Aspectos avanzados en Groovy

Índice

En la sesión anterior hacíamos un repaso completo sobre los tipos de tipos de datos simples y algunas características. Posteriormente hablábamos sobre colecciones y comentábamos las excelencias de los rangos, una tipo de datos propio de Groovy, además de los anteriormente conocidos listas y mapas.

Por último, echábamos un vistazo a las estructuras de control que podíamos utilizar en Groovy y a la combinación de éstas con los nuevos tipos de datos.

En la sesión 3 de este curso, veremos a fondo un nuevo concepto llamado *closure*. Ya hemos comentado algo sobre ellos en la sesión anterior, pero en esta veremos todas sus características.

Una vez vistos los closures, pasaremos a hablar sobre las características de Groovy como lenguaje orientado a objetos.

1. Closures

Aunque en apartados anteriores ya hayamos visto algunos ejemplos de closures, es conveniente dedicarle más tiempo a su explicación, ya que son una de las partes más importantes del lenguaje Groovy y más utilizados en Grails, y al mismo tiempo, puede ser un concepto difícil de entender, ya que no aparece en otros lenguajes de programación. Así que volveremos a ver lo que son los closures, como se declaran y como pueden ser posteriormente referenciados.

Más adelante, pasaremos a ver otros métodos disponibles en los closures y el ámbito de aplicación de los mismos, es decir, quien puede acceder a los mismos. Finalmente, definiremos varias tareas comunes que pueden ser realizadas con los closures, que hasta ahora se hacen de otras formas.

1.1. Definición de closure

Un *closure* es un trozo de código empaquetado como un objeto y definido entre llaves. Actúa como un método, al cual se le pueden pasar parámetros y pueden devolver valores. Es un objeto normal y corriente al cual se pasa una referencia de la misma forma que se le pasa a cualquier otro objeto.

Posiblemente estés pensando, que de momento los closures no te aportan nada que no puedas hacer nada con cualquier otro lenguaje de programación y posiblemente sea cierto. Sin embargo, los closures nos aportan agilidad a la hora de programar, que es lo que en principio buscamos utilizando un lenguaje como Groovy.

1.2. Declarando closures

Como comentábamos anteriormente, los closures son bloques de código encerrado entre

llaves {}. Pues en primer lugar, vamos a definir un closure para imprimir nuestro nombre.

```
def nombre = 'Juan'
def imprimeNombre = { println "Mi nombre es $nombre"}

imprimeNombre()

nombre = "Yolanda"
imprimeNombre()
```

Te habrás dado cuenta de que el closure que acabamos de crear no está parametrizado, con lo que si se cambiara el nombre de nuestra variable, el closure no se ejecutaría correctamente. Para definir parámetros en nuestros closures, podemos hacerlo al inicio del mismo introduciendo el nombre de nuestros parámetros (separados por comas si hay más de uno) seguido de los caracteres ->. El ejemplo anterior parametrizado quedaría así:

```
def imprimeNombre = { nombre -> println "Mi nombre es ${nombre}"}

imprimeNombre("Juan")
imprimeNombre "Yolanda" //Los paréntesis son opcionales

//Con múltiples parámetros
def quintetoInicial = { base, escolta, alero, alapivot, pivot ->
println "Quinteto inicial compuesto por: $base, $escolta, $alero,
$alapivot y $pivot"}

quintetoInicial "Calderón", "Navarro", "Jiménez", "Garbajosa", "Pau
Gasol"
```

En aquellos closures que sólo tienen un parámetro, es posible obviar su declaración al inicio del closure, puesto que Groovy pone a nuestra disposición la variable *it*. El siguiente ejemplo es idéntico al closure *imprimeNombre* anterior, pero sin declarar sus parámetros.

```
def imprimeNombre = { println "Mi nombre es $it" }

imprimeNombre("Juan")
imprimeNombre "Yolanda"
```

