El lenguaje Groovy

Índice

En está segunda sesión, vamos a tratar en profundidad los aspectos más importantes del lenguaje Groovy, tales como los tipos de datos simples y las colecciones, para terminar viendo las estructuras de control, tanto las heredadas de Java como las propias de Groovy y comprobaremos lo ameno que puede ser programar en Groovy.

1. Tipos de datos simples

1.1. Tipos primitivos y referencias

En Java, existen los tipos de datos primitivos (`int, double, float, char`, etc) y las referencias (`Object, String`, etc), Los tipos de datos primitivos son aquellos que tienen valor por si mismos, bien sea un entero, un carácter o un número en coma flotante y es imposible crear nuevos tipos de datos primitivos.

Mientras que las referencias son identificadores de instancias de clases Java y, como su nombre indica, simplemente es una referencia a un objeto. En los tipos de datos primitivos es imposible realizar llamadas a métodos y éstos no pueden ser utilizados en aquellos lugares donde se espera la presencia de un tipo *java.lang.Object*. Esto hace que determinados fragmentos de código de nuestros programas, se compliquen demasiado, como puede ser el siguiente ejemplo que realiza la suma posición por posición de un par de vectores de enteros.

```
ArrayList resultados = new ArrayList();
for (int i=0; i < listaUno.size(); i++){
    Integer primero = (Integer)listaUno.get(i);
    Integer segundo = (Integer)listaDos.get(i);

    int suma = primero.intValue() + segundo.intValue();
    resultados.add(new Integer(suma));
}
```

Todo parece indicar que en un futuro cercano, Java mejorará esta aproximación, pero mientras tanto Groovy ya ha dado su solución para no tener que realizar la conversión de datos para realizar simples sumas. En Groovy, todo es un objeto y una solución al ejemplo anterior podría ser utilizando el método `plus()` que Groovy añade al tipo Integer: `resultados.add(primero.plus(segundo))`, con lo que nos podríamos ahorrar el paso de la conversión de tipo de dato referencia a tipo de dato primitivo (`primero.intValue()`).

Sin embargo, esta solución también se podría conseguir en Java si se añadiera el método plus a la clase Integer. Así que Groovy decide ir un poco más lejos y permite la utilización de operadores entre objetos, con lo que la solución en Groovy sería `resultados.add (primero + segundo)`.

De esta forma, lo que en Groovy puede parecer una variable de tipo de dato primitivo, en realidad es una referencia a una clase Java, tal y como se muestra en la siguiente tabla.

Tipo primitivo	Clase utilizada
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

Así que cada vez que utilices un tipo de datos primitivo en tus programas en Groovy, en realidad estás utilizando la correspondiente clase indicada en la tabla anterior, con lo que, puedes ahorrarte el uso de tipos primitivos en Groovy, porque por detrás se está haciendo una conversión a un tipo de dato referencia.

1.2. Boxing, unboxing y autoboxing

La conversión de un tipo de dato primitivo a un tipo de dato referencia se conoce en Java como *boxing*, mientras que la conversión de un tipo de dato referencia a un tipo de dato primitivo se conoce *unboxing*. Groovy automatiza estas operaciones en lo que se conoce como *autoboxing*.

Pero, si Groovy convierte todo a un tipo de dato referencia, ¿qué pasa con aquellos métodos de Java que esperan un parámetro de tipo de dato primitivo? No hay de que preocuparse, el autoboxing de Groovy se encarga de eso. Por ejemplo, en el método *indexOf* de la clase *java.lang.String* se espera como parámetro un entero (`int`) que indica el carácter buscado en la cadena, devolviendo también un entero indicando la posición en la que se ha encontrado el caracter. Si probamos el siguiente ejemplo, veremos como todo funciona correctamente, ya que Groovy se encarga de realizar el autoboxing allí donde considere oportuno, en este caso, en el paso del parámetro a la función *indexOf*. El siguiente código trata de obtener la posición de la primera 'o' de la cadena.

```
assert 'Hola Mundo'.indexOf(111) == 1
```

En un principio Groovy, debería convertir el tipo de datos `int` del valor 111 a `Integer`, sin embargo, la función `indexOf()` requiere un parámetro de tipo `int`, con lo que el autoboxing de Groovy funciona de tal forma para convertir el parámetro al tipo de dato requerido, en este caso, `int`.

Otro aspecto interesante del autoboxing de Groovy es que no siempre se ejecuta el autoboxing para la realización de determinadas operaciones. Por ejemplo, en la operación

`1 + 2`, podemos pensar que los valores 1 y 2 son del tipo referencia *Integer*, lo cual es cierto y que para poder realizarse la operación, éstos deben ser convertidos a tipo *int*, lo cual no es cierto.

