Stream, Collection, etc.*
- *Closure instances taking 0 parameters are called and the returned value is converted to* `String` *by the same rules as for a general Object (a returned Closure will not be called).*
- *Closure instances taking 1 parameter are called with a* `Writer`*s the parameter; characters written to the* `Writer` *as the parameter; characters written to the* `Writer.html`*ecome part of the* `String`*esult. The return from the Closurelang/String result. The return from the Closure is ignored.*
- *Closure instances taking more than 1 parameter are not called; the GString throws* `GroovyRuntimeException`.

So, in the example above, `x` *contains an instance of Date resulting from new* `Date()` *being invoked at the time* `x` *was assigned.*

When we print `x` *later, that calculated value is placed in the appropriate place in the GString and output giving us the date at GString definition time.*

The other GString, `y`*, contains a 1 parameter closure which means that the closure will be stored away in the GString. When we print out* `y`*, the closure will be invoked and give the date at printing time.*

Véase una ejecución de un programa basado en el manual:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n lazygstrings.groovy
     1  println "Starting at ${new Date()}"
     2
     3  sleep(3000)
     4  x = "A few seconds later is ${ new Date() }"
     5  assert x.values[0] instanceof Date
     6
     7  y = "A few more seconds is ${ writer -> writer << new Date() }"
     8  assert y.values[0] instanceof Closure
     9
    10  println x
    11  sleep(3000)
    12  println y

generaciondecodigos@nereida:~/src/groovy/strings$ groovy lazygstrings.groovy
Starting at Tue Nov 17 11:03:42 WET 2009
A few seconds later is Tue Nov 17 11:03:45 WET 2009
A few more seconds is Tue Nov 17 11:03:48 WET 2009
generaciondecodigos@nereida:~/src/groovy/strings$
```

4.2. Introducción a las Expresiones Regulares

4.2.1. Tutorial de Regexps en codehaus.org

El siguiente texto está tomado de las páginas oficiales de Groovy, concretamente de Tutorial 4 - Regular expressions basics.

El siguiente programa puede ser útil para comprobar los ejemplos del tutorial:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n WisnieskiArg.groovy
def checkSpelling(spellingAttempt, spellingRegularExpression)
{
        if (spellingAttempt ==~ spellingRegularExpression)
        {
               println("Congratulations, you spelled it correctly.")
         } else {
               println("Sorry, try again.")
        }
}

theRegularExpression = /Wisniewski/
if (args.length > 0) {
  def attempt = args[0]
  checkSpelling(attempt, theRegularExpression)
}
else {
  println "Provide one argument"
}

```

Sigue un ejemplo de uso:

```
generaciondecodigos@nereida:~/src/groovy/strings$ groovy WisnieskiArg.groovy Wisniewski
Congratulations, you spelled it correctly.
```

> *Regular expressions are the Swiss Army knife of text processing. They provide the programmer the ability to match and extract patterns from strings. The simplest example of a regular expression is a string of letters and numbers. And the simplest expression involving a regular expression uses the* `==~` *operator. So for example to match Dan Quayle's spelling of* `potato`*:*
>
> ```
> "potatoe" ==~ /potatoe/
> ```
>
> *If you put that in the* `groovyConsole` *and run it, it will evaluate to true. There are a couple of things to notice. First is the* `==~` *operator, which is similar to the* `==` *operator, but matches patterns instead of computing exact equality. Second is that the regular expression is enclosed in* `/`*'s. This tells groovy (and also anyone else reading your code) that this is a regular expression and not just a string.*
>
> *But let's say that we also wanted to match the correct spelling, we could add a* `?` *after the* `e` *to say that the* `e` *is optional. The following will still evaluate to true.*
>
> ```
> "potatoe" ==~ /potatoe?/
> ```
>
> *And the correct spelling will also match:*
>
> ```
> "potato" ==~ /potatoe?/
> ```

But anything else will not match:

```
"motato" ==~ /potatoe?/
```

So this is how you define a simple boolean expression involving a regular expression. But let's get a little bit more tricky. Let's define a method that tests a regular expression. So for example, let's write some code to match Pete Wisniewski's last name:

```
def checkSpelling(spellingAttempt, spellingRegularExpression)
{
        if (spellingAttempt ==~ spellingRegularExpression)
        {
               println("Congratulations, you spelled it correctly.")
         } else {
               println("Sorry, try again.")
        }
}

theRegularExpression = /Wisniewski/
checkSpelling("Wisniewski", theRegularExpression)
checkSpelling("Wisnewski", theRegularExpression)
```

There are a couple of new things we have done here. First is that we have defined a function (actually a method, but I'll use the two words interchangably). A function is a collection of code similar to a closure. Functions always have names, whereas closures can be anonymous*". Once we define this function we can use it over and over later.*

In this function the `if` *statement tests to see if the parameter* `spellingAttempt` *matches the regular expression given to the function by using the* `==~` *operator.*

Now let's get a little bit more tricky. Let's say we also want to match the string if the name does not have the `w` *in the middle, we might:*

```
  theRegularExpression = /Wisniew?ski/
  checkSpelling("Wisniewski", theRegularExpression)
  checkSpelling("Wisnieski", theRegularExpression)
  checkSpelling("Wisniewewski", theRegularExpression)
```

The single `?` *that was added to the* `spellingRegularExpression` *says that the item directly before it (the character* `w`*) is optional. Try running this code with different spellings in the variable* `spellingAttempt` *to prove to yourself that the only two spellings accepted are now* `Wisniewski` *and* `Wisnieski`*. (Note that you'll have to leave the definition of* `checkSpelling` *at the top of your* `groovyConsole`*)*

The `?` *is one of the characters that have special meaning in the world of regular expressions. You should probably assume that any punctuation has special meaning.*

Now let's also make it accept the spelling if `ie` *in the middle is transposed. Consider the following:*

```
theRegularExpression = /Wisn(ie|ei)w?ski/
checkSpelling("Wisniewski", theRegularExpression)
checkSpelling("Wisnieski", theRegularExpression)
checkSpelling("Wisniewewski", theRegularExpression)
```

Once again, play around with the spelling. There should be only four spellings that work, `Wisniewski`*,* `Wisneiwski`*,* `Wisnieski` *and* `Wisneiski`*. The bar character* `|` *says that either the thing to the left or the thing to the right is acceptable, in this case* `ie` *or* `ei`*. The parentheses are simply there to mark the beginning and end of the interesting section.*

One last interesting feature is the ability to specify a group of characters all of which are ok. This is done using square brackets `[ ]`. *Try the following regular expressions with various misspellings of Pete's last name:*

```
theRegularExpression = /Wis[abcd]niewski/ // requires one of 'a', 'b', 'c' or 'd'
theRegularExpression = /Wis[abcd]?niewski/ // will allow one of 'a', 'b', 'c' or 'd', but not required (like above)
theRegularExpression = /Wis[a-zA-Z]niewski/ // requires one of any upper\- or lower-case letter
theRegularExpression = /Wis[^abcd]niewski/ // requires one of any character that is '''not''' 'a', 'b', 'c' or 'd'
```

The last one warrants some explanation. If the first character in the square brackets is a `^` *then it means anything but the characters specified in the brackets. The operators*
So now that you have a sense for how regular expressions work, here are the operators that you will find helpful, and what they do:

```
Regular Expression Operators
a?              matches 0 or 1 occurrence of *a*                'a' or empty string
a*              matches 0 or more occurrences of *a*             empty string or 'a', 'aa', 'aaa', etc
a+              matches 1 or more occurrences of *a*            'a', 'aa', 'aaa', etc
a|b             match *a* or *b*                                'a' or 'b'
.               match any single character                      'a', 'q', 'l', '_', '+', etc
[woeirjsd]      match any of the named characters               'w', 'o', 'e', 'i', 'r', 'j', 's', 'd'
[1-9]           match any of the characters in the range        '1', '2', '3', '4', '5', '6', '7', '8', '9'
[^13579]        match any characters not named  even digits, or any other character
(ie)            group an expression (for use with other operators)      'ie'
^a              match an *a* at the beginning of a line         'a'
a$              match an *a* at the end of a line               'a'
```

There are a couple of other things you should know. If you want to use one of the operators above to mean the actual character, like you want to match a question mark, you need to put a `'\'` *in front of it. For example:*

```
// evaluates to true, and will for anything ending in a question mark (that doesn't have a question mark in it)
"How tall is Angelina Jolie?" ==~ /[^\?]+\?/
```

This is your first really ugly regular expression. (The frequent use of these in PERL is one of the reasons it is considered a "write only"language). By the way, google knows how tall she is. The only way to understand expressions like this is to pick it apart:

```
/                                    [^?]                           +                  ?                            /
begin expression        any character other than '?'    more than one of those  a question mark         end expression
```

So the use of the `\` *in front of the* `?` *makes it refer to an actual question mark.*

4.2.2. Celsius to Farenheit

El operador `=~` permite hacer casamientos parciales. Retorna un objeto `util.regex.Matcher`lang/util.regex.Matcherutil/regex/Matcher.html:

```
generaciondecodigos@nereida:~/src/groovy/strings$ groovysh
Groovy Shell (1.6.5, JVM: 1.6.0_0)
Type 'help' or '\h' for help.
-------------------------------------------------------------------------------
groovy:000> ("New York, NY 10292" =~ /\d{5}/).class
===> class java.util.regex.Matcher
groovy:000> ("New York, NY 10292" =~ /\d{5}/)[0]
===> 10292
groovy:000> ("New York, NY" =~ /\d{5}/)[0]
ERROR java.lang.IndexOutOfBoundsException: index is out of range 0..-1 (index = 0)
        at groovysh_evaluate.run (groovysh_evaluate:2)
        ...
```

Como se ve, si no hay casamiento se produce un error, ya que el array está vacío. Es por tanto conveniente asegurarse que se ha producido casamiento:

```
groovy:000> m = ("New York, NY 10292" =~ /\d{5}/)
===> java.util.regex.Matcher[pattern=\d{5} region=0,18 lastmatch=]
groovy:000> if (m) { m[0] }
===> 10292
groovy:000> m = ("New York, NY" =~ /\d{5}/)
===> java.util.regex.Matcher[pattern=\d{5} region=0,12 lastmatch=]
groovy:000> if (m) { m[0] }
===> null
groovy:000> m = ("New York, NY" =~ /\d{5}/)
===> java.util.regex.Matcher[pattern=\d{5} region=0,12 lastmatch=]
groovy:000> m.find()
===> false
```

Si la expresión regular tiene paréntesis de captura en su interior se devuelve un array conteniendo lo que se ha capturado en los paréntesis:

```
groovy:000>  m = "foo car baz" =~ /(.)ar/
===> java.util.regex.Matcher[pattern=(.)ar region=0,11 lastmatch=]
groovy:000> m[0]
===> [car, c]
groovy:000> m[0][1]
===> c
```

Veamos un ejemplo de uso de expresiones regulares en Groovy. el siguiente programa traduce entre temperaturas en escala Celsius y Farenheit:

```
~/src/groovy/strings$ cat -n c2f.groovy
#!/opt/local/bin/groovy
def re = '''(?ix)
  ^                       # begin of string
  \\s*
  ([-+]?\\d+(?:\\.\\d*)?) # number
  \\s*
  ([cf])                  # type
  \\s*
  $                       # end of string
''';

print "Enter a temperature (i.e. 32F, 100C): ";

def stdin = new BufferedReader(new InputStreamReader(System.in))
stdin.eachLine{ input ->
    def m = (input =~ re);

    if (m) {
      def (num, type) = m[0][1..2]
      if ((type == 'C') || (type == 'c')) {
        celsius = num.toDouble()
        farenheit = (celsius * 9/5)+32
    }
    else {
      farenheit = num.toDouble()
```

```
          celsius = (farenheit -32)*5/9;
        }
          printf "%.2f C = %.2f F\n", celsius, farenheit;
        }                      // this message to System.err
        else System.err.println "Expecting a temperature, so don't understand \"$input\""

        print "Enter a temperature (i.e. 32F, 100C): ";
    } // end eachLine loop
```

Ejecución:

```
~/src/groovy/strings$ groovy c2f
Enter a temperature (i.e. 32F, 100C): 32f
0,00 C = 32,00 F
Enter a temperature (i.e. 32F, 100C):    100c
100,00 C = 212,00 F
Enter a temperature (i.e. 32F, 100C): juan
Expecting a temperature, so don't understand "juan"
Enter a temperature (i.e. 32F, 100C): ~/src/groovy/strings$
```

La última vez hemos presionado `CTRL-D` (unix) para generar el final de fichero.