Por último, existe otra forma de declarar un closure y es aprovechando un método ya existente. Con el operador *referencia &* podemos declarar un closure a partir de un método de una clase ya creada. El siguiente ejemplo, tenemos la clase MetodoClosureEjemplo, en la que existe un método para comprobar si la longitud de la cadena pasada por parámetro es superior a un límite. A partir de este método, crearemos un closure sobre dos instancias de esta clase creadas con límites diferentes. Se puede comprobar como ejecutando el mismo método, obtenemos resultado diferentes.

```
class MetodoClosureEjemplo {
    int limite

    MetodoClosureEjemplo (int limite){
        this.limite = limite
    }
```

```
    boolean validar (String valor){
        return valor.length() <= limite
    }
}

MetodoClosureEjemplo primero = new MetodoClosureEjemplo(8)
MetodoClosureEjemplo segundo = new MetodoClosureEjemplo(5)

Closure primerClosure = primero.&validar

def palabras = ["cadena larga", "mediana", "corta"]

assert "mediana" == palabras.find(primerClosure)
assert "corta" == palabras.find(segundo.&validar)
```

Con la variable *primero* estamos creando una instancia de la clase *MetodoClosureEjemplo* que validará aquellas palabras que tengan como mucho 8 caracteres, mientras que la variable *segundo* validará aquellas palabras con 5 caracteres o menos. Posteriormente, el closure *primerClosure* devolverá la primera palabra encontrada con 8 o menos caracteres y de la lista de palabras coincidiría con la palabra "mediana". En el segundo closure, el que valida la palabras de 5 o menos caracteres, la palabra devuelta sería "corta".

Otra característica interesante de los closures se refiere a la posibilidad de ejecutar diferentes métodos en función de los parámetros pasados y se conoce como *multimétodo*. La idea es crear una clase que sobrecarga un determinado método y posteriormente crear un closure a partir de ese método sobrecargado.

```
class MultimetodoClosureEjemplo{

    int metodoSobrecargado(String cadena){
        return cadena.length()
    }

    int metodoSobrecargado(List lista){
        return lista.size()
    }

    int metodoSobrecargado(int x, int y){
        return x * y
    }
}

MultimetodoClosureEjemplo instancia = new
MultimetodoClosureEjemplo()
Closure multiclosure = instancia.&metodoSobrecargado

assert 21 == multiclosure("una cadena cualquiera")
assert 4 == multiclosure(['una','lista','de','valores'])
assert 21 == multiclosure(7, 3)
```

1.3. Los closures como objetos

Anteriormente comentábamos que los closures son objetos y que como tales, pueden ser pasados como parámetros a funciones. Un ejemplo de este caso que ya hemos visto con anterioridad es el método `each()` de las listas, al cual se le puede pasar un closure para realizar una determinada operación sobre cada elemento de la lista.

```
def quintetoInicial = ["Calderón", "Navarro", "Jiménez",
"Garbajosa", "Pau Gasol"]

salida = ''
quintetoInicial.each {
    salida += it +', '
}
assert salida == 'Calderón, Navarro, Jiménez, Garbajosa, Pau Gasol,
'
```

1.4. Usos de los closures

Ahora que ya sabemos como declarar los closures, vamos a ver como utilizarlos y como podemos invocarlos. Si tenemos definido un *Closure x* y queremos llamarlo podemos hacerlo de dos formas:

- x.call()
- x()

```
def suma = { x, y ->
        x + y
}
assert 10 == suma(7,3)
assert 13 == suma.call(7,6)
```

A continuación, veremos un ejemplo sobre como pasar un closure como parámetro a un método. El ejemplo nos permitirá tener un campo de pruebas para comprobar que código es más rápido.

```
def campodepruebas(repeticiones, Closure proceso){
   inicio = System.currentTimeMillis()
   repeticiones.times{proceso(it)}
   fin = System.currentTimeMillis()
   return fin - inicio
}

lento = campodepruebas(10000) { (int) it / 2 }
rapido = campodepruebas(10000) { it.intdiv(2) }

//El método lento es al menos 10 más lento que el rápido
assert rapido * 10 < lento
```

Cuando ejecutamos `campodepruebas(10000)` le estamos pasando el primer parámetro, el que indica el número de repeticiones del código a ejecutar, mientras que el código encerrado entre llaves se corresponde con el *Closure* pasado como segundo parámetro. Cuando definimos un método que recibe como parámetro un closure, es obligatorio que éste sea definido el último parámetro del método.