De Groovy se dice que es incluso más orientado a objetos que Java y se dice por cuestiones como esta. En la operación `1 + 2`, lo que Groovy está realmente ejecutando es `1.plus(2)`, con lo que no es necesaria ninguna conversión para realizar esta operación.

1.3. Tipado dinámico

Hasta el momento, en prácticamente todos los ejemplos que hemos visto en Groovy, hemos obviado especificar los tipos de datos utilizados, dejando que Groovy lo haga por nosotros. Esto es lo que se conoce como tipado dinámico y en este punto, vamos a ver los pros y los contras de su uso. La siguiente tabla, muestra un ejemplo con definiciones de variables y como actúa Groovy en cada caso.

Sentencia	Tipo de variable
def a = 2	java.lang.Integer
def b = 0.4f	java.lang.Float
int c = 3	java.lang.Integer
float d = 4	java.lang.Float
Integer e = 6	java.lang.Integer
String f = '1'	java.lang.String

La palabra reservada `def` se utiliza cuando no queremos especificar ningún tipo de dato en especial y dejamos que Groovy decida por nosotros, tal y como aparece en los dos primeros ejemplos. En los dos ejemplos siguientes, podemos ver como independientemente de declarar una variable como tipo de dato primitivo, ésta acabará siendo un tipo de dato referencia. Los dos últimos ejemplos, servirán a la persona que esté leyendo el código para entender que esa variable es un objeto.

Aquí es importante resaltar que Groovy es un lenguaje de tipado dinámico de datos seguro, lo que quiere decir, que Groovy no nos va a permitir realizar operaciones de una determinada clase a un objeto definido de forma diferente. Por ejemplo, en el trozo de código `String f = '1'`, la variable `f` nunca va a poder ser utilizada para realizar operaciones matemáticas como si fuera de la clase `java.lang.Number` salvo que hagamos la correspondiente conversión.

Poder elegir si utilizamos tipado dinámico o estático, es una de las mejores cosas que tiene Groovy. En Internet existen muchos foros de discusión creados a partir de este debate donde se exponen los pros y los contras de cada método. El *tipado estático* nos proporciona más información para la optimización de nuestros programas y revelan información adicional sobre el significado de la variable o del parámetro utilizado en un

método determinado.

Por otro lado, el *tipado dinámico* no sólo es el método utilizado por los programadores vagos que no quieren estar definiendo los tipos de las variables, sino que también se utiliza cuando la salida de un método es utilizado como entrada de otro sin tener que hacer ningún trabajo extra por nuestra parte. De esta forma, el programador deja a Groovy que se encargue de la conversión de los datos en caso de que sea necesario y factible.

Otro uso interesante del tipado dinámico, es lo que se conoce como el *duck typing* (tipado de patos) y es que, si hay algo que camina como un pato y habla como un pato, lo más probable es que sea un pato. El tipado dinámico es interesante utilizarlo cuando se desconoce a ciencia cierta el tipo de datos de una determinada variable o parámetro. Esto nos proporciona un gran nivel de reutilización de nuestro código, así como la posibilidad de implementar funciones genéricas.

1.4. Sobrecarga de operadores

La sobrecarga de operadores es un concepto de la programación orientada a objetos que se refiere a la posibilidad de tener un método de una clase con un determinado comportamiento y disponer también de uno más específico para un subtipo de esta clase. Como veíamos en el ejemplo, cuando ejecutamos `1 + 2`, lo que realmente se está ejecutando es `1.plus(2)`. La siguiente tabla muestra una completa referencia de esta característica de Groovy, con el correspondiente método a reescribir en caso de querer sobrecargar un operador.

Operador	Nombre	Método	Funciona con
a + b	Suma	a.plus(b)	Números, cadenas, colecciones
a - b	Resta	a.minus(b)	Números, cadenas, colecciones
a * b	Multiplicación	a.multiply(b)	Números, cadenas, colecciones
a / b	División	a.div(b)	Números
a % b	Módulo	a.mod(b)	Número enteros
a++, ++a	Post-incremento, pre-incremento	a.next()	Números, cadenas, rangos
a--, --a	Post-decremento, pre-decremento	a.previous()	Números, cadenas, rangos
a**b	Potencia	a.power(b)	Números
a | b	Operación or	a.or(b)	Números enteros
a & b	Operación and	a.and(b)	Números enteros

a ^ b	Operación xor	a.xor(b)	Números enteros
-a	Negación	a.negate()	Números enteros y cadenas
a[b]	Elemento de array	a.getAt(b)	Objetos, listas, mapas, String, Array
a[b] = c	Asignación a un elemento de array	a.putAt(b,c)	Objetos, listas, mapas, StringBuffer, Array
switch (a){ case b:}	Clasificación	b.isCase(a)	Objetos, rangos, listas, colecciones, patrones, closures
a == b	Igualdad	a.equals(b)	Objetos
a != b	No igualdad	! a.equals(b)	Objetos
a <=> b	Comparación	a.compareTo(b)	java.lang.Comparable
a > b	Mayor que	a.compareTo(b) > 0	
a >= b	Mayor o igual que	a.compareTo(b) >= 0	
a < b	Menor que	a.compareTo(b) < 0	
a <= b	a.compareTo(b) <= 0	Menor o igual que	