4.3. Los métodos `find` y `findAll`

De nuevo tomado del blog de Ted Naleid Groovy, Grails and OS X tips and tricks:

> *Groovy 1.6.1 released with new find and findAll regexp methods on String*
>
> *Groovy 1.6.1 was released today, and it includes a patch I submitted a few weeks ago to make working with regular expressions much more groovy. Thanks to everyone that voted for the patch in the Groovy JIRA*
>
> *The main functionality is the addition of a variety of find and findAll regular expression aware methods that have been added to string.*
>
> *If all you need is to scan a string for the portion that matches a regular expression, use find:*
>
> ```
> generaciondecodigos@nereida:~/src/groovy/files$ groovysh
> Groovy Shell (1.6.5, JVM: 1.6.0_0)
> Type 'help' or '\h' for help.
> ---------------------------------------------------------------------------
> groovy:000> 'Total Amount: $32.99'.find(/(\d+).(\d{2})/)
> ===> 32.99
> ```
>
> *If you need to get access to the capture groups of the regular expression, you can specify a closure that any matched item will be passed to. You then have a chance to manipulate the return value:*
>
> ```
> generaciondecodigos@nereida:~/src/groovy/strings$ cat -n findNaleid.groovy
>      1  #!/usr/bin/env groovy
>      2  def roundedAmount(value) {
>      3
>      4      value.find(/(\d+).(\d{2})/) { fullMatch, dollars, cents ->
>      5          return dollars.toInteger() + (cents.toInteger() > 50 ? 1 : 0)
>      6      }
> ```

```
}

if (args.length > 0) {
  println roundedAmount(args[0])
}
else {
  println "Provide one phrase with a number with 2 decimal digits ('like 32.23')"
}
generaciondecodigos@nereida:~/src/groovy/strings$ ./findNaleid.groovy 'Me costó 43.25$'
43
generaciondecodigos@nereida:~/src/groovy/strings$ ./findNaleid.groovy 'I paid $44.28 for it'
44
generaciondecodigos@nereida:~/src/groovy/strings$ ./findNaleid.groovy 'Total Amount: $32.99'
33
```

If you're expecting more than one match in a string, you can use the new `findAll` *method to grab the values:*

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n findAllNaleid.groovy
#!/usr/bin/env groovy

if (args.length > 1) {
  regexp = ~args[0]
  string = args[1]

  println string.findAll(regexp)
}
else {
  println "Provide one regexp and one string"
}
generaciondecodigos@nereida:~/src/groovy/strings$ ./findAllNaleid.groovy '(\d+)' '12, 13, 14 and 15'
[12, 13, 14, 15]
generaciondecodigos@nereida:~/src/groovy/strings$ ./findAllNaleid.groovy '(\w+) Fine' "The stooges are: Moe Fine, Howard Fine, and Larry Fine"
[Moe Fine, Howard Fine, Larry Fine]
```

Again, if you want to access the capture groups, just add a closure:

```
def string = "The stooges are: Moe Fine, Howard Fine, and Larry Fine"
def firstNames = string.findAll(/(\w+) Fine/) { match, firstName -> firstName }

assert firstNames == ["Moe", "Howard", "Larry"]
```

All of the methods are "safe" in that they won't blow up if a match for the regular expression isn't found, they'll just return null. This means that they can also be easily used for groovy truth:

```
def withDigits = "There are 4 items"

if (withDigits.find(/\d+/)) {
   assert true
} else {
   assert false
}
```

In addition to taking a regular expression string, there is also a variant of each method that takes a precompiled `Pattern` *object. If you are iterating through a ton of strings (such as lines in a file) this version could be useful if you want to avoid recompiling the regex into a pattern every time:*

```
def pattern = ~/.ow/
def owWords = "how now blue cow".findAll(pattern)
assert owWords == ["how", "now", "cow"]
```

There are additional examples on how to use each of these methods in the groovy documentation for each method:

```
  * eachMatch(Pattern pattern) { fullMatch, group1, ... -> ... }
  * find(String regex)
  * find(String regex) { fullMatch, group1, ... -> ... }
  * find(Pattern pattern)
  * find(Pattern pattern) { fullMatch, group1, ... -> ... }
  * findAll(String regex)
  * findAll(String regex) { fullMatch, group1, ... -> ... }
  * findAll(Pattern pattern)
  * findAll(Pattern pattern) { fullMatch, group1, ... -> ... }
```

Thanks to the groovy committers for their suggestions and quick application of the patch (now they just need to apply my other pending patch to remember the font and window size of the groovy console :)

4.3.1. Práctica: Reescribir el conversor Celsius-Farenheit usando `find`

Reescriba el conversor Celsius-Farenheit usando `find`. rellene las partes que faltan:

```
#!/usr/bin/env groovy
def re = '''(?ix)
  ^                      # begin of string
  \\s*
  ([-+]?\\d+(?:\\.\\d*)?) # number
  \\s*
  ([cf])                 # type
  \\s*
  $                      # end of string
''';

print "Enter a temperature (i.e. 32F, 100C): ";

def stdin = new BufferedReader(new InputStreamReader(System.in))
print "Enter a temperature (i.e. 32F, 100C): ";
stdin.eachLine{ input ->
    ..................... { .................
      if ((type == 'C') || (type == 'c')) {
        celsius = num.toDouble()
        farenheit = (celsius * 9/5)+32
      }
      else {
        farenheit = num.toDouble()
        celsius = (farenheit -32)*5/9;
      }
```

```
      printf "%.2f C = %.2f F\n", celsius, farenheit;
    }                        // this message to System.err
    ....... System.err.println "Expecting a temperature, so don't understand '$input'"

    print "Enter a temperature (i.e. 32F, 100C): ";
}
```

Sigue un ejemplo de ejecución:

```
generaciondecodigos@nereida:~/src/groovy/strings$ ./c2fFindAll.groovy
Enter a temperature (i.e. 32F, 100C): Enter a temperature (i.e. 32F, 100C): 32F
0,00 C = 32,00 F
Enter a temperature (i.e. 32F, 100C): 100C
100,00 C = 212,00 F
Enter a temperature (i.e. 32F, 100C): generaciondecodigos@nereida:~/src/groovy/strings$
```

4.4. Reemplazo

Tomado de Groovy: Don't Fear the RegExp por Ted Naleid

> *In Groovy, the* `String` *class has been enhanced with a few* "replace*" *methods that allow you to leverage regular expressions. These methods originally come from the Matcher class, but attaching them directly to* `String` *puts them right at your fingertips.*
>
> `replaceFirst` *will replace the first substring matched by a regular expression within the specified* `String`*:*

```
~/Lgroovy/strings$ cat -n ReplaceSimple.groovy
     1  a = "Hello"
     2  b = a.replaceFirst(/$/," world!")
     3  println a
     4  println b
~/Lgroovy/strings$ groovy ReplaceSimple.groovy
Hello
Hello world!
```

Nótese que la presencia de `$` en la cadena de reeemplazo altera la sustitución literal:

```
groovy:000> a = "Hello"
===> Hello
groovy:000> b = a.replaceFirst(/$/," world! ($a)")
===> Hello world! (Hello)
```

> `replaceAll` *will replace all matching substrings within the specified* `String`*:*
>
> *There is an alternate version of* `replaceAll` *that takes a closure for the second parameter. This is especially useful in the situations where you want to manipulate the matched value, or groups within the match to dynamically determine the replacement text.*
>
> *For example, if we wanted to be able to turn a dashed phrase (*`foo-bar`*) into a camel case word (*`fooBar`*) we can't just remove all dash characters, we also need to make the first letter after the dash capitalized (the* `B` *in* `fooBar`*).*
>
> *To do this, we can use a regular expression that captures the first letter after a dash in a group using parenthesis.*

```
~/Lgroovy/strings$ cat -n Hyphen2CamelCase.groovy
     1  def dashedToCamelCase(orig) {
     2      orig.replaceAll(/-(\w)/)
```

```
            { fullMatch, firstCharacter ->
                firstCharacter.toUpperCase()
            }
}

def scanner = new Scanner(System.in)
print "Enter dashed string: ";
def dashed = scanner.nextLine()
def camelcase = dashedToCamelCase(dashed)
println camelcase
~/Lgroovy/strings$ groovy Hyphen2CamelCase.groovy
Enter dashed string: one-identifier-with-many-hyphens
oneIdentifierWithManyHyphens
```

Using the version of `replaceAll` *that takes a closure gives us a chance to manipulate the first character of the word and capitalize it. This* closure is always passed the full matched text of the regular expression as the first value, and then any groups as subsequent values.

Here we modify a phone number:

```
~/Lgroovy/strings$ cat -n Phonenumber.groovy
def scanner = new Scanner(System.in)
print 'Enter phone number (\\d{0,3})-(\\d{3})-(\\d{6})/): ';
def pn = scanner.nextLine()
def ac = pn.replaceAll(/(^\s*\d{0,3})[-\s]+(\d{3})[-\s]+(\d{6})\s*$/) {
    fullMatch, international, areaCode, localNumber ->

    return "$areaCode"
}

println ac
~/Lgroovy/strings$ groovy Phonenumber.groovy
Enter phone number (\d{0,3})-(\d{3})-(\d{6})/): 34-922-313131
922
~/Lgroovy/strings$ groovy Phonenumber.groovy
Enter phone number (\d{0,3})-(\d{3})-(\d{6})/): 34 922 313131
922
~/Lgroovy/strings$ groovy Phonenumber.groovy
Enter phone number (\d{0,3})-(\d{3})-(\d{6})/): 34---922--  313131
922
~/Lgroovy/strings$ groovy Phonenumber.groovy
Enter phone number (\d{0,3})-(\d{3})-(\d{6})/): djfkdjs
djfkdjs
```

Ejemplo: poner blanco después de cada coma

He aqui una solución que hace uso de `e` al siguiente ejercicio (véase 'Regex to add space after punctuation sign' en PerlMonks) Se quiere poner un espacio en blanco después de la aparición de cada coma, pero se quiere que la sustitución no tenga lugar si la coma esta incrustada entre dos dígitos. Además se pide que si hay ya un espacio después de la coma, no se duplique

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n commawhite.groovy
def scanner = new Scanner(System.in)
print 'Enter string: '
def line = scanner.nextLine()

```

```
def r = line.replaceAll(/(\d[,.]\d)|(,(?!\s))/) {
    fullMatch, nucnu, stcommanospace ->

    if (nucnu) {
      return nucnu
    }
    return ’, ’
}

println r
```

Se hace uso de un lookahead negativo `(?!\s)` para prohibir la sustitución cuando la coma va seguida de un blanco.

```
generaciondecodigos@nereida:~/src/groovy/strings$ groovy commawhite.groovy
Enter string: 2,3,4,a,b,5,6,c,d, e,f,g
2,3, 4, a, b, 5,6, c, d, e, f, g
```

4.4.1. Práctica: Comentarios C

Escribe un programa Groovy que elimine los comentarios de un programa C. Estudia que son los cuantificadores lazy como `*?` y `+?`. La siguiente sesión con `groovysh` puede ayudarte:

```
groovy:000> m = "/*a*/ /*b*/" =~’(/\\*.*\\*/)’
===> java.util.regex.Matcher[pattern=(/\*.*\*/) region=0,11 lastmatch=]
groovy:000> m[0]
===> [/*a*/ /*b*/, /*a*/ /*b*/]
groovy:000> m[1]
ERROR java.lang.IndexOutOfBoundsException: index is out of range -1..0 (index = 1)
        at groovysh_evaluate.run (groovysh_evaluate:2)
        ...
groovy:000> m = "/*a*/ /*b*/" =~’(/\\*.*?\\*/)’
===> java.util.regex.Matcher[pattern=(/\*.*?\*/) region=0,11 lastmatch=]
groovy:000> m[0]
===> [/*a*/, /*a*/]
groovy:000> m[1]
===> [/*b*/, /*b*/]
```

4.5. Práctica: Ficheros Grandes y Viejos: Unidades

Escriba un programa que recibe como argumentos un tamaño `-t tam`, un número de días `-d days` y una lista de nombres de ficheros y directorios y muestra la lista de ficheros cuyo tamaño (método `public long length()`) es superior a `tam` y que no hayan sido modificado en los últimos `days` días (método `public long lastModified()`). Deberá también admitir en la línea de comandos la opción (`-h`) para la ayuda. El programa deberá admitir que los dias y el tamaño se especifiquen en unidades adecuadas. Por ejemplo una especificación de días `2y3m4d` se traducirá como ”2 años 3 meses y 4 días”, esto es cómo `2*365+3*31+4` días. Una especificación de tamaño como `4M` se traducirá como ”4 MegaBytes.esto es `4*1024*1024`. Para traducir la cadena `2y3m4d` reemplacela (véanse los métodos `replaceFirst` y `replaceAll`) por `2*year+3*month+4` (donde `year` y `month` son constantes predefinidas con los valores adecuados) y después pase a evaluarla. Repase las secciones

- Funciones para la Manipulación de Ficheros 3.4,
- Procesado de la Línea de Comandos 3.5,
- Directorios 3.9 y

- LLamando scripts Groovy desde código Groovy 1.9.