Hasta el momento, siempre que hemos creado un closure con parámetros, le hemos pasado tantas variables como parámetros tenía el closure. Sin embargo, al igual que en los métodos, es posible establecer un valor por defecto para los parámetros de un closure de la siguiente forma:

```
def suma = { x, y=3 ->
        suma = x + y
}
assert 7 == suma(4,3)
assert 7 == suma(4)
```

1.5. Más métodos de los closures

La clase `groovy.lang.Closure` es una clase como cualquier otra, aunque es cierto que con una potencia increíble. Hasta ahora, sólo hemos visto la existencia del método `call()`, pero existen muchos más, de los que vamos a ver los más importantes.

Cuando en su momento hablamos sobre los mapas, vimos que podíamos pasar al método `each()`, tanto los elementos *clave* y *valor* como un valor *item* que contenía ambos elementos (clave y valor). Para saber como actuar, el closure debe saber el número de parámetros pasados y para ello, se dispone del método `getParameterTypes()`.

```
def llamador (Closure closure){
    closure.getParameterTypes().size()
}

assert llamador { uno -> } == 1
assert llamador { uno, dos -> } == 2
```

Existe una técnica en programación llamada *currying* en honor a su creador Haskell Brooks Curry, que consiste en transformar una función con múltiples parámetros en otra con menos parámetros. Un ejemplo puede ser la función que suma dos valores. Si tenemos esta función con dos parámetros, y queremos crear otra que acepte un sólo parámetro, está claro que debe ser perdiendo el segundo parámetro, lo que conlleva a sumar siempre el mismo valor en esta nueva función. El método *curry()* devuelve un clon de la función principal, eliminando uno o más parámetros de la misma.

```
def suma = { x, y -> x + y }
def sumaUno = suma.curry(1)

assert suma(4,3) == 7
assert sumaUno(5) == 6
```

El nuevo closure `sumaUno` siempre toma como segundo parámetro el valor 1.

1.6. Valores devueltos en los closures

Los *closures* tienen dos formas de devolver valores:

- *De forma implícita*. El resultado de la última expresión evaluada por el closure, es lo

que éste devuelve. Esto lo que hemos hecho hasta ahora.
- *De forma explícita*. La palabra reservada *return* también nos servirá en los closures para devolver valores

En el siguiente código de ejemplo, ambos closures tienen el mismo efecto, que es la duplicación de los valores de la lista.

```
[1,2,3].collect { it * 2 }
[1,2,3].collect { return it * 2 }
```

Si queremos salir de un closure de forma prematura, también podemos hacer uso del return. Por ejemplo, si en el ejemplo anterior sólo queremos duplicar aquellos valores impares, deberíamos tener algo así.

```
[1,2,3].collect {
        if (it%2==1) return it * 2
        return it
}
```

2. Groovy como lenguaje orientado a objetos

Un concepto erróneo que se suele decir de los lenguajes scripts, debido a su dejadez con el tipado de datos y determinadas estructuras de control, es que son lenguajes destinados más a los hackers que a los programadores serios. Esta reputación viene de las primeras versiones del lenguaje Perl, donde la falta de encapsulación y otras características típicas del modelo orientado a objetos, provocaba un mala gestión del código, con frecuentes trozos de código duplicados e indescifrables fallos de programación.