Pero pasemos a la acción en cuanto a la sobrecarga de operadores con un ejemplo. Imaginemos que tenemos una clase *Dinero* y que necesitamos implementar dos métodos para comprobar la igualdad y la suma de cantidades monetarias. Una forma de hacerlo sería sobrecargando los métodos `equals` `==` y `plus` `+`. Por supuesto, la suma sólo se podrá efectuar cuando las dos cantidades a sumar utilicen la misma moneda, al igual que dos cantidades no serían iguales si no se refiriese a la misma moneda.

```
class Dinero {
    private int cantidad
    private String moneda

    Dinero (cantidadValor, monedaValor){
        cantidad = cantidadValor
        moneda = monedaValor
    }

    boolean equals (Object otro){
        if (null == otro)
            return false
        if (! (otro instanceof Dinero))
            return false
        if (moneda != otro.moneda)
            return false
        if (cantidad != otro.cantidad)
            return false
        return true
    }

    int hashCode(){
```

```
        cantidad.hashCode() + moneda.hashCode()
    }

    Dinero plus (Dinero otro){
        if (null == otro)
            return null
        if (otro.moneda != moneda){
            throw new IllegalArgumentException(
                "no puedes sumar $otro.moneda a $moneda")
        }
        return new Dinero(cantidad + otro.cantidad, moneda)
    }
}

def uneuro = new Dinero(1,'EURO')
assert uneuro == new Dinero(1,'EURO')
assert uneuro + uneuro == new Dinero(2,'EURO')
assert uneuro == new Dinero(2,'EURO') : "un euro no son dos euros"
```

1.5. Trabajo con cadenas

Groovy nos facilita el trabajo con las cadenas de texto en mayor medida que lo hace Java, añadiendo su propia librería `GString`, con lo que además de los métodos ofrecidos por la clase `java.lang.String`, Groovy cuenta con más metodos ofrecidos por `groovy.lang.GString`. Una característica de como trabaja Groovy con las cadenas de texto es que nos permite introducir variables en las cadenas sin tener que utilizar caracteres de escape como por ejemplo `"hola $minombre"`, donde en la misma cadena se introduce el valor de una variable. Esto es típico de algunos lenguajes de programación como PHP y facilita la lectura de nuestro código.

La siguiente tabla muestra las diferentes formas que hay en Groovy para crear una cadena de texto:

Caracteres utilizados	Ejemplo	Soporte GString
Comillas simples	'hola Juan'	No
Comillas dobles	"hola $nombre"	Sí
3 comillas simples	'''---------- Total:0.02 ----------'''	No
3 comillas dobles	"""---------- Total:$total ----------"""	Sí
Símbolo /	/x(\d*)y/	Sí

La diferencia entre las comillas simples y las dobles es básicamente que las dobles son tratadas como cadenas de tipo GString y la posibilidad de incluir variables precedidas del símbolo $ para mostrar su valor. Las cadenas introducidas con el símbolo / son utilizadas con expresiones regulares, como veremos más adelante.

1.6. La librería GString

La librería GString (`groovy.lang.GString`) añade determinados métodos para facilitarnos el trabajo con cadenas de texto, las cuales normalmente se crean utilizando comillas dobles. Básicamente, una cadena de tipo GString nos va a permitir introducir variables precedidas del símbolo $. También es posible introducir expresiones entre llaves (`${expresion}`), tal y como si estuviéramos escribiendo un *closure*. Veamos algunos ejemplos:

```
nombre = 'Fran'
apellidos = 'García'
salida = "Apellidos, nombre: $apellidos, $nombre"

fecha = new Date(0)
salida = "Año $fecha.year, Mes $fecha.month, Día $fecha.date"

salida = "La fecha es ${fecha.toGMTString()}"

sentenciasql = """
SELECT nombre, apellidos
FROM usuarios
WHERE anyo_nacimiento=$fecha.year
"""
```