4.6. Colecciones y Expresiones Regulares: grep

El método `grep` permite seleccionar los elementos de una lista:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n grep.groovy
regexp = ~ ".*${args[0]}.*"

args[1 .. -1].each { filename ->

  file = new File(filename)
  list = file.readLines()

  select = list.grep(regexp)
  select.each {
    println it
  }
}
```

Sigue un ejemplo de ejecución:

```
generaciondecodigos@nereida:~/src/groovy/strings$ groovy grep.groovy '\d+' c2f.groovy  grep.groovy
print "Enter a temperature (i.e. 32F, 100C): ";
      def (num, type) = m[0][1..2]
        farenheit = (celsius * 9/5)+32
      celsius = (farenheit -32)*5/9;
      printf "%.2f C = %.2f F\n", celsius, farenheit;
    print "Enter a temperature (i.e. 32F, 100C): ";
regexp = ~ ".*${args[0]}.*"
args[1 .. -1].each { filename ->
```

Colecciones y Expresiones Regulares por Ted Naleid El siguiente parrafo también está tomado de Groovy: Don't Fear the RegExp por Ted Naleid

> *Groovy also makes significant additions to what you can do with Collections. In addition to each, collect, inject, etc, there is a regular expression aware iterator called* `grep` *that will pass each item in the Collection through a filter and return a subset of items that match the filter. We can use a regular expression as a filter:*

```
// regular expression says 0 or more characters (".*") followed by the string "bar" that is at the end of the string ("$")
assert ["foobar", "bazbar"] == ["foobar", "bazbar", "barquux"].grep(~/.*bar$/)
```

> *You can achieve the same thing with* `findAll` *but it takes a little more typing:*

```
assert ["foobar", "bazbar"] == ["foobar", "bazbar", "barquux"].findAll { it ==~ /.*bar$/ }
```

> *Working with Matchers*
>
> *As we've seen, using the* `=~` *operator will return a* `Matcher` *object. Many of the existing regular expression examples on the web work by treating the* `Matcher` *as a list and getting the first (zero-based) element out of the list:*

```
def matcher = "foobazaarquux" =~ "o(b.*r)q"
assert ["obazaarq", "bazaar"] == matcher[0]
assert "bazaar" == matcher[0][1] // get the first grouping of the first map
```

This is a little fragile as `matcher[0]` *will throw an error if there was not actually a match. Calling* `matches()` *doesn't help as matches only checks if the regular expression matches the WHOLE string:*

```
("foobazaarquux" =~ "o(b.*r)q").matches()  // returns false!
("foobazaarquux" =~ ".*(b.*r).*").matches()  // returns true, ".*" matches 0 or more chars of any type
```

You can check `getCount()` *to see how many matches there were for some safety:*

```
def m = "foobar" =~ /quux/
if (m.getCount()) {
        // example won't get here as "quux" doesn't exist in "foobar", the count is 0
        println m[0]
}
```

A groovier way to work with Matchers leverages collection iterators and the built in closures that Groovy provides to them. `Matcher` *supports the* `iterator()` *method and with that, gets everything else that any groovy* `List` *or* `Collection` *would have, including* `collect`*,* `inject`*,* `findAll`*, etc.*

```
def paragraph = """
    Lorem ipsum dolor 12:30 AM sit amet,
    consectetuer adipiscing 1:15 AM elit.
    Nunc rutrum diam sagittis nisi 9:22 PM.
"""

def HOUR = /10|11|12|[0-9]/
def MINUTE = /[0-5][0-9]/
def AM_PM = /AM|PM/
def time = /($HOUR):($MINUTE) ($AM_PM)/

assert ["12:30 AM", "1:15 AM", "9:22 PM"] == (paragraph =~ time).collect { it }

assert ["12:30 AM", "1:15 AM"] == (paragraph =~ time).grep(~/.*AM$/)
```

A limitation of the iterator-based methods is that they don't give you access to the individual groups (hour, minute, am/pm), just the full matched string `("12:30 AM")`*. The* `each` *method is more powerful because as it iterates through, it passes the full match as well as each of the individual groups into the closure.*

```
("foo1 bar30 foo27 baz9 foo600" =~ /foo(\d+)/).each { match, digit -> println "+$digit" }

// result:
// +1
// +27
// +600
```

Another example (using the paragraph and time Matcher from above) showing how to pretty print all of the timestamps:

```
(paragraph =~ time).each {match, hour, minute, amPm ->
    println "$hour:$minute ${amPm == 'AM' ? 'this morning' : 'this evening' }"
}

// result:
// 12:30 this morning
// 1:15 this morning
// 9:22 this evening
```

Regular expressions are a powerful tool that Groovy makes as accessible as any other top-tier scripting language. Using techniques to break more complicated regular expressions into their component pieces can make them much more readable (as in the time example above).

If you're doing any sort of string processing beyond a simple contains or split, regular expressions in groovy can turn mountains of Java into a couple of lines of code.

4.7. La clase `Scanner`

Véase http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html.

La clase `Scanner` provee un sencillo scanner que puede analizar tipos Java primitivos utilizando expresiones regulares.

Un scanner descompone su entrada en tokens utilizando un patrón delimitador, que por defecto es blancos. Los tokens resultantes pueden ser convertidos en diversos tipos de valores usando diversos métodos `next`.

`next` y `hasNext`

The `next()` *and* `hasNext()` *methods and their primitive-type companion methods (such as* `nextInt()` *and* `hasNextInt()`*) first skip any input that matches the delimiter pattern, and then attempt to return the next token. Both* `hasNext` *and* `next` *methods may block waiting for further input. Whether a* `hasNext` *method blocks has no connection to whether or not its associated next method will block.*

A scanner will default to interpreting numbers as decimal unless a different radix has been set by using the `useRadix(int)` *method.*

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n UseScanner3.groovy
     1  #!/usr/bin/env groovy
     2  // On a Java 5 or 6 JVM, Groovy can also make use of Scanners:
     3  Scanner sc = new Scanner(new File("myNumbers"));
     4  sc.useDelimiter("\\D+")
     5  while (sc.hasNext()) {
     6    n = sc.next();
     7    println "One more $n"
     8  }
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n myNumbers
     1  123
     2  ab 456
     3  78910
     4  45
generaciondecodigos@nereida:~/src/groovy/strings$ ./UseScanner3.groovy
One more 123
One more 456
One more 78910
One more 45
```

`findInLine`

The `findInLine(java.lang.String)`*,* `findWithinHorizon(java.lang.String, int)`*, and* `skip(java.util.regex.Pattern)` *methods operate independently of the delimiter pattern. These methods will attempt to match the specified pattern with no regard to delimiters in the input and thus can be used in special circumstances where delimiters are not relevant. These methods may block waiting for more input.*

Esto es lo que dice el manual sobre `findInLine`:

> `public String findInLine(Pattern pattern`*)*
>
> *Attempts to find the next occurrence of the specified pattern ignoring delimiters. If the pattern is found before the next line separator, the scanner advances past the input that matched and returns the string that matched the pattern. If no such pattern is detected in the input up to the next line separator, then null is returned and the scanner's position is unchanged. This method may block waiting for input that matches the pattern.*
>
> *Since this method continues to search through the input looking for the specified pattern, it may buffer all of the input searching for the desired token if no line separators are present.*
>
> - *Parameters: pattern - the pattern to scan for*
> - *Returns: the text that matched the specified pattern*
> - *Throws: IllegalStateException - if this scanner is closed*
>
> *Veamos un ejemplo:*

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n findInLine.groovy
     1  instr = "Name: Joe Age: 28 ID: 77";
     2
     3  Scanner conin = new Scanner(instr);
     4
     5  conin.findInLine("Age:"); // find Age
     6
     7  while (conin.hasNext()) {
     8    n = conin.next();
     9    println n
    10  }
generaciondecodigos@nereida:~/src/groovy/strings$ groovy findInLine.groovy
28
ID:
77
```

El siguiente ejemplo recoje el match en un objeto:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n UseScanner.groovy
     1  #!/usr/bin/env groovy
     2  // On a Java 5 or 6 JVM, Groovy can also make use of Scanners:
     3  data = 'hippopotamus means river horse'
     4  println data
     5
     6  s = new Scanner(data)
     7  s.findInLine(/(.{5}).{8}(.{5}) (.{5}) (.*)/)
     8  m = s.match()
     9  println m.groupCount()
    10
    11  fields = []
    12  (1..m.groupCount()).each{ fields << m.group(it) }
    13
    14  assert fields == ['hippo', 'means', 'river', 'horse']
    15  println fields
generaciondecodigos@nereida:~/src/groovy/strings$ ./UseScanner.groovy
hippopotamus means river horse
4
[hippo, means, river, horse]
```

Locales

> *An instance of this class is capable of scanning numbers in the standard formats as well as in the formats of the scanner's locale. A scanner's initial locale is the value returned by the Locale.getDefault() method; it may be changed via the useLocale(*`java.util.Locale`*) method.*
>
> *The localized formats are defined in terms of the following parameters, which for a particular locale are taken from that locale's DecimalFormat object, df, and its and DecimalFormatSymbols object, dfs.*

`LocalGroupSeparator`	`The character used to separate thousands groups, i.e., dfs.getGroupingSeparator()`
`LocalDecimalSeparator`	`The character used for the decimal point, i.e., dfs.getDecimalSeparator()`
`LocalPositivePrefix`	`The string that appears before a positive number (may be empty), i.e., df.getPositivePrefix()`
`LocalPositiveSuffix`	`The string that appears after a positive number (may be empty), i.e., df.getPositiveSuffix()`
`LocalNegativePrefix`	`The string that appears before a negative number (may be empty), i.e., df.getNegativePrefix()`
`LocalNegativeSuffix`	`The string that appears after a negative number (may be empty), i.e., df.getNegativeSuffix()`
`LocalNaN`	`The string that represents not-a-number for floating-point values, i.e., dfs.getInfinity()`
`LocalInfinity`	`The string that represents infinity for floating-point values, i.e., dfs.getInfinity()`

> *The strings that can be parsed as numbers by an instance of this class are specified in terms of the following regular-expression grammar, where* `Rmax` *is the highest digit in the radix being used (for example,* `Rmax` *is 9 in base 10).*

El siguiente ejemplo lee números en formato español de un fichero y los suma:

```
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n ScanSum.groovy
     1  Scanner s = null;
     2  sum = 0.0;
     3  s = new Scanner(new File("esnumbers.txt"));
     4  s.useLocale(Locale.FRANCE);
     5
     6  while (s.hasNext()) {
     7    if (s.hasNextDouble()) {
     8            next = s.nextDouble();
     9            sum += next;
    10            println "next = $next sum=$sum"
    11    } else {
    12        s.next();
    13    }
    14  }
    15  s.close();
    16
    17  System.out.println(sum);
generaciondecodigos@nereida:~/src/groovy/strings$ cat -n esnumbers.txt
     1  3,5
     2  4,2
     3  83
generaciondecodigos@nereida:~/src/groovy/strings$ groovy ScanSum.groovy
next = 3.5 sum=3.5
next = 4.2 sum=7.7
next = 83.0 sum=90.7
90.7
```

4.8. Lectura y Reconocimiento de Números Complejos

Sigue un ejemplo:

```
~/Lgroovy/overloading$ cat -n  ComplexNumber.groovy
class MyComplex  {
  def real, imaginary

  def plus(other) {
    new MyComplex(
          real: real+other.real,
          imaginary: imaginary+other.imaginary
    )
  }

  String toString() {
    "$real "+(imaginary > 0? "+${imaginary}i" : '')
  }

  def static _numre = /\s*([-+]?\d+(?:\.\d*)?)\s*/
  def static _complexre = ~/^\s*${_numre}\s*(?:\+\s*${_numre}i)?$/

  def static read() {

    def scanner = new Scanner(System.in)

    print "Enter a complex number (e.g. 2+1i): ";
    def input = scanner.nextLine()

    def m = (input =~ _complexre);
    if (m) {
      m = m[0]
      def re = m[1].toDouble()
      def im = m[2]? m[2].toDouble() : 0.0;

      return new MyComplex(real : re, imaginary: im)
    }

    throw new Exception("\n\tExpected a complex number. Found: '$input'\n")
  }

}

c1 = MyComplex.read()
c2 = MyComplex.read()

println c1+c2
```

Ejecución:

```
~/Lgroovy/overloading$ groovy ComplexNumber.groovy
Enter a complex number (e.g. 2+1i): 2+3i
Enter a complex number (e.g. 2+1i): 4+5i
6.0 +8.0i
~/Lgroovy/overloading$ groovy ComplexNumber.groovy
```

```
Enter a complex number (e.g. 2+1i): juf+plos i
Caught: java.lang.Exception:
        Expected a complex number. Found: 'juf+plos i'

        at MyComplex.read(ComplexNumber.groovy:34)
        at ComplexNumber.run(ComplexNumber.groovy:39)
~/Lgroovy/overloading$ groovy ComplexNumber.groovy
Enter a complex number (e.g. 2+1i): 2
Enter a complex number (e.g. 2+1i): 3
5.0
~/Lgroovy/overloading$ groovy ComplexNumber.groovy
Enter a complex number (e.g. 2+1i): 2+i
Caught: java.lang.Exception:
        Expected a complex number. Found: '2+i'

        at MyComplex.read(ComplexNumber.groovy:34)
        at ComplexNumber.run(ComplexNumber.groovy:39)
```

4.9. Práctica: DEFINEs en Programas C

Se trata de escribir un programa Groovy que lea un fichero fuente en C reconociendo las definiciones de macros

```
#define PI 3.14159
```

y produzca como salida una lista con las parejas nombre, definición. Para mas detalles lea la sección 4.1.1 'Code Munger 1' del capítulo 4 (disponible en la web de Manning) del libro *Code Generation in Action* de Jack Herrington

Capítulo 5

Colecciones

5.1. Expando

The `Expando` *is not a collection in the strictest sense, but in some ways it is similar to a* `Map`*, or objects in JavaScript that do not have to have their properties defined in advance. It allows you to create dynamic objects by making use of Groovy's closure mechanisms. An* `Expando` *is different from a map in that you can provide synthetic methods that you can call on the object.*

```
~/Lgroovy/collections$ cat -n expando1.groovy
def player = new Expando()
player.name = "Dierk"
player.greeting = { "Hello, my name is $name" }

println player.greeting()
player.name = "Jochen"
println player.greeting()

~/Lgroovy/collections$ groovy expando1.groovy
Hello, my name is Dierk
Hello, my name is Jochen
```

The player.greeting assignment passes in a closure to execute when `greeting()` *is called on the* `Expando`*. Notice that the closure has access to the properties assigned to the* `Expando`*, even though these values may change over time, using Groovy's* `GString "$variableOrProperty"` *notation.*

Sigue otro ejemplo que muestra diferencias de conducta entre un `Map` y un `Expando`:

```
~/Lgroovy/objects$ cat -n expando.groovy
def map_boy = [age: 6, doSomething: { println 'pick nose'}, toString: {'hi'}]
def exp_boy = new Expando(age: 6, doSomething: { println 'pick nose' }, toString:{'hi'} )

def boys = [map_boy, exp_boy]

boys.each{boy->
  println boy.age
  boy.doSomething()
  println boy.toString()
}
~/Lgroovy/objects$ groovy expando.groovy
6
```

```
pick nose
[age:6, doSomething:expando$_run_closure1@6d732ed2, toString:expando$_run_closure2@25071521]
6
pick nose
hi
```