Sin embargo, este panorama ha cambiado drásticamente en los últimos años, ya que lenguajes como el mismo Perl, Python y más recientemente, Ruby, han añadido características del modelo orientado a objetos, que los hacen incluso más productivos que lenguajes como Java o C++. Groovy también se ha subido al carro de estos lenguajes ofreciendo características similares, con lo que ha pasado de ser un lenguaje de script basado en Java, a ser un lenguaje que nos ofrece nuevas características del modelo orientado a objetos.

Hasta ahora hemos visto que Groovy nos ofrece tipos de datos referencia donde Java simplemente nos ofrecía tipos de datos simples, tenemos rangos y closures, y muchas estructuras de control para trabajar de forma muy ágil y sencilla con colecciones de objetos. Pero esto es sólo la punta del iceberg, y a partir de ahora veremos otras características, que hacen de Groovy un lenguaje con mucho futuro. Empezaremos repasando las clases y los scripts en Groovy, seguiremos por la organización de nuestras clases y terminaremos viendo características avanzadas del modelo orientado a objetos en Groovy.

2.1. Clases y scripts

La definición de clases en Groovy es prácticamente idéntica a como se hace en Java. Las clases se declaran utilizando la palabra reservada *class* y una clase puede tener *campos*, *constructores*, *inicializadores* y *métodos*. Los métodos y los constructores pueden utilizar *variables locales*. Por otro lado tenemos los *scripts* que puede contener la definición de variables y métodos, así como la declaración de clases.

La **declaración de variables** debe realizarse antes de que se utilicen. Declarar una variable supone indicar un nombre a la misma y un tipo, aunque en Groovy esto es opcional. Los scripts pueden utilizar variables que no han sido declaradas previamente.

Groovy utiliza los mismos modificadores que Java, que son: *private*, *protected* y *public* para modificar la visibilidad de las variables; *final* para evitar la modificación de variables; y *static* para la declaración de las variables de la clase.

La definición del tipo de la variable es opcional en Groovy y cuando no se especifica, debemos introducir previamente al nombre de la variable, la palabra reserva *def*. Por último y aunque pueda resultar obvio, en Groovy es imposible asignar valores a variables que no coincidan en el tipo. Por ejemplo, un valor numérico no puede ser asignado a una variable definida de tipo *String*. Como vimos anteriormente, Groovy se encarga de hacer el llamado *autoboxing* siempre y cuando sea posible.

Otro aspecto interesante de Groovy es la asignación de valores a las propiedades de las clases. Si hemos definido un campo en una clase en Groovy, podemos acceder al valor del mismo de la forma habitual `objeto.campo` o bien `objeto['campo']`. Esto nos permite una mayor facilidad para acceder a los campos de las clases de forma dinámica, tal y como se hace en el siguiente fragmento de código.

```
class miClase {
    public campo1, campo2, campo3, campo4 = 0
}

def miobjeto = new miClase()

miobjeto.campo1 = 2
assert miobjeto.campo1 == 2

miobjeto['campo2'] = 3
assert miobjeto.campo2 == 3

def salida = ''
for(i=1;i<=4;i++)
    miobjeto['campo'+i] = i - 1

assert miobjeto.campo1 == 0
assert miobjeto['campo2'] == 1
assert miobjeto.campo3 == 2
assert miobjeto['campo4'] == 3
```

La **declaración de los métodos** sigue los mismos criterios que acabamos de ver con las variables. Se pueden utilizar los modificadores Java, es opcional devolver algo con la sentencia *return* y si no se utilizan modificadores ni queremos especificar el tipo de dato a

devolver, debemos utilizar la palabra reservada *def* para declarar nuestros métodos. Por defecto, la visibilidad de los métodos en Groovy es *public*.