Ahora que ya podemos declarar variables de texto, vamos a ver algunos métodos que podemos utilizar en Groovy:

```
saludo = 'Hola Juan'

assert saludo.startsWith('Hola')

assert saludo.getAt(3) == 'a'
assert saludo[3] == 'a'

assert saludo.indexOf('Juan') == 5
assert saludo.contains('Juan')

assert saludo[5..8] == 'Juan'

assert 'Buenos días' + saludo - 'Hola' == 'Buenos días Juan'

assert saludo.count('a') == 2

assert 'b'.padLeft(3) == '  b'
assert 'b'.padRight(3,'_') == 'b__'
assert 'b'.center(3) == ' b '
assert 'b' * 3 == 'bbb'
```

1.7. Expresiones regulares

Las expresiones regulares nos permiten especificar un patrón y buscar si éste aparece en un fragmento de texto determinado. Groovy deje que sea Java la encargada del tratamiento de las expresiones regulares, pero además, añade tres métodos para facilitarnos este trabajo:

- El operador =~: *find*

- El operador ==~: *match*
- El operador ~String: *pattern*

Con los patrones de las expresiones regulares, realmente estamos indicando que estamos buscando exactamente. Veamos algunos ejemplos:

Patrón	Significado
algo de texto	simplemente encontrará la frase "algo de texto"
algo de\s+texto	encontrará frases que empiecen con "algo de", vayan seguidos por uno o más caracteres y terminen con la palabra texto
\d\d/\d\d/\d\d\d\d	detectará fechas como por ejemplo 28/06/2008

El punto clave de los patrones de las expresiones regulares, son los símbolos, que los podemos sustituir por determinados fragmentos de texto. La siguiente tabla presenta estos símbolos:

Símbolo	Significado
.	Cualquier carácter
^	El inicio de una línea
$	El final de una línea
\d	Un dígito
\D	Cualquier cosa excepto un dígito
\s	Un espacio en blanco
\S	Cualquier cosa excepto un espacio en blanco
\w	Un carácter de texto
\W	Cualquier carácter excepto los de texto
\b	Límite de palabras
()	Agrupación
(x|y)	O x o y
x*	Cero o más ocurrencias de x
x+	Una o más ocurrencias de x
x?	Cero o una ocurrencia de x
x{m,n}	Entre m y n ocurrencias de x
x{m}	Exactamente m ocurrencias de x

[a-d]	Incluye los caracteres a, b, c y d
[^a]	Cualquier carácter excepto la letra a

Las expresiones regulares nos ayudarán en Groovy a:

- Indicarnos si un determinado patrón encaja completamente con un texto
- Si existe alguna ocurrencia de un patrón en una cadena
- Contar el número de ocurrencias
- Hacer algo con una determinada ocurrencia
- Reemplazar todas las ocurrencias con un determinado texto
- Separar una cadena en múltiples cadenas a partir de las ocurrencias que aparezcan en la misma

El siguiente fragmento de código muestra el funcionamiento básico de las expresiones regulares.

```
refran = "tres tristes tigres tigraban en un tigral"

//Compruebo que hay al menos un fragmento de código que empieza por t,
//le siga cualquier caracter y posteriormente haya una g
assert refran =~ /t.g/

//Compruebo que el refrán esté compuesto sólo
//por palabras seguidas de un espacio
assert refran ==~ /(\w+ \w+)*/

//Compruebo que el valor de una operación de tipo match es un booleano
assert (refran ==~ /(\w+ \w+)*/) instanceof java.lang.Boolean

//A diferencia que una operación de tipo find,
//las operaciones match se evalúan por completo contra una cadena
assert (refran ==~ /t.g/) == false

//Sustituyo las palabras por el caracter x
assert (refran.replaceAll(/\w+/,'x')) == 'x x x x x x x'

//Devuelve un array con todas las palabras del refrán
palabras = refran.split(/ /)
assert palabras.size() == 7
assert palabras[2] == 'tigres'
assert palabras.getAt(3) == 'tigraban'
```

Es importante resaltar la diferencia entre el operador *find* y el operador *match*. El operador *match* es más restrictivo puesto que intenta hacer coincidir un patrón con la cadena entera, mientras que el operador *find*, sólo pretende encontrar una ocurrencia del patrón en la cadena.