Véase la cuestión Still fuzzy on the Expando class - have any good examples of its use? en coderanch:

> *Both map and*`Expando` *allowed me to mock out properties and closures. The closures let me make calls on the map or*`Expando` *as if I were calling a normal method.*
>
> *The difference was on* `toString` *and my hunch is that* `equals` *and* `hashCode` *work the same way* -`Expando` *lets us essentially override them while map does not.*
>
> *If this truly is the only difference, I think I'd rather stick to using maps unless I absolutely need to use toString, hashCode, or equals.*

Capítulo 6

Clausuras

6.1. Introducción

En palabras de la Wikipedia:

> *In computer science, a closure is a first-class function with free variables that are bound in the lexical environment. Such a function is said to be çlosed over”its free variables. A closure is defined within the scope of its free variables, and the extent of those variables is at least as long as the lifetime of the closure itself.*

Un ejemplo:

```
generaciondecodigos@nereida:~/src/groovy/closures$ cat -n closure.groovy
#!/usr/bin/env groovy

def p(x) {
  return { y -> x + y}
}

p2 = p(2)

println p2(5)
println p2(9)

println p2(4.5)

ph = p("hola ")
println ph("mundo")

```

Al ejecutar este programa obtenemos la salida:

```
generaciondecodigos@nereida:~/src/groovy/closures$ ./closure.groovy
7
11
6.5
hola mundo
```

6.2. Uso de métodos como clausuras

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n closure2.groovy
#!/usr/bin/env groovy
class Example {
```

```
  def limit

  Example(limit) {
    this.limit = limit
  }

  boolean validate(value) {
    value.length() <= limit
  }
}

first = new Example(6)
second = new Example(5)

firstClosure = first.&validate

words = ['long string', 'medium', 'short', 'tiny']

println words.find(firstClosure)
println words.find(second.&validate)
```

- El método `find` de los objetos de la clase `Collection` permite encontrar el primer elemento de una lista que cumple una condición:

  ```
  def fibList = [1, 1, 2, 3, 5, 8, 13]
  assert fibList.find { it > 1 } == 2
  ```

- La sintáxis `reference.&somemethod` permite obtener una clausura a partir de un método

Ejecución:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ ./closure2.groovy
medium
short
```

La construcción de clausuras a partir de métodos se limita a métodos de objeto (instance methods).

6.3. Multimétodos

Las clausuras de multimétodos actúan como multimétodos. Recuérdese la definición que de multimétodos da la wikipedia:

> *Multiple dispatch or multimethods is the feature of some object-oriented programming languages in which a function or method can be dynamically dispatched based on the run time (dynamic) type of more than one of its arguments. This is an extension of single dispatch polymorphism where a method call is dynamically dispatched based on the actual derived type of the object. Multiple dispatch generalizes the dynamic dispatching to work with a combination of two or more objects.*

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n multipledispatch.groovy
#!/usr/bin/env groovy
/*
Multiple dispatch or multimethods is the feature of some object-oriented
programming languages in which a function or method can be dynamically
```

```
dispatched based on the run time (dynamic) type of more than one
of its arguments. This is an extension of single dispatch polymorphism
where a method call is dynamically dispatched based on the actual
derived type of the object. Multiple dispatch generalizes the dynamic
dispatching to work with a combination of two or more objects.
*/
class MultiMethod {
  int f(String val) {
    val.length()
  }

  int f(List list) {
    list.size()
  }

  int f(int x, int y) {
    x+y
  }
}

x = new MultiMethod()

println "Using method call:"
println "On a string: ${x.f('string argument')}"
println "On a list: ${x.f([1, 2, 3, 4, 'string'])}"
println "On numbers: ${x.f(3, 4)}"

c = x.&f

println "Using closure:"
println "On a string: ${c('string argument')}"
println "On a list: ${c([1, 2, 3, 4, 'string'])}"
println "On numbers: ${c(3, 4)}"
generaciondecodigos@nereida:~/Lgroovy/closures$
```

Sigue el resultado de una ejecución:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ ./multipledispatch.groovy
Using method call:
On a string: 15
On a list: 5
On numbers: 7
Using closure:
On a string: 15
On a list: 5
On numbers: 7
```

Cuando se trabaja con multimétodos pueden surgir dudas sobre que método concreto será llamado. ¿Que métodos serán llamados en este ejemplo?

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n mm.groovy
#!/usr/bin/env groovy
class Person {
    String name
```

```
    Person(String name) {
        this.name = name
    }

    Person() {
        this.name = ""
    }

    boolean equals(Object o) {
        println("equals(Object o)")
        // delegate on Groovy equality
        name == o
    }

    boolean equals(Person emp) {
        println("equals(Person emp)")
        name == emp.name
    }
}

     void test1() {
        println "==================== test1"
        Object e1 = new Person("Juana")
        Object e2 = new Person("Juana")
        println(e1.equals(e2))       // Person
        println(e1.equals(e1))       // Person
        println(e1.equals("Juana"))  // Object
    }

     void test2() {
        println "==================== test2"
        Person e1 = new Person("Juana")
        Person e2 = new Person("Juana")
        println(e1.equals(e2))        // Person
        println(e1.equals(e1))        // Person
        println(e1.equals("Juana"))   // Object
    }


     void test3() {
        println "==================== test3"
        e1 = new Person("Juana")
        e2 = new Person("Juana")
        println(e1.equals(e2))        // Person
        println(e1.equals(e1))        // Person
        println(e1.equals("Juana"))   // Object
    }


test1()
test2()
test3()
```

En la primera prueba los objetos son explícitamente declarados de la clase `Object`. En la segunda de la clase `Person`. En la tercera no hay declaración explícita.

El resultado de las ejecuciones es el mismo en los tres casos:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ ./mm.groovy
==================== test1
equals(Person emp)
true
equals(Person emp)
true
equals(Object o)
true
==================== test2
equals(Person emp)
true
equals(Person emp)
true
equals(Object o)
true
==================== test3
equals(Person emp)
true
equals(Person emp)
true
equals(Object o)
true
```

6.4. Creando y LLamando clausuras

El siguiente ejemplo muestra varios métodos de creación y uso de clausuras:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n creatingandusingclosures.groovy
#!/usr/bin/env groovy

map = [ 'a' : 1, 'b' : 2 ]
println map

map.each { key, val -> map[key] *= 2 }

println map

map.each {  map[it.key] *= 2 }

println map

duplicate = {  map[it.key] *= 2 }

map.each duplicate

println map

def dup(e) {
  map[e.key] *= 2
}
```

```

map.each this.&dup

println map
```

Las clausuras pueden

1. Declararse y usarse sobre la marcha:

   ```
   map.each { key, val -> map[key] *= 2 }
   ```

2. Asignarse a variables:

   ```
   duplicate = {  map[it.key] *= 2 }
   ```

3. Construirse a partir de métodos:

   ```
   map.each this.&dup
   ```

Ejecución:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ ./creatingandusingclosures.groovy
[a:1, b:2]
[a:2, b:4]
[a:4, b:8]
[a:8, b:16]
[a:16, b:32]
```

6.5. La clase `Eval`

El siguiente ejemplo usa la clase groovy.util.Eval para leer desde la entrada estandar el texto de descripción de una función

$$f : \mathcal{Z} \rightarrow \mathcal{Z} \tag{6.1}$$

$$x \rightarrow f(x) \tag{6.2}$$

a continuación solicitar un entero para posteriormente construir una clausura a partir del texto de descripción de la función y evaluarla sobre el entero obtenido (véase la documentación de la clase groovy.util.Eval).

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n eval.groovy
#!/usr/bin/env groovy

input = new Scanner(System.in)
print "What's your function(x)? "
functiontxt = input.nextLine()
functiontxt = "{ x -> $functiontxt }"

print "x? "
x = input.nextLine().toInteger()

eo = new Eval()
df = eo.x(x, functiontxt)

println "f($x) = ${df(x)}"
```

El manual de groovy.util.Eval dice sobre el método `x`:

```
public static Object x(Object x, String expression) throws CompilationFailedException

    evaluate expression and make x available inside the expression as 'x'

    Parameters:
        expression - the Groovy expression to evaluate
    Returns:
        the result of the expression
    Throws:
        CompilationFailedException - if expression is no proper Groovy
```

Ejecución:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ ./eval.groovy
What's your function(x)? x+2
x? 4
f(4) = 6
generaciondecodigos@nereida:~/Lgroovy/closures$ ./eval.groovy
What's your function(x)? 2*x
x? 5
f(5) = 10
```

6.6. Valores por defecto

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n default.groovy
#!/usr/bin/env groovy

add = { x, y = 2 -> x+y }

println add(4,9)
println add(4)
```

Ejecución:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ ./default.groovy
13
6
```

6.7. El método `call` de los objetos `Closure`

Si `x` es una referencia a una clausura, podemos llamarla usando la sintáxis `x()` o bien `x.call()`:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n call.groovy
#!/usr/bin/env groovy

def benchmark(number, task) {

  start = System.currentTimeMillis()

    number.times { task.call(it) }

```

```
  stop = System.currentTimeMillis()

  (stop - start)/1.0E03;
}

def num = 10000

if (args.length > 0) {
  Scanner s = new Scanner(args[0]);
  if (s.hasNextInt()) {
    num = s.nextInt();
  }
}

floatd = benchmark(num) { (int) it/2 }
intd = benchmark(num) { it.intdiv(2) }

println "float division $floatd s. Integer division: $intd s. Ratio = ${floatd/intd}"
```

El método `currentTimeMillis` tiene la siguiente firma:

```
public static long currentTimeMillis()
```

Esto es lo que dice el manual sobre `currentTimeMillis`:

```
Returns the current time in milliseconds. Note that while the unit
of time of the return value is a millisecond, the granularity of
the value depends on the underlying operating system and may be
larger. For example, many operating systems measure time in units
of tens of milliseconds.

Returns:
    the difference, measured in milliseconds, between the current
    time and midnight, January 1, 1970 UTC.
```

Siguen algunas ejecuciones:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ ./call.groovy
float division 0.343 s. Integer division: 0.046 s. Ratio = 7.4565217391
generaciondecodigos@nereida:~/Lgroovy/closures$ ./call.groovy 100000
float division 1.191 s. Integer division: 0.082 s. Ratio = 14.5243902439
generaciondecodigos@nereida:~/Lgroovy/closures$ ./call.groovy xyz
float division 0.364 s. Integer division: 0.059 s. Ratio = 6.1694915254
```

6.8. Número y tipo de los parámetros de una clausura

Es posible obtener el número y tipo de parámetros que espera una clausura:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n op.groovy
#!/usr/bin/env groovy

def randapply(f) {
  random = new Random()

  if (f.getParameterTypes().size() == 2) {
```

```
    x = random.nextInt(50)
    y = random.nextInt(50)
    println "f($x, $y) = ${f(x, y)}"
  }
  else {
    x = random.nextInt(10)
    x *= x
    println "f($x) = ${f(x)}"
  }
}

randapply { x, y -> x+y }
randapply { Math.sqrt(it) }
```

Un ejemplo de como utilizar esta información lo a el método `each` de los `map`s. Si se le pasan dos parámetros los interpreta como una pareja clave-valor. Si se le pasa uno sólo lo interpreta como un objeto `Map.Entry`:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ cat -n creatingandusingclosures.groovy
#!/usr/bin/env groovy

map = [ 'a' : 1, 'b' : 2 ]
println map

map.each { key, val -> map[key] *= 2 }

println map

map.each {  map[it.key] *= 2 }

println map

duplicate = {  map[it.key] *= 2 }

map.each duplicate

println map

def dup(e) {
  map[e.key] *= 2
}

map.each this.&dup

println map
```

El método `getParameterTypes` tiene la firma:

```
public Class[] getParameterTypes()
```

El manual dice:

```
Returns an array of Class objects that represent the formal parameter
types, in declaration order, of the method represented by this Method
object. Returns an array of length 0 if the underlying method takes
```

```
    no parameters.

    Returns:
        the parameter types for the method this object represents
```

Ejecución:

```
generaciondecodigos@nereida:~/Lgroovy/closures$ groovy op.groovy
f(39, 24) = 63
f(9) = 3.0
generaciondecodigos@nereida:~/Lgroovy/closures$ groovy op.groovy
f(28, 0) = 28
f(64) = 8.0
```

Sigue un ejemplo en el que examinamos las firmas de los métodos de una clase:

```
generaciondecodigos@nereida:~/src/java/reflection$ cat -n GetPar.groovy
import java.lang.reflect.*

class MyClass
{
    // Declare the "sample" method - just for testing:
    public void sample(int i, float f, double d, String str, byte[] ba)
    { }
}

// Declare a Class object:
Class mc = MyClass.class

// Declare an array of methods:
Method[] methArr = mc.getMethods()

// Get the parameter types:
for (i in 0..<methArr.length) {
    if ("sample" != methArr[i].getName()) continue
    println(methArr[i])
    println("The parameter types are:")

    // Declare the array of parameter types:
    Class[] paramTyArr = methArr[i].getParameterTypes()
    k = 1
    paramTyArr.each { println("${k++}\t$it") }
}
```

La ejecución produce la salida:

```
generaciondecodigos@nereida:~/src/java/reflection$ groovy GetPar.groovy
public void MyClass.sample(int,float,double,java.lang.String,byte[])
The parameter types are:
1       int
2       float
3       double
4       class java.lang.String
5       class [B
```

6.9. Currying

Introducción

De la Wikipedia:

> *In mathematics and computer science, currying or Schönfinkelisation, invented by Moses Schönfinkel and later re-invented by Haskell Curry, is the technique of transforming a function that takes multiple arguments (or an n-tuple of arguments) in such a way that it can be called as a chain of functions each with a single argument.*

Véase Practically Groovy: Functional programming with curried closures (Groovy's everyday coding construct goes where no closure has gone before) Recojemos un extracto que introduce las clausuras:

> *. . . You've probably never cooked with* `curry()` *before, so we'll start from a simple, familiar base. The listing shows a closure referenced as multiply. It has formal parameters* `x` *and* `y` *and returns the product of these two values . . .*

```
generaciondecodigos@nereida:~/src/groovy/curry$ cat -n hellocurry.groovy
     1  #!/usr/bin/env groovy
     2
     3  def multiply = { x, y -> return x * y }  // closure
     4
     5  // triple = { y -> return 3 * y }
     6  def triple = multiply.curry(3)
     7
     8  // quadruple = { y -> return 4 * y }
     9  def quadruple = multiply.curry(4)
    10
    11  def p = triple.call(4)                   // explicit call
    12  println "p: ${p}"                        // p is 12
    13
    14  def q = quadruple(5)                     // implicit call
    15  println "q: ${q}"                        // q is 20
    16
generaciondecodigos@nereida:~/src/groovy/curry$ ./hellocurry.groovy
p: 12
q: 20
```

> *This closure is all well and good, but we're going to curry it up just the same. When calling the* `curry()` *method you need not supply the full complement of actual parameters. The curried call gives rise to the partial application of the closure. The partial application of a closure is another Closure object in which some values have been fixed.*

Ejemplo de Currying: Logging

La currificación permite seguir una técnica en la que

1. Se comienza por definir funciones de una gran generalidad y abstracción que contemplan el esquema general de resolución pero cuya conducta esta parametrizada en conductas (esto es, en clausuras)

2. Se derivan progresivamente mediante currificación - es decir definiendo los parámetros conductuales, las clausuras - familias de soluciones al problema

Sigue un ejemplo modificado del libro de König 'Groovy in Action':

```
~/src/groovy/curry$ cat -n log.groovy
#!/usr/bin/env groovy

def configurator = { format, filter, line ->
  filter(line) ? format(line) : null
}

def appender = { config, append, line ->
  def out = config(line)
  if (out) append(out)
}

/*****************************************/

def dateFormatter   = { line -> "${new Date()}: $line" }
def debugFilter     = { line -> line.contains('debug') }
def consoleAppender = { line -> println line }

/*****************************************/

def myConf = configurator.curry(dateFormatter, debugFilter)
def myLog  = appender.curry(myConf, consoleAppender)

/*****************************************/

myLog '*********** debug:myLog ************'
myLog 'here is a debug message with myLog'
myLog 'here is a warning message (does not appear)'
myLog 'here is an error message (does not appear)'
myLog 'here is another debug message'

/*****************************************/
/* Another Log */
import java.lang.management.*
// something like: 1466@machine
name = ManagementFactory.getRuntimeMXBean().getName();
pid = Integer.parseInt(name[0..name.indexOf("@")-1])

def datePIDFormatter   = { line -> "${new Date()}:$pid: $line" }
def noFilter     = { line -> 1 }
def stderrAppender = { line -> System.err.println line }

def aConf = configurator.curry(datePIDFormatter, noFilter)
def aLog  = appender.curry(aConf, stderrAppender)

aLog '*********** aLog ************'
aLog 'here is a debug message'
aLog 'here is a warning message'
aLog 'here is an error message'
aLog 'here is another debug message'
```

The package `java.lang.management` *provides the management interface for monitoring and management of the Java virtual machine as well as the operating system on which the Java virtual machine is running.*

The interface `RuntimeMXBean` *is The management interface for the runtime system of the Java virtual machine. A Java virtual machine has a single instance of the implementation class of this interface. This instance implementing this interface is an* `MXBean`[1] *that can be obtained by calling the* `ManagementFactory.getRuntimeMXBean()` *method.*

The `getName` *method returns the name representing the running Java virtual machine. The returned name string can be any arbitrary string and a Java virtual machine implementation can choose to embed platform-specific useful information in the returned name string. Each running virtual machine could have a different name.*

La siguiente sesión en la shell Groovy explica el significadod e las líneas 33-36:

```
generaciondecodigos@nereida:~/src/java/reflection$ groovysh
Groovy Shell (1.6.5, JVM: 1.6.0_0)
Type 'help' or '\h' for help.
----------------------------------------------------------------
groovy:000> import java.lang.management.*
===> [import java.lang.management.*]
groovy:000> ManagementFactory
===> class java.lang.management.ManagementFactory
groovy:000> ManagementFactory.getRuntimeMXBean()
===> sun.management.RuntimeImpl@384ab40a
groovy:000> name = ManagementFactory.getRuntimeMXBean().getName()
===> 29139@nereida
groovy:000> pid = Integer.parseInt(name[0..name.indexOf("@")-1])
===> 29139
groovy:000>
```

Veamos algunas ejecuciones de `log.groovy`:

```
~/src/groovy/curry$ ./log.groovy
Wed Dec 02 06:56:17 GMT 2009: *********** debug:myLog ************
Wed Dec 02 06:56:17 GMT 2009: here is a debug message with myLog
Wed Dec 02 06:56:17 GMT 2009: here is another debug message
Wed Dec 02 06:56:17 GMT 2009:1596: *********** aLog ************
Wed Dec 02 06:56:17 GMT 2009:1596: here is a debug message
Wed Dec 02 06:56:17 GMT 2009:1596: here is a warning message
Wed Dec 02 06:56:17 GMT 2009:1596: here is an error message
Wed Dec 02 06:56:17 GMT 2009:1596: here is another debug message
~/src/groovy/curry$
```

Redirigimos la salida estandard:

```
~/src/groovy/curry$ ./log.groovy > salida
Wed Dec 02 07:01:46 GMT 2009:1882: *********** aLog ************
Wed Dec 02 07:01:46 GMT 2009:1882: here is a debug message
Wed Dec 02 07:01:46 GMT 2009:1882: here is a warning message
Wed Dec 02 07:01:46 GMT 2009:1882: here is an error message
Wed Dec 02 07:01:46 GMT 2009:1882: here is another debug message
~/src/groovy/curry$ cat -n salida
     1  Wed Dec 02 07:01:46 GMT 2009: *********** debug:myLog ************
     2  Wed Dec 02 07:01:46 GMT 2009: here is a debug message with myLog
     3  Wed Dec 02 07:01:46 GMT 2009: here is another debug message
```

Redirigimos stderr:

[1] In the Java Management Extensions API, `MXBean` refers to a managed bean - sometimes simply referred to as an MBean - is a type of JavaBean that implement resources and their instrumentation

```
~/src/groovy/curry$ ./log.groovy 2> salida
Wed Dec 02 07:04:45 GMT 2009: *********** debug:myLog ************
Wed Dec 02 07:04:45 GMT 2009: here is a debug message with myLog
Wed Dec 02 07:04:45 GMT 2009: here is another debug message
~/src/groovy/curry$ cat -n salida
     1  Wed Dec 02 07:04:45 GMT 2009:1961: *********** aLog ************
     2  Wed Dec 02 07:04:45 GMT 2009:1961: here is a debug message
     3  Wed Dec 02 07:04:45 GMT 2009:1961: here is a warning message
     4  Wed Dec 02 07:04:45 GMT 2009:1961: here is an error message
     5  Wed Dec 02 07:04:45 GMT 2009:1961: here is another debug message
```

Juntemos stdout y stderr y los enviamos a un fichero:

```
~/src/groovy/curry$ ./log.groovy 2&>1&> salida
~/src/groovy/curry$ cat -n salida
     1  Wed Dec 02 07:11:12 GMT 2009: *********** debug:myLog ************
     2  Wed Dec 02 07:11:12 GMT 2009: here is a debug message with myLog
     3  Wed Dec 02 07:11:12 GMT 2009: here is another debug message
     4  Wed Dec 02 07:11:12 GMT 2009:2140: *********** aLog ************
     5  Wed Dec 02 07:11:12 GMT 2009:2140: here is a debug message
     6  Wed Dec 02 07:11:12 GMT 2009:2140: here is a warning message
     7  Wed Dec 02 07:11:12 GMT 2009:2140: here is an error message
     8  Wed Dec 02 07:11:12 GMT 2009:2140: here is another debug message
```

6.10. Variables Implícitas en una Clausura

Within a Groovy Closure, several variables are defined that have special meaning:

- `it`

  If you have a Closure that takes a single argument, you may omit the parameter definition of the Closure, like so:

  ```
  def clos = { print it }
  clos( "hi there" )              //prints "hi there"
  ```

- `this`, `owner`, *and* `delegate`

  - `this` *: as in Java,* `this` *refers to the instance of the enclosing class where a Closure is defined*
  - `owner` *: the enclosing object (this or a surrounding Closure)*
  - `delegate` *: by default the same as owner, but changeable for example in a builder or ExpandoMetaClass*

  Example:

  ```
  ~/Lgroovy/closures$ cat -n ThisDelegate.groovy
       1  def clos = new Class1().closure
       2  clos.delegate = this
       3  clos()
       4
       5  class Class1 {
       6    def closure = {
       7      println "Inside closure"
       8      println "this = "+this
       9      println "this.class.name = "+this.class.name
      10      println "delegate = "+delegate
      11      println "delegate.class.name = "+delegate.class.name
  ```

```
    println "owner = "+owner
    println "owner.class.name = "+owner.class.name

    def nestedClos = {
      println " Inside nestedClos"
      println " this = "+this
      println " delegate = "+delegate
      println " owner.class.name = "+owner.class.name
    }

    nestedClos()
  }
}
~/Lgroovy/closures$ groovy ThisDelegate.groovy
Inside closure
this = Class1@57d840cd
this.class.name = Class1
delegate = ThisDelegate@5d04e28b
delegate.class.name = ThisDelegate
owner = Class1@57d840cd
owner.class.name = Class1
    Inside nestedClos
    this = Class1@57d840cd
    delegate = Class1$_closure1@78ff9053
    owner.class.name = Class1$_closure1
```

Capítulo 7

Objetos

Las clases en Groovy son como en Java. Se declaran mediante la palabra reservada `class` y pueden contener los mismos tipos de entes: atributos, constructores, inicializadores y métodos. A su vez los métodos y constructores pueden contener variables locales. Los scripts son diferentes: pueden contener código, definiciones de variables, métodos y clases.

7.1. Declaraciones en Java

Comenzaremos repasando brevemente el manejo de estos entes en Java.
Estas secciones son un extracto de algunas partes del tutorial de Sun ”Classes and Objects”:

7.1.1. Inicializadores en Java

You can often provide an initial value for a field in its declaration. It is not necessary to declare fields at the beginning of the class definition, although this is the most common practice. It is only necessary that they be declared and initialized before they are used. Example:

```
public class BedAndBreakfast {

    public static int capacity = 10;  //initialize to 10

    private boolean full = false;  //initialize to false
}
```

This works well when the initialization value is available and the initialization can be put on one line. However, this form of initialization has limitations because of its simplicity. If initialization requires some logic (for example, error handling or a for loop to fill a complex array), simple assignment is inadequate. Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes static initialization blocks.

Static Initialization Blocks *A* static initialization block *is a normal block of code enclosed in braces,* `{ }`*, and preceded by the* `static` *keyword. Here is an example:*

```
static {

    // whatever code is needed for initialization goes here
}
```

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

There is an alternative to static blocks *—you can write a* private static method*:*

```
class Whatever {
    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() {

        //initialization code goes here
    }
}
```

The advantage of `private static` *methods is that they can be reused later if you need to reinitialize the class variable.*

Initializing Instance Members

Normally, you would put code to initialize an instance variable in a constructor. There are two alternatives to using a constructor to initialize instance variables: initializer blocks *and final methods.*

Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

```
{

    // whatever code is needed for initialization goes here
}
```

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

A final method *cannot be overridden in a subclass. Here is an example of using a final method for initializing an instance variable:*

```
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {

        //initialization code goes here
    }
}
```

This is especially useful if subclasses might want to reuse the initialization method. The method is `final` *because calling non-final methods during instance initialization can cause problems.*

7.2. Declaraciones en Groovy

Groovy, al igual que Java, dispone de *variables locales*, las cuáles residen en el ámbito del método en el que fueron definidas y *atributos* (*fields*) los cuales están asociados con las clases o con las objetos/instancias de esas clases.

7.2.1. Declaración de Variables

Declaraciones Opcionales

Dentro de las clases los atributos y las variables locales deben ser declaradas antes de su uso. Groovy utiliza los mismos *modificadores* que Java para modificar la visibilidad:

- `private`: sólo los miembros de la clase tienen acceso
- `protected`: pueden ser accedidos por las subclases
- `public` (valor por defecto): accesible desde otras clases

El modificador `final` se usa para desactivar nuevas asignaciones y `static` para denotar *variables de clase*. Otro nombre para los atributos no estáticos es *variables de instancia*.