Veamos una clase ejemplo y con ella, algunas diferencias con la misma clase en Java.

```
class MiClase{

       static main(args){
           def algo = new MiClase()
           algo.metodoPublicoVacio()
           assert "hola" == algo.metodoNoTipado()
           assert 'adios' == algo.metodoTipado()
           metodoCombinado()
       }

       void metodoPublicoVacio(){
           ;
       }

       def metodoNoTipado(){
           return 'hola'
       }

       String metodoTipado(){
           return 'adios'
       }

       protected static final void metodoCombinado(){

       }
}
```

En la primera sesión comentábamos que el método *main* típico de Java y C, ya no era necesario en Groovy, puesto que podíamos ejecutar nuestro código sin necesidad de incluirlo en dicho método. Sin embargo, si queremos introducir parámetros a nuestro programa, tendremos que utilizarlo, tal y como aparece en el ejemplo. No obstante, este método main es algo diferente al de Java, puesto que no es necesario indicarle que el método es público, ya que, por defecto y salvo que se diga que lo contrario, todo método en Groovy es *public*. La segunda diferencia con Java es que los argumentos debían ser del tipo `String[]` mientras que en Groovy simplemente es un objeto y no es necesario especificarle el tipo. Puesto que en la función main no se va a devolver nada, es posible obviar la etiqueta *void* en la declaración del método. Resumiendo, mientras que en Java tendríamos esto `public static void main (String[] args)`, en Groovy quedaría algo así `static main (args)`.

Groovy nos ahorra también bastante trabajo en cuanto a la comprobación de errores. En este sentido, cuando intentamos llamar a un método o un objeto cuya referencia es *null*, obtendremos una excepción del tipo `NullPointerException`, lo cual es muy útil para comprobar que nuestro código funciona tal y como debe funcionar. Veamos un ejemplo:

```
def mapa = [a:[b:[c:1]]]

assert mapa.a.b.c == 1
```

```
//Protección con cortocircuito
if (mapa && mapa.a && mapa.a.x){
    assert mapa.a.x.c == null
}

//Protección con un bloque try/catch
try{
    assert mapa.a.x.c == null
} catch (NullPointerException npe){}

//Protección con el operador ?.
assert mapa?.a?.x?.c == null
```

En el ejemplo, estamos intentando acceder a una propiedad que no existe como es `mapa.a.x`. Antes de acceder a dicha propiedad, protegemos el acceso para comprobar que no sea `null` y en caso de que no lo sea, acceder a su valor. Aparecen tres tipos de protección de este tipo de errores. La comprobación en cortocircuito con un bloque `if`, es decir, que cuando se detecte una condición de la expresión que sea falsa, nos salimos del if. En segundo lugar, intentamos proteger el acceso erróneo con un bloque `try/catch`. Y por último, con el operador *?.*, el cual no es en sí una comprobación y es la opción que menos código emplea.

Por último, es necesario mencionar algo sobre los constructores en Groovy. Los constructores tienen la función de inicializar los objetos de una determinada clase y en caso de que no se especifique ningún constructor para la clase, éstos son creados directamente por el compilador. Nada que no se haga ya en Java. Sin embargo, era extraño que la gente de Groovy no hiciera algo más y así es, hay más.

Existen dos formas de llamar a los constructores de las clases creadas en Groovy. Por un lado, el método tradicional pasando parámetros de forma posicional, donde el primer parámetro significa una cosa, el segundo otra y así sucesivamente. Y por otro lado, tenemos también la posibilidad de pasar los parámetros a los constructores utilizando directamente los nombres de los campos. Esta característica surge debido a que pasar parámetros según su posición tiene el inconveniente de aquellos constructores que tienen demasiados campos y no es sencillo recordar el orden de los mismos.