Ya sabemos como localizar fragmentos de texto en cadenas, pero ¿y si queremos hacer algo con estas cadenas encontradas? Groovy nos vuelve a facilitar esta tarea y pone a nuestra disposición un par de formas para recorrer las ocurrencias encontradas: *each* y *eachMatch*. Por un lado, al método *eachMatch* se le pasa una cadena con un patrón de expresión regular como parámetro: `String.eachMatch(patron)`, mientras que al método *each* se le pasa directamente el resultado de una operación de tipo `find()` o `match()`: `Matcher.each()`. Veámos ambos métodos en funcionamiento.

```
refran = "tres tristes tigres tigraban en un tigral"

//Busco todas las palabras que acaben en 'es'
rima = /\b\w*es\b/
resultado = ''
refran.eachMatch(rima) { match ->
    resultado += match + ' '
}

assert resultado == 'tres tristes tigres '

//Hago lo mismo con el método each
resultado = ''
(refran =~ rima).each { match ->
    resultado += match + ' '
}

assert resultado == 'tres tristes tigres '

//Sustituyo todas las rimas por guiones bajos
assert (refran.replaceAll(rima){ it-'es'+'__'} == 'tr__ trist__ tigr__
tigraban en un tigral')
```

1.8. Números

El GDK de Groovy introduce algunos métodos interesantes en cuanto al tratamiento de números. Estos métodos funcionan como closures y nos servirán como otras formas de realizar bucles. Estos métodos son:

- `times()`, se utiliza para realizar repeticiones
- `upto()`, utilizado para realizar una secuencia de acciones de forma creciente
- `downto()`, igual que el anterior pero de forma decreciente
- `step()`, es el método general para realizar una secuencia paso a paso

Pero como siempre, veamos varios ejemplos.

```
def cadena = ''
10.times {
    cadena += 'g'
}
assert cadena == 'gggggggggg'

cadena = ''
1.upto(5) { numero ->
    cadena += numero
}

assert cadena == '12345'

cadena = ''
2.downto(-2) { numero ->
    cadena += numero + ' '
}

assert cadena == '2 1 0 -1 -2 '

cadena = ''
0.step(0.5, 0.1) { numero ->
    cadena += numero + ' '
}
```

```
assert cadena == '0 0.1 0.2 0.3 0.4 '
```

2. Colecciones

Ahora que ya hemos introducido los tipos de datos simples, llega el turno de hablar de las colecciones presentes en Groovy. En este apartado vamos a ver tres tipos de datos. Por un lado, las *listas* y los *mapas*, que tienen prácticamente las mismas connotaciones que en Java, con alguna nueva característica que añade Groovy, y por otro, los *rangos*, un concepto que no existe en Java.

2.1. Rangos

Empecemos por lo novedoso. Cuantas veces no nos habremos encontrado con un bloque de código similar al siguiente

```
for (int i=0;i<10;i++){
        //hacer algo con la variable i
}
```

El anterior fragmento de código se ejecutará empezando en un límite inferior (0) y terminará de ejecutarse cuando la variable *i* llegue al valor 10. Uno de los objetivos de Groovy consiste en facilitar al programador la lectura y la comprensión del código, así que los creadores de Groovy pensaron que sería útil introducir el concepto de rango, el cual tendría por definición un límite inferior y uno superior.

Para especificar un rango, simplemente se escribe el límite inferior seguido de dos puntos y el límite superior, `limiteInferior..limiteSuperior`. Este rango indicaría que ambos valores establecidos están dentro del rango, así que si queremos indicarle que el límite superior no está dentro del rango, debemos utilizar el operador ..<, `limiteInferior..<limiteSuperior`. También existen los *rangos inversos*, en los que el límite inferior es mayor que el límite superior. Veamos algunos ejemplos:

```
//Rangos inclusivos
assert (0..10).contains(5)
assert (0..10).contains(10)

//Rangos medio-exclusivos
assert (0..<10).contains(9)
assert (0..<10).contains(10) == false

//Comprobación de tipos
def a = 0..10
assert a instanceof Range

//Definición explícita
a = new IntRange(0,10)
assert a.contains(4)

//Rangos para fechas
def hoy = new Date()
```

```
def ayer = hoy - 1
assert (ayer..hoy).size() == 2

//Rangos para caracteres
assert ('a'..'f').contains('e')

//El bucle for con rangos
def salida = ''
for (elemento in 1..5){
    salida += elemento
}
assert salida == '12345'

//El bucle for con rangos inversos
salida = ''
for (elemento in 5..1){
    salida += elemento
}
assert salida == '54321'

//Simulación del bucle for con rangos inversos
//y el método each con un closure
salida = ''
(5..<1).each { elemento ->
    salida += elemento
}
assert salida == '5432'
```

Los rangos son objetos y como tales, pueden ser pasados como parámetros a funciones o bien ejecutar sus propios métodos. Un uso interesante de los rangos es el filtrado de datos. También es interesante verlos como clasificador de grupos y su utilidad se puede comprobar en los bloques de código *switch*.

```
//Rangos como clasificador de grupos
edad = 31
switch (edad){
    case 16..20: interesAplicado = 0.25; break
    case 21..50: interesAplicado = 0.30; break
    case 51..65: interesAplicado = 0.35; break
}
assert interesAplicado == 0.30

//Rangos para el filtrado de datos
edades = [16,29,34,42,55]
joven = 16..30
assert edades.grep(joven) == [16,29]
```