Dentro de las clases, no es obligatorio definir el tipo de una variable pero si que es obligatorio declararlo. La palabra reservada `def` es usada para declarar una variable para especificar que es del tipo genérico Object. Veamos un ejemplo:

```
generaciondecodigos@nereida:~/Lgroovy/learning$ cat -n dec.groovy
class T {
  def static int x = 4
  def static String[] s = [ 1 .. 4 ];
  static Collection<?> c = [1, 2, "a"];
  def static y = x

 public static void main(String[] args) {
    println x
    println y
    println s
    println c
  }
}
```

Obsérvese como Groovy ha realizado las conversiones adecuadas en todos los casos en los que ha sido necesario:

```
generaciondecodigos@nereida:~/Lgroovy/learning$ groovy dec.groovy
4
4
[1..4]
[1, 2, a]
```

Conversiones y Coherciones de Entero a String

La conversión ocurre de la forma que cabría esperar, en el siguiente ejemplo el entero es promocionado a un array de una `String`:

```
generaciondecodigos@nereida:~/Lgroovy/learning$ cat dec2.groovy
class T {
  def static int x = 4
  def static String[] s = [ 1 .. 4 ];

 public static void main(String[] args) {
    s = x
    println s
  }
}
generaciondecodigos@nereida:~/Lgroovy/learning$ groovy dec2.groovy
[4]
```

Conversiones a Números

Sin embargo, esta otra asignación produce una excepción:

```
generaciondecodigos@nereida:~/Lgroovy/learning$ cat dec3.groovy
class T {
  def static int x = 4
  def static String[] s = [ 1 .. 4 ];

 public static void main(String[] args) {
    x = s
    println x
  }
}
generaciondecodigos@nereida:~/Lgroovy/learning$ groovy dec3.groovy
Caught: org.codehaus.groovy.runtime.typehandling.GroovyCastException:
Cannot cast object '[Ljava.lang.String;@125d855f'
with class '[Ljava.lang.String;' to class 'java.lang.Number'
                                    at T.main(dec3.groovy:6)
```

Por tanto, las asignaciones de expresiones a objetos de tipo numérico deben ser de tipo numérico:

```
generaciondecodigos@nereida:~/src/groovy/objects$ cat -n Conform.groovy
     1  int a
     2  try {
     3    a = 'hello'
     4  }
     5  catch(e){
     6    println "Assignments to typed references must conform to the type"
     7  }
```

La ejecución muestra como se produce la ejecución del código que captura la excepción:

```
generaciondecodigos@nereida:~/src/groovy/objects$ groovy Conform.groovy
Assignments to typed references must conform to the type
```

Conversiones a Objetos Genéricos

Si el objeto no esta tipado presenta una conducta polimorfa, aceptandose la asignación:

```
generaciondecodigos@nereida:~/src/groovy/objects$ cat -n Nontyped.groovy
     1  def a
     2  int b = 4
     3  String c = 'hello'
     4  Float d = 4.5
     5
     6  a = b
     7  println a
     8
     9  a = c
    10  println a
    11
    12  a = d
    13  println a
```

Ninguna de las asignaciones a `a` produce un error:

```
generaciondecodigos@nereida:~/src/groovy/objects$ groovy Nontyped.groovy
4
hello
4.5
```

Modificadores y Declaraciones

El siguiente ejemplo ilustra una variedad de declaraciones, incluyendo diversos modificadores:

```
generaciondecodigos@nereida:~/src/groovy/objects$ cat -n Declarations.groovy
class SomeClass {
  public fieldWithModifier
  String typedField
  def unTypedField
  protected field1, field2, field3
  private assignedField = new Date()

  static classField

  public static final String CONSTA = 'a', CONSTB = 'b'

  def someMethod() {
    def localUntypedMethodVar = 1
    int localtypedMethodVar = 1
    def localVarWithoutAssignment, andAnotherOne
  }
}

def localvar = 1
boundVar1 = 1

def someMethod() {
  localMethodVar = 1
  boundVar2 = 1
}
```

La siguiente sesión con `groovysh` muestra las formas de acceso a algunos de los entes definidos en la clase `SomeClass`:

```
generaciondecodigos@nereida:~/Lgroovy/objects$ groovysh
Groovy Shell (1.6.5, JVM: 1.6.0_0)
Type 'help' or '\h' for help.
-----------------------------
groovy:000> import Declarations
===> [import Declarations]
groovy:000> a = new SomeClass()
groovy:000> a.fieldWithModifier = 4
===> 4
groovy:000> SomeClass.fieldWithModifier
ERROR groovy.lang.MissingPropertyException: No such property: fieldWithModifier for class: SomeClass
        at groovysh_evaluate.run (groovysh_evaluate:3)
        ...
groovy:000> SomeClass.classField = 'hola'
===> hola
groovy:000> SomeClass.CONSTA
===> a
groovy:000> SomeClass.CONSTA = 'c'
ERROR groovy.lang.ReadOnlyPropertyException: Cannot set readonly property: CONSTA for class: SomeClass
        at groovysh_evaluate.run (groovysh_evaluate:3)
        ...
```

```
groovy:000> a.assignedField
===> Thu Feb 18 10:21:28 WET 2010
groovy:000> a.typedField = 'hola'
===> hola
groovy:000> a.typedField = [1,2]
===> [1, 2]
groovy:000> a.typedField.class
===> class java.lang.String
groovy:000> (0..<a.typedField.size()).each { print "<${a.typedField[it]}> " }; println ''
<[> <1> <,> < > <2> <]>
===> null
```

Usando JUnit para Comprobar una Excepción de Conversión de Tipos

Obsérvese en el siguiente ejemplo que el valor asignado a la variable de tipo cadena `PI` no está entre comillas.

```
generaciondecodigos@nereida:~/src/groovy/objects$ cat -n Shouldfail.groovy
     1  final String PI = 3.14
     2
     3  println "PI.class.name = ${PI.class.name}"
     4  println "PI.length() = ${PI.length()}"
     5
     6  new GroovyTestCase().shouldFail(org.codehaus.groovy.runtime.typehandling.GroovyCastException) {
     7    Float area = PI
     8  }
     9
    10  try {
    11    Float area = PI
    12  }
    13  catch(e){
    14    println "Assignments to typed references must conform to the type. ${e.class}"
    15  }
    16
```

La asignación deja en `PI` la cadena `"3.14"`. La asignación de`PI` a `area` produce una excepción:

```
generaciondecodigos@nereida:~/src/groovy/objects$ groovy Shouldfail.groovy
PI.class.name = java.lang.String
PI.length() = 4
Assignments to typed references must conform to the type.
class org.codehaus.groovy.runtime.typehandling.GroovyCastException
```

Un objeto `GroovyTestCase` es construido en las líneas:

```
     6  new GroovyTestCase().shouldFail(org.codehaus.groovy.runtime.typehandling.GroovyCastException) {
     7    Float area = PI
     8  }
```

La clase `GroovyTestCase` provee una serie de métodos como

```
protected  String       shouldFail(Class clazz, Closure code)
      Asserts that the given code closure fails when it is evaluated
      and that a particular exception is thrown.
protected  String     shouldFail(Closure code)
      Asserts that the given code closure fails when it is evaluated
```

que simplifican el proceso de pruebas JUnit.

> *JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit based on the design by Kent Beck (the creator of Extreme Programming), originally implemented for Smalltalk as SUnit. JUnit has been ported to many other languages.*

Como `JUnit` forma parte del entorno de ejecución Groovy, es posible escribir pruebas JUnit dentro de las clases Groovy y Java utilizando la sintáxis de Groovy.

7.3. Referenciado Dinámico de Atributos

Además de la sintáxis de punto `obj.atributo` es posible referenciar un atributo utilizando el operador de subíndice `obj[atributo]`. Una ventaja de esta sintáxis es que permite acceder a los atributos de forma dinámica.

En el siguiente ejemplo el nombre del atributo se solicita desde `System.in`. La expresión leída es utilizada para obtener el valor del atributo dinámicamente:

```
generaciondecodigos@nereida:~/Lgroovy/objects$ cat dynamicReferences.groovy
class DynamicAccess {
  public count = 9
  String attr = 'hola'
  def list = (1..10).toList()
}

def stdin = new BufferedReader(new InputStreamReader(System.in))

x = new DynamicAccess();
print "> "
stdin.eachLine{ input ->
  println x[input]
  print "> "
}
```

Para finalizar la entrada pulsamos `CTRL-D`:

```
generaciondecodigos@nereida:~/Lgroovy/objects$ groovy dynamicReferences.groovy
> count
9
> attr
hola
> list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> ^D
```

7.4. Exensión del Operador de Acceso a un Atributo (dot)

Para extender el operador dot (punto) a atributos de manera que pueda aplicarse a cualesquiera cadenas, se deben proveer métodos con nombres `get` y `set` con la siguiente *firma* (signature):

```
Object get(String name)
void   set(String name, Object value)
```

estos métodos no efectan a los atributos existentes.

En este ejemplo usamos esta técnica para acceder a las claves de un mapa del objeto como si fueran atributos:

```
generaciondecodigos@nereida:~/Lgroovy/objects$ cat -n overloadDot.groovy
class PretendFieldCounter {
  public count = 10

  def attr = [:]

  Object get(String name) {
    attr[name]
  }

  void set(String name, Object value) {
    attr[name] = value
  }
}

def p = new PretendFieldCounter()

p.length = 3.5
p.color = 'blue'
println "p.color = ${p.color}"
println "p.length = ${p.length}"
println "p.count = ${p.count}"
```

Véase la salida:

```
generaciondecodigos@nereida:~/Lgroovy/objects$ groovy overloadDot.groovy
p.color = blue
p.length = 3.5
p.count = 10
```

Si queremos interceptar los accesos a un atributo definido `attribute` deberemos escribir métodos `getAttribute` y `setAttribute`. Véase la siguiente variante del ejemplo anterior:

```
generaciondecodigos@nereida:~/Lgroovy/objects$ cat overloadDot2.groovy
class PretendFieldCounter {
  def count = 10

  def attr = [:]

  Object get(String name) {
    attr[name]
  }

  Object getCount() {
     2*count+10
  }


  void set(String name, Object value) {
    attr[name] = value
  }
```

```
}

def p = new PretendFieldCounter()

p.length = 3.5
p.color = 'blue'
println "p.color = ${p.color}"
println "p.length = ${p.length}"
println "p.count = ${p.count}"

generaciondecodigos@nereida:~/Lgroovy/objects$ groovy overloadDot2.groovy
p.color = blue
p.length = 3.5
p.count = 30
```

Al ejecutar este programa obtenemos la siguiente salida:

```
generaciondecodigos@nereida:~/Lgroovy/objects$ groovy overloadDot2.groovy
p.color = blue
p.length = 3.5
p.strange name = 8
p.count = 30
```

7.5. El Método `main`

En general, el modiificador `public` puede omitirse a la hora de declarar un método ya que es el valor que Groovy toma por defecto. En el caso de `main` el array `args` es normalmente declarado de tipo `String[]`. Una consecuencia del mecanismo de dispatch de Groovy es que esto no es estrictamente necesario. Además, como el tipo de retorno no forma parte del mecanismo de dispatch también se puede omitir la declaración del tipo de retorno `void`. Así, la siguiente declaración minimal funciona:

```
~/Lgroovy/objects$ cat -n main.groovy
     1  class SomeClass {
     2    static main(args) {
     3      args.each { println it }
     4    }
     5  }
```

En efecto:

```
~/Lgroovy/objects$ groovy main.groovy one two three
one
two
three
```

7.6. Parámetros con Nombre

Introducción

> *When calling a method you can pass in named parameters. Parameter names and values are separated by a colon (like the Map syntax) though the parameter names are identifiers rather than Strings.*
>
> *Currently this kind of method passing is only implemented for calling methods which take a* `Map` *or for constructing JavaBeans.*

```
~/Lgroovy/objects$ cat -n namedPar.groovy
#!/usr/bin/env groovy
def bean = new Expando(name:"James", location:"London", id:123)
println "name = " + bean.name
println "id   = ${bean.id}"
```

El resultado de la ejecución es:

```
~/Lgroovy/objects$ ./namedPar.groovy
name = James
id   = 123
```

Sintáxis de una LLamada Usando Parámetros con Nombre

> *Named arguments requires the following method signature*
> `def foo(T t, p1, p2, ..., pn)`
> *The type* `T` *can be* `Object`, `Map` *or no explicit type.*
> *The real work is done by the compiler where the method call is defined. A method call*
> `foo(p1:e1, p2:e2, ..., pn:en, q1, q2, ..., qm)`
> *will always be transformed to*
> `foo([p1:e1, p2:e2, ..., pn:en], q1, q2, ..., qm)`
> *If you mix the positions of the* `pi` *and* `qi` *elements, then the compiler will still force the same transformed signature.*

Ejemplo. Cuando se ejecuta este programa:

```
~/Lgroovy/objects$ cat -n namedPar3.groovy
def foo(x,y,z) {[x,y,z]}
println foo(a:1,b:2,3,4)
println foo(a:1,3,b:2,4)
println foo(3,4,a:1,b:2)

println foo(4,3,b:2,a:1)
```

Se obtienen las siguientes salidas:

```
~/Lgroovy/objects$ groovy namedPar3.groovy
[[a:1, b:2], 3, 4]
[[a:1, b:2], 3, 4]
[[a:1, b:2], 3, 4]
[[b:2, a:1], 4, 3]
```

> *In case of* `def foo(T t, p1, p2, ..., pn)` *all named arguments will be in* `t`*, but that also means that you can not make a method call where you access* `pi` *by name.*

Por ejemplo, en el siguiente programa la llamada `foo(x:1,y:2, 8)` es correcta, pero la llamada `tutu(x:1,y:2)` produce una excepción ya que `tutu` espera un argumento no opcional sin nombre que no ha sido especificado en la llamada.

```
~/Lgroovy/objects$ cat -n namedPar4.groovy
def foo(x,y){
  println "<$x> <$y>"
}

def tutu(x,y){
  println "<$x> <$y>"
```

```
}
foo(x:1,y:2, 8)
try {
  tutu(x:1,y:2)
}
catch (e) {
  println e.class
}
```

El resultado de la ejecución es:

```
~/Lgroovy/objects$ groovy namedPar4.groovy
<[x:1, y:2]> <8>
class groovy.lang.MissingMethodException
```

Estructura Típica de un Método que usa Parámetros con Nombres

La estructura habitual de un método que usa parámetros con nombres consiste en una primera fase en la que se comprueban que argumentos con nombre han sido definidos. A los que no figuran se les da un valor por defecto:

```
~/Lgroovy/objects$ cat -n namedPar5.groovy
class SomeClass {
  def TypicalNamedMethod(Map args) {
    args.get('a', 'default for a')
    args.get('b', 'default for b')
    args.get('c', 'default for c')

    return "<$args.a> <$args.b> <$args.c>"
  }
}

def x = new SomeClass()

println x.TypicalNamedMethod(a:1, b:1);
println x.TypicalNamedMethod(a:1, b:1, c:1);
println x.TypicalNamedMethod(c:1, b:1);
```

Este es el resultado de ejecutar el script anterior:

```
~/Lgroovy/objects$ groovy namedPar5.groovy
<1> <1> <default for c>
<1> <1> <1>
<default for a> <1> <1>
```

7.7. Argumentos con Valores por Defecto

Groovy supports also the usage of methods with default values for parameters. Any parameter `T t` can have a default value using `T t=x`, where `x` is the value. The usage of the default value is not bound to a special type. In fact this is a short form to declare an overloaded method.