Cuando creamos un objeto de una clase mediante su constructor, podemos pasarle los parámetros de tres formas diferentes en Groovy:

- Mediante la forma tradicional, pasando parámetros en orden
- Mediante la palabra reservada *as* y una lista de parámetros
- Mediante una lista de parámetros

Veamos un ejemplo:

```
class Libro{
    String titulo, autor

    Libro(titulo, autor){
        this.titulo = titulo
```

```
        this.autor = autor
    }
}

//Forma tradicional
def primero = new Libro('Groovy in action', 'Dierk König')

//Mediante la palabra reservada as y una lista de parámetros
def segundo = ['Groovy in action','Dierk König'] as Libro

//Mediante una lista de parámetros
Libro tercero = ['Groovy in action','Dierk König']

assert primero.getTitulo() == 'Groovy in action'
assert segundo.getAutor() == 'Dierk König'
assert tercero.titulo == 'Groovy in action'
```

Además del motivo de tener que recordar el orden de los parámetros pasados al constructor de la clase, otra razón para utilizar este tipo de llamadas a los constructores es que podemos ahorrarnos la creación de varios constructores. Imagina el caso en que tengamos dos campos de la clase, y ambos sean opcionales. Esto supone crear cuatro constructores: sin parámetros, con uno de los campos, con el otro campo y con los dos campos. Si utilizamos este tipo de llamadas a los constructores nos ahorraremos la creación de estos cuatro constructores. Veamos el mismo caso que antes con la clase *Libro*. En el ejemplo, dejamos que Groovy cree por nosotros los constructores.

```
class Libro {
        String titulo, autor
}

def primero = new Libro()
def segundo = new Libro(titulo: 'Groovy in action')
def tercero = new Libro(autor: 'Dierk König')
def cuarto = new Libro(titulo: 'Groovy in action', autor: 'Dierk
König')

assert primero.getTitulo() == null
assert segundo.titulo == 'Groovy in action'
assert tercero.getAutor() == 'Dierk König'
assert cuarto.autor == 'Dierk König'
```

2.2. Organizando nuestras clases y scripts

En este apartado, vamos a ver como podemos organizar nuestras clases y ficheros de la aplicación, y la relación entre ellos. También veremos la utilización de paquetes (*packages*) y el alias de tipos. Comencemos por la relación entre las clases y los ficheros fuentes.

La relación entre los ficheros fuente y las clases en Groovy no es tan estricta como en Java y los ficheros fuente en Groovy pueden contener tantas definiciones de clases como queramos, siguiente una serie de reglas:

- Si el fichero .groovy no tiene la declaración de ninguna clase, éste se trata como si

fuera un *script* y automáticamente se genera una clase de tipo *Script* con el mismo nombre que el fichero .groovy.
- Si el fichero .groovy contiene una sola clase definida con el mismo nombre que el fichero, la relación es la misma que en Java, es decir, un fichero .class por cada fichero .groovy
- Si el fichero .groovy contiene más de una clase definida, el compilador de groovy, *groovyc*, creará tantos ficheros .class como sean necesarios para cada una de las clases definidas en el fichero .groovy. Si quisiéramos llamar a nuestros scripts directamente a través de la línea de comando, deberíamos añadir el método *main* a la primera de las clases definidas en el fichero .groovy.
- Un fichero Groovy puede mezclar la definición de clases con el código script. En este caso, el código script se convierte en la clase principal a ejecutar, con lo que no se puede declarar una clase con el mismo nombre que el fichero fuente.

Otro aspecto importante para la organización de los ficheros de nuestros proyectos en Groovy es la **organización en paquetes**. En este sentido, Groovy sigue el mismo convenio que Java y su organización jerárquica. De esta forma, la estructura de paquetes se corresponde con los ficheros *.class* de la estructura de directorios.

Sin embargo, como ya se ha comentado en la primera sesión de este curso, no es necesario compilar nuestros archivos .groovy, con lo que se añade un problema a la hora de ejecutar nuestros archivo. Si no existen los archivos compilados *.class*, ¿dónde los va a buscar? Groovy soluciona este *problema*, buscando también en los archivos .groovy aquellas clases necesarias. Así, que el *classpath* no sólo nos va a servir para indicar los directorios donde debe buscar los archivos .class, sino también para decirle donde pueden estar los archivos .groovy, en caso de que los archivos .class no estén. En el caso de que Groovy encuentre en el classpath, tanto ambos archivos, .groovy y .class, se quedará con el más reciente, y así se evitarán los problemas producidos porque se nos haya olvidado compilar nuestro archivo .groovy.