Como se ha podido comprobar, podemos especificar rangos para fechas e incluso para cadenas. En realidad, cualquier tipo de dato puede ser utilizado en un rango, siempre que se cumplan una serie de condiciones:

- El tipo implemente los métodos `next()` y `previous()`, que sobrecargan los operadores `++` y `--`
- El tipo implemente `java.lang.Comparable`, implementando el método `compareTo()` que sobrecarga el operador `<=>`

El siguiente fragmento de código se refiere a la clase *DiasDeLaSemana* y vamos a conseguir que se pueda utilizar en rangos, con los requisitos comentados.

```
class DiasDeLaSemana implements Comparable {
       static final DIAS = ['Lun','Mar','Mie','Jue','Vie','Sab','Dom']
       private int index = 0

       DiasDeLaSemana(String dia){
           index = DIAS.indexOf(dia)
       }

       DiasDeLaSemana next(){
           return new DiasDeLaSemana(DIAS[(index+1) % DIAS.size()])
       }

       DiasDeLaSemana previous(){
           return new DiasDeLaSemana(DIAS[(index-1)])
       }

       int compareTo(Object otro){
           return this.index <=> otro.index
       }

       String toString(){
           return DIAS[index]
       }
}

def lunes = new DiasDeLaSemana('Lun')
def viernes = new DiasDeLaSemana('Vie')

def diasLaborables = ''
for (dia in lunes..viernes){
    diasLaborables += dia.toString() + ' '
}

assert diasLaborables == 'Lun Mar Mie Jue Vie '
```

Comprobemos los requisitos para poder utilizar nuestra clase `DiasDeLaSemana` en un rango:

- La clase DiasDeLaSemana se han sobrecargado los métodos `next()` y `previous()`
- La clase DiasDeLaSemana implementa la clase `java.lang.Comparable`
- La clase DiasDeLaSemana sobrecarga el método `compareTo()`

Con esto, como se puede comprobar en la parte final del código, nuestra nueva clase puede ser incluida en la definición de rangos.

2.2. Listas

En Java, agregar un nuevo elemento a un array no es algo trivial. Una solución es convertir el array a una lista del tipo `java.util.List`, añadir el nuevo elemento y volver a convertir la lista en un array. Otra solución pasa por construir un nuevo array del tamaño del array original más uno, copiar los viejos valores y el nuevo elemento. Eso es la parte negativa de los arrays en Java. La parte positiva es que nos permite trabajar con índices en los arrays para recuperar su información, así como modificar su valor, como por ejemplo, `miarray[indice] = nuevoelemento`. Groovy se aprovecha de la parte positiva de Java en este sentido, y añade nuevas características para mejorar su parte negativa.

La definición de una lista en Groovy se consigue utilizando los corchetes [] y especificando los valores de la lista. Si no especificamos ningún valor entre los corchetes, declararemos una lista vacía. Por defecto, las listas en Groovy son del tipo `java.util.ArrayList`. Podemos rellenar fácilmente las listas a partir de otras con el método `addAll()`. También se pueden definir listas a partir de otras con el constructor de la clase *LinkedList*.

```
miLista = [1,2,3]

assert miLista.size() == 3
assert miLista[2] == 3
assert miLista instanceof ArrayList

listaVacia = []
assert listaVacia.size() == 0

listaLarga = (0..1000).toList()
assert listaLarga[324] == 324

listaExplicita = new ArrayList()
listaExplicita.addAll(miLista)
assert listaExplicita.size == 3
listaExplicita[2] = 4
assert listaExplicita[2] == 4

listaExplicita = new LinkedList(miLista)
assert listaExplicita.size == 3
listaExplicita[2] = 4
assert listaExplicita[2] == 4
```

En el fragmento de código anterior, hemos visto como se puede especificar un valor a un elemento de la lista. Pero, ¿qué pasa si queremos especificar un mismo valor a toda la lista o un trozo de la misma? Los creadores de Groovy ya han pensado en ese problema y podemos utilizar rangos y colecciones en las listas.

```
miLista = ['a','b','c','d','e','f']

assert miLista[0..2] == ['a','b','c']//Acceso con Rangos
assert miLista[0,2,4] == ['a','c','e']//Acceso con colección de índices

//Modificar elementos
miLista[0..2] = ['x','y','z']
assert miLista == ['x','y','z','d','e','f']

//Eliminar elementos de la lista
miLista[3..5] = []
assert miLista == ['x','y','z']

//Añadir elementos a la lista
miLista[1..1] = ['y','1','2']
assert miLista == ['x','y','1','2','z']

miLista = []

//Añado objetos a la lista con el operador +
miLista += 'a'
assert miLista == ['a']

//Añado colecciones a la lista con el operador +
miLista += ['b','c']
```

```
assert miLista == ['a','b','c']

miLista = []
miLista << 'a' << 'b'
assert miLista == ['a','b']

assert miLista - ['b'] == ['a']

assert miLista * 2 == ['a','b','a','b']
```