`def foo(p1, p2, ..., pn, T t=x, q1, q2, ..., qm)`

becomes a method

`def foo(p1, p2, ..., pn, T t, q1, q2, ..., qm)`,

where `t` has no default value and a method

`def foo(p1, p2, ..., pn, q1, q2, ..., qm)`

with this implementation:

```
def foo(p1, p2, ..., pn, q1, q2, ..., qm) {
  // note: this is not working groovy code, just pseudo code!
  foo(p1, p2, ..., pn, x, q1, q2, ..., qm)
}
```

If multiple default values are used, then the parameter with the default value most right will be eliminated like seen here and the resulting method signature will be processed again.

Example:

```
def foo(x=1,y=2) {x+y}
assert foo() == 3
assert foo(2) == 4
assert foo(5,10) == 15
```

7.8. Argumentos Opcionales

La Sintáxis `T... args`

Groovy soporta un número variable de argumentos. Esto es también posible en Java5 utilizando la sintáxis `T... args`. He aqui un ejemplo en Java que calcula el promedio de una lista de `double`:

```
~/Lgroovy/objects$ cat -n VarargsTest.java
public class VarargsTest
{
   // calculate average
   public static double average( double... numbers )
   {
      double total = 0.0; // initialize total

      // calculate total using the enhanced for statement
      for ( double d : numbers )
         total += d;

      return total / numbers.length;
   } // end method average

   public static void main( String args[] )
   {
      double d1 = 10.0;
      double d2 = 20.0;
      double d3 = 30.0;
      double d4 = 40.0;

      System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
         d1, d2, d3, d4 );

      System.out.printf( "Average of d1 and d2 is %.1f\n",
         average( d1, d2 ) );
      System.out.printf( "Average of d1, d2 and d3 is %.1f\n",
         average( d1, d2, d3 ) );
      System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
         average( d1, d2, d3, d4 ) );
   } // end main
} // end class VarargsTest
```

Este es el resultado de una ejecución:

```
~/Lgroovy/objects$ java VarargsTest
d1 = 10,0
d2 = 20,0
d3 = 30,0
d4 = 40,0

Average of d1 and d2 is 15,0
Average of d1, d2 and d3 is 20,0
Average of d1, d2, d3 and d4 is 25,0
```

To use variable arguments in Groovy you have to give a special method signature where the last parameter is an array.

`def foo(p1, ...., pn, T... args)`

Here `foo` supports `n` arguments by default, but also an unspecified number of further arguments exceeding `n`.

Example:

```
~/Lgroovy/objects$ cat optionalArgs1.groovy
def foo(Object... args) {
  args.length
}
println foo()
println foo(1,2)
println foo([1,2],3,4)
~/Lgroovy/objects$ groovy optionalArgs1.groovy
0
2
3
```

In this example we defined a method `foo`, that is able to take any number of arguments, including no arguments at all. `args.length` will then specify the number of arguments. `T...` is the same syntax as used in Java and is internally represented as array where the parameter has a special flag set.

La Sintáxis `T[]`

Groovy allows as alternative `T[]`.

```
~/Lgroovy/objects$ cat optionalArgs2.groovy
def foo(Object[] args) {
  args.length
}
println foo()
println foo(1,2)
println foo([1,2],3,4)
~/Lgroovy/objects$ groovy optionalArgs2.groovy
0
2
3
```

That means any method that has an array as last parameter is seen by Groovy as a method that takes a variable number of arguments.

Argumentos Variables: Excepciones

If in case of `def foo(T[] args)`, `foo` is called with null, then `args` will be `null` and not an array of length 1 with null as only element. If `foo` is called with an array as argument, then `args` will be that array instead of an array of length 1 containing the given array as only element. `T...` will not behave different. This is no special logic we thought of, that is how variable arguments behave in Java too.

```
~/Lgroovy/objects$ cat -n optionalArgs3.groovy
def foo(Object[] args) {
  if (args == null) {
     println "args is null"
     0
  }
  else {
    args.length
  }
}
foo(null)
Object[] a = [1, 2]
println a.class
println foo(a)
b = [1, 2]
println b.class
println foo(b)
~/Lgroovy/objects$ groovy optionalArgs3.groovy
args is null
class [Ljava.lang.Object;
2
class java.util.ArrayList
1
```

Overloaded Methods and Variable Arguments

One more important point to mention is maybe a method overloaded with a method that takes variable arguments. Groovy (and java) will select the most specific method, that means if there is a method taking one argument and the parameter type is an array and there is also another method, that takes also only one argument and the argument is not an array, then the other method is preferred.

Examples:

```
def foo(Object[] args) {1}
def foo (x) {2}
def foo (x,y} {3}

assert foo() == 1
assert foo(1) == 2
assert foo(1,2) == 3
def x = [1,2] as Object[]
assert foo(x) == 1
assert foo(1,1,1) == 1
```

Argumentos Variables y Argumentos con Valores por Defecto

El siguiente ejemplo muestra el comportamiento en un caso en el que se usa un argumento opcional con valor por defecto junto con la sintáxis `T[]`:

```
~/Lgroovy/objects$ cat -n optionalArgs4.groovy
```

```
def foo(c=8, Object[] args) {
  println "c = $c"
  args.length
}
println "num optional args = "+foo(6)
println "num optional args = "+foo(4,5,6)
~/Lgroovy/objects$ groovy optionalArgs4.groovy
c = 6
num optional args = 0
c = 4
num optional args = 2
```

Capítulo 8

MOP: Meta Object Protocol

In object-oriented programming, a metaclass *is a class whose instances are classes. Just as an ordinary class defines the behavior of certain objects, a metaclass defines the behavior of certain classes and their instances. Not all object-oriented programming languages support metaclasses. Among those that do, the extent to which metaclasses can override any given aspect of class behavior varies. Each language has its own metaobject protocol, a set of rules that govern how objects, classes, and metaclasses interact.*

Each groovy object has a metaClass *that is used to manage the dynamic nature of the language. This class intercepts calls to groovy objects to ensure that the appropriate grooviness can be added.*

Everytime you invoke a method on an object, Groovy's dispatch mechanism routes the call through the MetaClass related to the object, so if you can alter the MetaClass in any way you may change the object's behavior at runtime.

8.1. Objetos Groovy y Objetos Java

Los objetos Groovy extienden las capacidades de los objetos Java. El intérprete Groovy maneja de forma diferente las llamadas a métodos para objetos Groovy y para objetos Java.

Existen tres tipos de objetos:

1. POJOs: Plain Old Java Objects
2. POGOs: Plan Old Groovy Objects
3. Interceptadores Groovy. Los interceptadores extienden la clase `GroovyInterceptable`

Los objetos Groovy son definidos en la clase Java `GroovyObject` dentro del paquete `groovy.lang`. Véase el correspondiente fuente Java:

```
lhp@nereida:~/src/groovy/groovyinterpreter/groovy-1.7.2/src/main/groovy/lang$ sed -ne '/^package/,$p' GroovyObject.java | cat -n
package groovy.lang;

/**
 * The interface implemented by all Groovy objects.
 * <p/>
 * Especially handy for using Groovy objects when in the Java world.
 *
 * @author <a href="mailto:james@coredevelopers.net">James Strachan</a>
 * @version $Revision: 9880 $
 */
public interface GroovyObject {

```

```
    /**
     * Invokes the given method.
     *
     * @param name the name of the method to call
     * @param args the arguments to use for the method call
     * @return the result of invoking the method
     */
    Object invokeMethod(String name, Object args);

    /**
     * Retrieves a property value.
     *
     * @param propertyName the name of the property of interest
     * @return the given property
     */
    Object getProperty(String propertyName);

    /**
     * Sets the given property to the new value.
     *
     * @param propertyName the name of the property of interest
     * @param newValue     the new value for the property
     */
    void setProperty(String propertyName, Object newValue);

    /**
     * Returns the metaclass for a given class.
     *
     * @return the metaClass of this instance
     */
    MetaClass getMetaClass();

    /**
     * Allows the MetaClass to be replaced with a derived implementation.
     *
     * @param metaClass the new metaclass
     */
    void setMetaClass(MetaClass metaClass);
}
```

Existe la correspondiente clase `GroovyObject` que permite el acceso a estos métodos y atributos desde Groovy. Los métodos `invokeMethod`, `getProperty`, `setProperty` pueden ser utilizados para crear dinámicamente atributos y métodos.

Los métodos `getMetaClass` y `setMetaClass` facilitan la intercepción e inyección de métodos en POGOs.

Una clase en la JVM es un metaobjeto. Una clase cargada en la JVM no puede ser modificada. Sin embargo si que es posible modificar su `MetaClass` asociada llamando al método `MetaClass()`.

8.2. Algoritmo de Invocación de un Método

Cuando se llama a un objeto Groovy comprueba si se trata de un POJO o de un POGO. La forma en la que se manejan las llamadas es diferente dependiendo de que se trate de un POJO o de un POGO.

POJOs

Para un POJO Groovy carga su metaclase usando `MetaClassRegistry` y delega la invocación a la misma.

Cualesquiera interceptores que se hayan definido en su metaclase tomarán precedencia sobre el método original del POJO.

POGOs

1. Si la clase implementa `GroovyInterceptable` llamar a su `inokeMethod()` y terminar.
2. Si el método existe en la `MetaClass` o en la clase llamar al interceptor o al método original y terminar.
3. Si tiene una propiedad/atributo con ese nombre y el atributo es una clausura llamar a la clausura y terminar.
4. Si existe un método `methodMissing()` llamarlo y terminar.
5. Si existe un método `invokeMethod()` llamarlo y terminar.
6. `throw MissingMethodException()`

8.3. Categorías

Introducción

Groovy proporciona el método `use` el cual permite aumentar la disponibilidad de métodos dinámicos utilizando las llamadas categorías.

> *Categories are a mechanism to extend existing types (even final classes from the JDK or third-party libraries), to add new methods to them.*

Véase este ejemplo tomado de MetaProgramming with Groovy I en http://groovy.dzone.com/::

```
~/Lgroovy/objects$ cat pouncer.groovy
class Pouncer {
   static pounces( Integer self ){
      (0..<self).inject("") { s, n ->
         s += "boing! "
      }
   }
}

use( Pouncer ) {
  println 3.pounces()
}
~/Lgroovy/objects$ groovy pouncer.groovy
boing! boing! boing!
```

> *The magic part happens in*
>
> ```
> use( Pouncer ) {
>   println 3.pounces()
> }
> ```
>
> *where we use a method available to all objects* `use()`*. You can use any class (or a list of classes for that matter) as a parameter for* `use()`*, the point is that all methods on that class are candidates to be categorized methods if they follow these two rules:*

- *any* `public static` *method is a candidate to be a category method*
- *the type of the first parameter is the target type that may have the new methods*

The class is called a Category *if all its methods hold these two properties. The methods are called* category methods*.*

The method `pounces()` *is added to* `Integer` *because it is* `static` *and the first parameter is of type* `Integer`*.*

The `pounces()` *method is available only in the scope of the* `use()` *block, if you try to call it before or after the block you'll get an exception.*

Cadenas con Conversión Automática a Números

Veamos otro ejemplo inspirado en uno tomado del libro de König [1]. Aumentamos las cadenas con una suma y producto similares a los de Perl, de manera que cadenas que contienen números puedan sumarse y multiplicarse directamente, sin necesidad de conversiones explícitas:

```
~/Lgroovy/objects$ cat -n perlPlus.groovy
class PerlPlus {
  static def plus(String self, String operand) {
    try {
      (self.toBigDecimal()+operand.toBigDecimal()).toString()
    }
    catch (NumberFormatException e)  {
      (self << operand).toString()
    }
  }

  static def multiply(String self, String operand) {
    try {
      (self.toBigDecimal()*operand.toBigDecimal()).toString()
    }
    catch (NumberFormatException e)  {
      (self * operand.toInteger()).toString()
    }
  }
}

use(PerlPlus) {
  println ’1’+’2’*’3’
  println ’3’*’2’+’4’
  println ’x’+’4’
  println ’x’*’8’
}
```

Este es el resultado de la ejecución:

```
~/Lgroovy/objects$ groovy perlPlus.groovy
7
10
x4
xxxxxxxx
```

DOMCategory

La categoría DOMCategory nos permite tratar objetos DOM como si fueran arrays y maps de modo que los podemos utilizar en conjunción con el lenguaje de expresiones de caminos Groovy y manejarlos como JavaBeans. Sigue un ejemplo en el que los objetos DOM se manejan como si fueran JavaBeans y son accedidos con expresiones GPath[1]:

```
~/Lgroovy/objects$ cat -n categories1.groovy
import groovy.xml.*

def html = DOMBuilder.newInstance().html {
  head {
    title (class:'mytitle', 'Test')
  }
  body {
    p (class:'mystyle', 'This is a test.')
  }
}

use (groovy.xml.dom.DOMCategory) {
  assert html.head.title.text() == 'Test'
  assert html.body.p.text() == 'This is a test.'
  assert html.find{ it.tagName == 'body' }.tagName == 'body'
  assert html.getElementsByTagName('*').grep{ it.'@class' }.size() == 2
}

try {
  html.head
} catch(MissingPropertyException mpe) {
  println "Categories wear off"
}
```

Véase la sección Groovy Categories en Codehaus.