Al igual que en Java, las clases Groovy deben especificar su pertenencia a un paquete antes de su declaración. El siguiente fragmento de código muestra un ejemplo de un archivo con dos clases definidas que forman parte de un mismo paquete.

```
package negocio

class Cliente {
        String nombre, producto
        Direccion direccion = new Direccion()
}

class Direccion {
        String calle, ciudad, provincia, pais, codigopostal
}
```

Para poder utilizar las clases declaradas *Cliente* y *Direccion*, debemos *importar* el paquete correspondiente *negocio*, tal y como aparece en el siguiente ejemplo.

```
import negocio.*
```

```
def clienteua = new Cliente()
clienteua.nombre = 'Universidad de Alicante'
clienteua.producto = 'Pîzarras digitales'

assert clienteua.getNombre == 'Universidad de Alicante'
```

El último aspecto importante a comentar en cuanto a la organización de las clases y los archivos de nuestras aplicaciones, se refiere a la posibilidad de establecer alias a los paquetes importados. Imaginad el caso de tener dos paquetes procedentes de terceras partes que contengan una clase con el mismo nombre. En Groovy podemos solucionar este conflicto de nomenclatura utilizando los alias. Veamos como:

```
import agenteexterno1.OtraClase as OtraClase1
import agenteexterno2.OtraClase as OtraClase2

def otraClase1 = new OtraClase1()
def otraClase2 = new OtraClase2()
```

2.3. Características avanzadas del modelo orientado a objetos

Ahora que ya tenemos unos conocimientos básicos de lo que Groovy nos permite en cuanto al modelo orientado a objetos, pasemos a ver conceptos algo más avanzados de los vistos hasta ahora, como son la *herencia*, los *interfaces* y los *multimétodos*.

Cuando hablamos de **herencia** en el modelo orientado a objetos, nos referimos a la posibilidad de añadir campos y métodos a una clase a partir de una clase base. Groovy permite la herencia en los mismos términos que lo hace Java. Es más, una clase Groovy puede extender una clase Java y viceversa.

Groovy también soporta el modelo de *interfaces* de Java. En Java, un interface es una clase especial en la que todos sus métodos son abstractos y públicos sin necesidad de declararlos. Sin embargo, estos métodos no están implementados en la clase interface, sino que esa labor se deja para la clase que extienda esa interface. Los interfaces son lo más parecido a la herencia múltiple, ya que una clase puede implementar más de un interface pero sólo puede extender una clase.

Por último, comentar que en Groovy el tipo de los datos se elige de manera dinámica cuando estos son pasados como parámetros a los métodos de nuestras clases. Esta característica de Groovy se le conoce como *multimétodo* y lo mejor es que veamos un ejemplo.

```
def multimetodo(Object o) { return 'objeto' }
def multimetodo(String o) { return 'string' }

Object x = 1
Object y = 'foo'

assert 'objeto' == multimetodo(x)
assert 'string' == multimetodo(y)//En Java, esta llamada hubiera
```

```
devuelto la palabra 'objeto'
```

En el ejemplo anterior, el tipo de datos estático de la variable *x* es *Object* mientras que su tipo dinámico es *Integer*, mientras que el tipo estático de *y* es *Object* mientras que su tipo dinámico es *String*. Debido a que Groovy intenta utilizar el tipo dinámico de las variables, en el segundo caso se ejecuta el método que tiene como parámetro una variable de tipo String.

2.4. GroovyBeans

En Java, los JavaBeans se introdujeron en su momento para definir un modelo de componentes para la construcción de aplicaciones Java. Básicamente el modelo consiste en una serie de convenciones en cuanto a los nombres que permiten a las clases Java comunicarse unas con otras. Groovy utiliza también el concepto inherente a los JavaBeans, pero lo transforma para dar paso a los GroovyBeans, con unas mejoras particulares, como facilitar el acceso a los métodos. Vayamos paso a paso.