En ocasiones las listas son utilizadas juntos a estructuras de control para controlar el flujo de nuestro programa.

```
miLista = ['a','b','c']

//Listas como clasificador de grupos
letra = 'a'
switch (letra){
    case miLista: assert true; break;
    default: assert false
}

//Listas como filtrado de datos
assert ['x','y','a'].grep(miLista) == ['a']

//Bucle for con lista
salida = ''
for (i in miLista){
    salida += i
}
assert salida == 'abc'
```

Las listas tienen una larga lista de métodos disponibles en el API de Java en la interfaz `java.util.List` para por ejemplo ordenar, unir e intersectar listas. En la siguiente sesión veremos algunos ejemplos de closures con listas como parámetros.

2.3. Mapas

Un mapa es prácticamente igual que una lista, con la salvedad de que los elementos están referenciados a partir de una clave única (sin caracteres extraños ni palabras reservadas por Groovy), `miMapa['clave'] = valor`. Podemos especificar un mapa al igual que lo hacíamos con las listas utilizando los corchetes, pero ahora debemos añadir la clave a cada valor introducido, como por ejemplo `miMapa = [a:1, b:2, c:3]`. Los mapas creados implícitamente son del tipo `java.util.HashMap`. Veámoslo con ejemplos:

```
def miMapa = [a:1, b:2, c:3]

assert miMapa instanceof HashMap
assert miMapa.size() == 3
assert miMapa['a'] == 1

//Definimos un mapa vacio
def mapaVacio = [:]
assert mapaVacio.size() == 0

//Definimos un mapa de la clase TreeMap
def mapaExplicito = new TreeMap()
mapaExplicito.putAll(miMapa)
```

```
assert mapaExplicito['c'] == 3
```

Las operaciones más comunes con los mapas se refieren a la recuperación y almacenamiento de datos a partir de la clave. Veamos algunos métodos de acceso y modificación de los elementos de un mapa:

```
def miMapa = [a:1, b:2, c:3]

//Varias formas de obtener los valores de un mapa
assert miMapa['a'] == 1
assert miMapa.a == 1
assert miMapa.get('a') == 1
//Si no existe la clave, devuelve un valor por defecto, en este caso 0
assert miMapa.get('a',0) == 1

//Asignación de valores
miMapa['d'] = 4
assert miMapa.d == 4
miMapa.e = 5
assert miMapa.e == 5
```

Los mapas en Groovy utilizan los mismos métodos que los mapas en Java y éstos están en el API de Java referente a `java.util.Map`, pero además, Groovy añade un par de métodos llamados `any()` y `every()`, los cuales, utilizados como closures, permite evaluar si todos (every) o al menos uno (any) de los elementos del mapa cumplen una determinada condición. Además, en el siguiente fragmento de código, vamos a ver como iterar sobre los mapas.

```
def miMapa = [a:1, b:2, c:3]

def resultado = ''
miMapa.each { item ->
    resultado += item.key + ':'
    resultado += item.value + ', '
}
assert resultado == 'a:1, b:2, c:3, '

resultado = ''
miMapa.each { key, value ->
    resultado += key + ':'
    resultado += value + ', '
}
assert resultado == 'a:1, b:2, c:3, '

resultado = ''
for (key in miMapa.keySet()){
    resultado += key + ':'
    resultado += miMapa[key] + ', '
}
assert resultado == 'a:1, b:2, c:3, '

resultado = ''
for (value in miMapa.values()){
    resultado += value + ' '
}
assert resultado == '1 2 3 '

def valor1 = [1, 2, 3].every { it < 5 }
assert valor1

def valor2 = [1, 2, 3].any { it > 2 }
```

```
assert valor2
```

Y para terminar con los mapas, vamos a ver otros métodos añadidos por Groovy para el manejo de los mapas, que nos permitirán:

- Crear un submapa de un mapa dado a partir de algunas claves: `subMap()`
- Encontrar todos los elementos de un mapa que cumplen una determinada condición: `findAll()`
- Encontrar un elemento de un mapa que cumpla una determinada condición: `find()`
- Realizar operaciones sobre los elementos de un mapa: `collect()`

```
def miMapa = [a:1, b:2, c:3]
def miSubmapa = miMapa.subMap(['a','b'])
assert miSubmapa.size() == 2

def miOtromapa = miMapa.findAll { entry -> entry.value > 1 }
assert miOtromapa.size() == 2
assert miOtromapa.c == 3

def encontrado = miMapa.find { entry -> entry.value < 3}
assert encontrado.key == 'a'
assert encontrado.value == 1

def miMapaDoble = miMapa.collect { entry -> entry.value *= 2}
//Todos los elementos son pares
assert miMapaDoble.every { item -> item % 2 == 0 }
```

3. Estructuras de control

Aunque a medida que hemos avanzado, hemos estado viendo diferentes estructuras de control sin necesidad de comentarlas, ha llegado el momento de ver un resumen de las mismas.

3.1. La sentencia if

La sentencia `if` es idéntica a la misma sentencia en Java, así que no vamos a entrar en más detalle. Simplemente recordar que en los bloques con una sólo línea, no es necesario utilizar las llaves {}.

```
if (true)
        assert true
else
        assert false

if (0)
        assert false
else if ([])
        assert false
else
        assert true
```

Groovy nos permite utilizar diferentes tipos de datos a evaluar en la condición, tal y como se muestra en la siguiente tabla:

Tipo	Criterio de evaluación
Boolean	True o false
Matcher	La instancia de Matcher tiene un match
Collection	La colección no está vacía
Map	El mapa no está vacío
String.GString	La cadena no está vacía
Number, Character	El valor es distinto de cero
Ninguno de los anteriores	La referencia al objeto no es nulo

3.2. El operador ternario ?:

Groovy soporta el uso del operador ternario `?:` para realizar pequeñas comprobaciones en una única línea. Este operador, ejecuta la segunda expresión, en caso de que la primera expresión sea cierta y en caso contrario, se ejecutará la tercera expresión.

```
def resultado = (1==1) ? 'OK' : 'Mal'
assert resultado == 'OK'

resultado = (1==2) ? 'OK' : 'Mal'
assert resultado == 'Mal'
```

Además, Groovy dispone también del llamado *operador Elvis* que es una forma abreviada del operador ternario en la que sólo se tiene en cuenta el valor a devolver por la expresión en el caso de que la condición sea `false` o `null`.

```
def nombreMostrado = usuario.nombre ?: "Anónimo"
```

3.3. La sentencia switch

En Java, la sentencia `switch` es demasiado restrictiva y sólo se permite su uso con los tipos de datos `int`, `byte`, `char` y `short`. Sin embargo, Groovy permite un amplio abanico de tipos de datos a utilizar con la sentencia switch. A lo largo de esta sesión hemos visto que estos datos pueden ser listas y rangos, pero ahora veremos que también podremos utilizar comprobaciones de tipos de datos con closures o incluso expresiones regulares.

```
switch (14){
    case 0://14 no es 0
            assert false; break
    case 0..13://14 no está en el rango
            assert false; break
    case [1,4,12]://14 no está en la lista
            assert false; break
    case Float://14 no es de tipo Float
            assert false; break
    case { it%3 == 0}://Comprobamos con un closure que no es múltiplo de
14
            assert false; break
```

```
    case ~/../:///14 tiene dos caracteres
            assert true; break
    default:
            assert false; break
}
```

3.4. El bucle while

El bucle `while` en Groovy es similar al mismo bucle en Java, simplemente tiene algunas diferencias con la evaluación de la expresión booleana, que coinciden con las vistas anteriormente en la sentencia *if*.

```
def lista = [1,2,3]
while (lista){
    lista.remove(0)
}
assert lista == []
```

3.5. El bucle for

Posiblemente, el bucle `for` es el más comúnmente utilizado por todos los programadores en sus programas. En Groovy se puede utilizar la estructura del bucle `for` que conocemos todos `for (int i=0;i<10;i++) print i`, sin embargo, sus creadores pensaron que era posible mejorar esta forma de iterar para hacerlo más intuitivo y crearon la siguiente estructura `for (variable in iterable) {cuerpo}`, donde la *variable* tendrá el valor en cada iteración, mientras que el tipo de datos de *iterable* suele ser *rangos*, *colecciones*, *mapas*, *arrays*, *iteradores* y *enumeradores*.

```
def resultado = ''
for (String i in 'a'..'d')    resultado += i
assert resultado == 'abcd'

resultado = ''
for (i in ['a','b','c','d'])    resultado += i
assert resultado == 'abcd'
```

3.6. La sentencia return

La sentencia *return* en Groovy tiene prácticamente las mismas connotaciones que en Java, con la diferencia de que su utilización en los métodos es opcional y en caso de que no se utilice, se devolverá el resultado de la última expresión ejecutada en el método.