8.4. Programación Orientada a Aspectos: Interceptando LLamadas con `GroovyInterceptable`

Introducción a la Programación Orientada a Aspectos

> *In computing, aspect-oriented programming (AOP) is a programming paradigm in which secondary or supporting functions are isolated from the main program's business logic. It aims to increase modularity by allowing the separation of cross-cutting concerns, forming a basis for aspect-oriented software development.*
>
> *AOP includes programming techniques and tools that support the modularization of concerns at the level of the source code, while aspect-oriented software development refers to a whole engineering discipline.*
>
> *Aspect-oriented programming entails breaking down a program into distinct parts (so-called* concerns, *cohesive areas of functionality). All programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing abstractions (e.g. procedures, modules, classes, methods) that can be used for implementing, abstracting and composing these concerns. But some concerns defy these forms of implementation and are called* crosscutting concerns *because they çut across”multiple abstractions in a program.*

[1]GPath is a path expression language integrated into Groovy which allows parts of nested structured data to be identified. In this sense, it has similar aims and scope as XPath does for XML

Ejemplo

> *Assume that for whatever reason we want to know the time before and after each call to a method in a class called SimplePOGO. To accomplish this task in Groovy, we simply implement the* `GroovyInterceptable` *interface and the* `invokeMethod()` *like this:*

```
generaciondecodigos@nereida:~/Lgroovy/objects$ cat -n AOP1.groovy
class SimplePOGO implements GroovyInterceptable {
    void simpleMethod1(){
        System.out.println("simpleMethod1() called")
    }

    void simpleMethod2(String param1, Integer param2){
        System.out.println("simpleMethod2(${param1},${param2}) called")
        System.out.println("sleeping...")
        Timer.sleep(2000)
    }

    def invokeMethod(String name, args){
        System.out.println("time before ${name} called: ${new Date()}")

        //Get the method that was originally called.
        def calledMethod = SimplePOGO.metaClass.getMetaMethod(name, args)

        //The "?" operator first checks to see that the "calledMethod" is not
        //null (i.e. it exists).
        calledMethod?.invoke(this, args)

        System.out.println("time after ${name} called: ${new Date()}\n")
    }
}

simplePogo = new SimplePOGO()
simplePogo.simpleMethod1()
simplePogo.simpleMethod2("stringParam", 24)
simplePogo.simpleMethod3(1, 2, 4)
```

- Si una clase Groovy implementa la interface GroovyInterceptable, entonces su método `invokeMethod()` es llamado cada vez que cualquiera de sus métodos es llamado. Tanto si el método existe como si no: en el ejemplo la llamada a `simplePogo.simpleMethod3` será también interceptada.
- El atributo `metaClass` de la clase `SimplePOGO` es de la clase HandleMetaClass y representa la metaclase asociada con `SimplePOGO`.
- La interface HandleMetaClass proporciona el método `getMetaMethod`:

  ```
  public MetaMethod getMetaMethod(String name, Object[] args)
  ```

  > *It retrieves an instance* `MetaMethod` *for the given* `name` *and* `args` *values, using the types of the argument values to establish the chosen MetaMethod. It returns a MetaMethod or* `null` *if it doesn't exist*

- Así pues la llamada

  ```
  def calledMethod = SimplePOGO.metaClass.getMetaMethod(name, args)
  ```

deja en `calledMethod` un objeto MetaMethod que referencia a un método de la clase `SimplePOGO` cuyo nombre es `name` y cuya firma coincide con `args`.

- Una vez que hemos guardado en `calledMethod` el MetaMethod que esta siendo llamado pasamos a invocarlo usando `invoke` cuyo prototipo es:

```
public abstract Object invoke(Object object, Object[] arguments)
```

Ejecución

```
generaciondecodigos@nereida:~/Lgroovy/objects$ groovy AOP1.groovy
time before simpleMethod1 called: Tue Feb 23 16:18:11 WET 2010
simpleMethod1() called
time after simpleMethod1 called: Tue Feb 23 16:18:11 WET 2010

time before simpleMethod2 called: Tue Feb 23 16:18:11 WET 2010
simpleMethod2(stringParam,24) called
sleeping...
time after simpleMethod2 called: Tue Feb 23 16:18:13 WET 2010

time before simpleMethod3 called: Tue Feb 23 16:18:13 WET 2010
time after simpleMethod3 called: Tue Feb 23 16:18:13 WET 2010
```

Inconvenientes

Esta técnica es adecuada siempre que no se de una de estas circunstancias:

- No somos el autor de la clase
- La clase es una clase Java
- Queremos introducir intercepción dinámicamente

Véase también

- Véase Easy AOP with GroovyInterceptable by Justin Spradlin
- Véase Aspect-oriented programming

8.5. Programación Orientada a Aspectos: Interceptando LLamadas Utilizando Metaclases

La mayor diferencia de este código con el anterior es que la casle `SimplePOGO` no implementa GroovyInterceptable y no dispone de un método `invokeMethod()`.

```
generaciondecodigos@nereida:~/Lgroovy/objects$ cat -n AOP2.groovy
class SimplePOGO {
    void simpleMethod1(){
        System.out.println("simpleMethod1() called")
    }

    void simpleMethod2(String param1, Integer param2){
        System.out.println("simpleMethod2(${param1},${param2}) called")
        System.out.println("sleeping...")
        Timer.sleep(2000)
    }
```

```

}

SimplePOGO.metaClass.invokeMethod = { String name, args ->
    System.out.println("time before ${name} called: ${new Date()}")

    //Get the method that was originally called.
    def calledMethod = SimplePOGO.metaClass.getMetaMethod(name, args)

    calledMethod?.invoke(delegate, args)

    System.out.println("time after ${name} called: ${new Date()}\n")
}

simplePogo = new SimplePOGO()
simplePogo.simpleMethod1()
simplePogo.simpleMethod2("stringParam", 24)
simplePogo.simpleMethod3(1, 2, 4)
```

En la llamada

```
calledMethod?.invoke(delegate, args)
```

debemos referirnos a `delegate` ya que `this` hace alusión al objeto clausura. Dentro de la clausura `delegate` se refiere al objeto cuyos métodos estan siendo interceptados.

```
generaciondecodigos@nereida:~/Lgroovy/objects$ groovy AOP2.groovy
time before simpleMethod1 called: Tue Feb 23 16:47:40 WET 2010
simpleMethod1() called
time after simpleMethod1 called: Tue Feb 23 16:47:40 WET 2010

time before simpleMethod2 called: Tue Feb 23 16:47:40 WET 2010
simpleMethod2(stringParam,24) called
sleeping...
time after simpleMethod2 called: Tue Feb 23 16:47:42 WET 2010

time before simpleMethod3 called: Tue Feb 23 16:47:42 WET 2010
time after simpleMethod3 called: Tue Feb 23 16:47:42 WET 2010
```

8.6. Intercepción de Clases Java

Una de las ventajas de utilizar intercepción basada en Metaclases es que puede ser aplicada a clases Java. Veamos un ejemplo:

```
generaciondecodigos@nereida:~/Lgroovy/mop$ cat -n InterceptingInteger.groovy
     1  Integer.metaClass.invokeMethod = { String name, args ->
     2    println "Call to ${name}() intercepted on $delegate ..."
     3    def validMethod = Integer.metaClass.getMetaMethod(name, args)
     4
     5    println("Running pre-filter ...")
     6      def result = validMethod?.invoke(delegate,args)
     7    println("Running post-filter ...")
     8
     9    result
    10  }
```

```

println 5.floatValue()
5.doesNotExists()
```

La ejecución de este programa produce la siguiente salida:

```
generaciondecodigos@nereida:~/Lgroovy/mop$ groovy InterceptingInteger.groovy
Call to floatValue() intercepted on 5 ...
Running pre-filter ...
Running post-filter ...
5.0
Call to doesNotExists() intercepted on 5 ...
Running pre-filter ...
Running post-filter ...
```

El atributo `metaClass` de la clase `Integer` es de la clase HandleMetaClass y representa la metaclase asociada con `Integer`:

```
groovy:000> Integer.metaClass.getClass().name
===> org.codehaus.groovy.runtime.HandleMetaClass
```

Esta clase dispone del atributo `protected` MetaClass `delegate` y de varios métodos, entre los que está

```
public MetaMethod getMetaMethod(String name, Object[] args)
```

It retrieves an instance MetaMethod for the given `name` and `args` values, using the types of the argument values to establish the chosen `MetaMethod`. It returns a MetaMethod or `null` if it doesn't exist. The class MetaMethod represents a Method on a Java object: a little like Java class `Method` except without using reflection to invoke the method. CacheMethod.

The MetaMethod object `validMethod` has a number of methods. Among them

```
public abstract Object invoke(Object object, Object[] arguments)
```

8.7. Creación Dinámica de Métodos

```
~/Lgroovy/objects$ cat -n dynamicmethodnamecreation.groovy
class Person {
   String name = "Fred"
}

def methodName = "to-Upper"

Person.metaClass."${methodName}" = {->
   println "this = ${this.class.name}"
   println "owner = ${owner.class.name}"
   println "delegate = ${delegate.class.name}"
   delegate.name.toUpperCase()
}

def p = new Person()

println p.name

println p.'to-Upper'()

```

```
println p.name
~/Lgroovy/objects$ groovy dynamicmethodnamecreation.groovy
Fred
this = dynamicmethodnamecreation
owner = dynamicmethodnamecreation
delegate = Person
FRED
Fred
```

Veamos otro ejemplo tomado de http://freeopenidea.blogspot.com/2009/02/dynamic-method-creation.html:

```
~/Lgroovy/objects$ cat -n DynamicMethodCreation.groovy
class CoolObject {
    public def invokeMethod(String name, def args) {
        if (name.startsWith("say")) {
            def words = []
            def currentWord = ""

            name.substring(3).each { letter ->
                if (letter.equals(letter.toUpperCase())) {
                    words.add currentWord
                    currentWord = ""
                }
                currentWord += letter
            }
            words.add currentWord

            print "${args[0]} you are"
            words.collect {
                it == "Dash" ? "-" : it
            }.each {
                print "$it "
            }
            println ""

            return args[0]
        }

        super.invokeMethod(name, args)
    }
}

CoolObject coolObject = new CoolObject()

println coolObject.saySomeoneCool("John")
println coolObject.saySomeoneReallyDashReallyCool("Mary")
println coolObject.toString()
~/Lgroovy/objects$ groovy DynamicMethodCreation.groovy
John you are Someone Cool
John
Mary you are Someone Really - Really Cool
Mary
CoolObject@6c913dc1
```

8.8. Metaprogramación con Convenio Basado en el Nombre del Paquete

> *The idea is that any* `package.class` *can have a custom meta class loaded at startup time by placing it into a well known package with a well known name.*
>
> `groovy.runtime.metaclass.[YOURPACKAGE].[YOURCLASS]MetaClass`
>
> *So your class* `Foo` *in package* `"bar"` *could have a custom meta class* `FooMetaClass` *in package* `"groovy.runtime.metaclass.bar"`.

```
generaciondecodigos@nereida:~/Lgroovy/objects$ tree
.
|-- DelegatingMetaClassPackageImpliedTest.groovy
`-- groovy
    `-- runtime
        `-- metaclass
            `-- java
                `-- lang
                    |-- StringMetaClass.class
                    `-- StringMetaClass.groovy


generaciondecodigos@nereida:~/Lgroovy/objects$ cat -n groovy/runtime/metaclass/java/lang/StringMetaClass.groovy
package groovy.runtime.metaclass.java.lang

class StringMetaClass extends groovy.lang.DelegatingMetaClass
{
    StringMetaClass(MetaClass delegate)
    {
        super(delegate);
    }

    public Object invokeMethod(Object a_object, String a_methodName, Object[] a_arguments)
    {
        return "changed ${super.invokeMethod(a_object, a_methodName, a_arguments)}"
    }
}

generaciondecodigos@nereida:~/Lgroovy/objects$ cat -n DelegatingMetaClassPackageImpliedTest.groovy
class DelegatingMetaClassPackageImpliedTest
{
    static void main(String[] args) {
        println "hello world".toString()
    }
}

generaciondecodigos@nereida:~/Lgroovy/objects$ groovyc groovy/runtime/metaclass/java/lang/StringMetaClass.groovy
generaciondecodigos@nereida:~/Lgroovy/objects$ groovy DelegatingMetaClassPackageImpliedTest.groovy
changed hello world
```

Véase También

- Metaprogramming in Groovy by Pecha Kucha
- MetaProgramming with Groovy I:

Índice alfabético

Bibliografía

[1] Dierk König. *Groovy in Action*. Manning, New York, 2007.

[2] Kenneth A. Barclay and W. J. Savage. Groovy programming, 2007.