Empecemos por la declaración de los GroovyBeans. Imaginemos que tenemos de nuevo la clase Libro con tan solo la propiedad título. En Java tendríamos el siguiente JavaBean:

```
public class Libro implements java.io.Serializable {
    private String titulo;

    public String getTitulo(){
        return titulo;
    }

    public void setTitulo(String valor){
        titulo = valor;
    }
}
```

Mientras que en Groovy, simplemente necesitaríamos tener esto otro:

```
class Libro implements java.io.Serializable {
        String titulo
}
```

Las diferencias son evidentes, ¿no crees? Pero no sólo es una cuestión de ahorro en código, sino también de eficiencia. Imagina simplemente que por cualquier motivo, tuvieras que cambiar el nombre al campo *titulo*, ese simple cambio en Java supondría cambiar el código en tres lugares diferentes, sin embargo, en Groovy, simplemente cambiando el nombre de la propiedad lo tendríamos todo. Además, Groovy también nos ahorra tener que escribir los métodos *setTitulo()* y *getTitulo()*. Esto lo hace únicamente cuando no se han especificado dichos métodos en el GroovyBean. Además, Groovy sabe perfectamente cuando debe especificar un método *set* para acceder a una propiedad. Por ejemplo, si establecemos que una propiedad de nuestro GroovyBean es *final*, esta propiedad solamente será de lectura, así que Groovy no implementa su correspondiente método *set*.

De igual forma, Groovy también permite el acceso a las propiedades utilizando el operador ., como por ejemplo `libro.titulo = 'Groovy in action'`. Pero, como siempre, hay algo más. Observemos detenidamente el siguiente ejemplo.

```
class Persona {
    String nombre, apellidos

    String getNombreCompleto(){
        return "$nombre $apellidos"
    }
}

def juan = new Persona(nombre:"Juan")
juan.apellidos = "Martínez"

assert juan.nombreCompleto == "Juan Martínez"
```

Tenemos un método (*getNombreCompleto()*) para obtener el nombre completo de la persona en cuestión. Pues Groovy además, crea al vuelo una propiedad equivalente al método *get*, en nuestro caso, *nombreCompleto*.

En ocasiones, los métodos *get* no tienen porque devolver el valor concreto del campo en cuestión, sino que es posible que hayan hecho algún tratamiento sobre dicho campo para modificar el valor devuelto. Imaginemos una clase que duplique el valor de su única propiedad cuando se invoca por los métodos tradicionales (bien mediante el método get o mediante el operador .).

```
class DobleValor {
    def valor

    void setValor(valor){
        this.valor = valor
    }

    def getValor(){
        valor * 2
    }
}

def doble = new DobleValor(valor: 300)

assert 600 == doble.getValor()
assert 600 == doble.valor
```

Por un lado tenemos que el valor devuelto por el método *getValor()* es 600, pero en realidad el valor de la propiedad es 300. ¿Cómo podemos recuperar ese valor? Muy sencillo. Groovy introduce el operador .@ para solucionarnos la papeleta en este tipo de casos. Así que al fragmento de código anterior le podemos añadir `assert 300 == doble.@valor`.

2.5. Otros características interesantes de Groovy

Groovy presenta además, otra característica conocida como el operador *spread* *. Este operador permite pasar una lista a un método que contiene una serie de parámetros. De esta forma, se consigue en cierta forma sobrecargar el método en cuestión como si permitiese también la introducción de sus parámetros en forma de lista. Veamos un ejemplo. Imagina que tienes un método que devuelve una lista de resultados, los cuales deben pasados uno por uno a otro método.

```
def getLista(){
    return [1,2,3,4,5]
}

def suma(a, b, c, d, e){
    return a + b + c + d + e
}

assert 15 == suma(*lista)
```

Sencillo, y sin tener que destripar la lista en varios parámetros.