Groovy & Grails: Desarrollo rápido de aplicaciones

Sesión 2: El lenguaje Groovy

El lenguaje Groovy

- Tipos de datos simples
- Colecciones
- Estructuras de control

Tipos de datos simples

- Tipos de datos primitivos
  - *int*
  - *double*
  - *float*
  - *char*

Tipos de datos simples

- Tipos de datos referencias
  - *Object*
  - *String*

Tipos de datos primitivos y referencias

- Sobre los tipos de datos primitivos no es posible realizar llamadas a funciones.

```
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));
}
```

Tipos de datos primitivos y referencias

- En Groovy, todo es un **objeto**

```
resultados.add(primero.plus(segundo))
```

Tipos de datos primitivos y referencias

- Groovy permite también la utilización de operadores entre objetos

```
resultados.add (primero + segundo)
```

Tipos de datos primitivos y referencias

- Equivalencia en Groovy de datos primitivos y referencias

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

Tipos de datos primitivos y referencias

- **Consejo:** evita el uso de tipos de datos primitivos, ya que en realidad estás utilizando tipos de datos referencia

Boxing, unboxing y autoboxing

- **Boxing**: conversión tipo de dato primitivo a tipo de dato referencia
- **Unboxing**: conversión tipo de dato referencia a tipo de dato primitivo

Boxing, unboxing y autoboxing

- **Autoboxing**: groovy automatiza las operaciones de boxing y unboxing.

  Groovy sabe cuando debe hacer autoboxing y cuando no.

```
'Hola Mundo'.indexOf(111)
```

Boxing, unboxing y autoboxing

- **Autoboxing**: Groovy no hace autoboxing cuando hay formas más rápidas de realizar operaciones

```
1 + 2
```

```
1.plus(2)
```

Tipado dinámico

- Consiste en dejar al sistema que elija él mismo el tipo de datos de una variable determinada

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

Tipado dinámico

- Tipado dinámico seguro
- Pros y contras del tipado dinámico
- Duck typing

Sobrecarga de operadores

- Se produce cuando algunos o todos los operadores (+, = o ==) tienen diferentes implementaciones dependiendo del contexto en que se usan.

Sobrecarga de operadores

Operador	Método
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.multiply(b)
a / b	a.div(b)
a % b	a.mod(b)
a++, ++a	a.next()
a--, --a	a.previous()
a**b	a.power(b)

Sobrecarga de operadores

Operador	Método
a | b	a.or(b)
a & b	a.and(b)
a ^ b	a.xor(b)
-a	a.negate()
a[b]	a.getAt(b)
a[b] = c	a.putAt(b,c)
switch(a){ case b: }	a.isCase(b)
a == b	a.equals(b)

Sobrecarga de operadores

Operador	Método
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

Sobrecarga de operadores

- Ejemplo: Clase Dinero

Trabajo con cadenas

- Groovy permite trabajar tanto con las funciones de la clase *java.lang.String* como la clase propia de Groovy *groovy.lang.GString*.
- Permite incluir en las cadenas de texto variables sin tener que utilizar caracteres de escape.

```
“hola $minombre”
```

Trabajo con cadenas

- Varias formas de definir cadenas de texto en Groovy

Caracteres utilizados	Ejemplo
Comillas simples	‘hola Juan’
Comillas dobles	“hola $nombre”
3 comillas simples	‘‘‘---------- Total:0.02 ----------‘‘‘
3 comillas dobles	“““---------- Total:$total ----------”””
Símbolo /	/x(\d*)y/

La librería GString

```
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
"""
```

La librería GString

```
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'
```

Expresiones regulares

- Permiten especificar un patrón y buscar si éste aparece en una cadena de texto
- Groovy deja a Java que se encargue de las expresiones regulares
- Groovy añade 3 métodos
  - =~: find
  - ==~: match
  - ~String: pattern

Expresiones regulares

Símbolo	Ejemplo
.	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

Expresiones regulares

Símbolo	Ejemplo
(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

Expresiones regulares

- Las expresiones regulares permiten:
  - 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

Expresiones regulares

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

assert refran =~ /t.g/
assert refran ==~ /(\w+ \w+)*/
assert (refran ==~ /(\w+ \w+)*/) instanceof java.lang.Boolean
assert (refran ==~ /t.g/) == false
assert (refran.replaceAll(/\w+/,'x')) == 'x x x x x x x'

palabras = refran.split(/ /)
assert palabras.size() == 7
assert palabras[2] == 'tigres'
assert palabras.getAt(3) == 'tigraban'
```

Expresiones regulares

- ¿Qué hacer con las cadenas encontradas?
  - eachMatch()
  - each()

Expresiones regulares

```
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 ’
```

Expresiones regulares

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

//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')
```

Números

- El GDK de Groovy introduce algunos métodos interesantes para el tratamiento de números
- Estos métodos funcionan como closures
  - *times()*
  - *upto()*
  - *downto()*
  - *step()*

Números

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

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

assert cadena == '12345’
```

Números

```
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 '
```

Colecciones

- Rangos
- Listas
- Mapas

Rangos

- Cumplen con uno de los objetivos de Groovy que consiste en facilitar la lectura del código fuente de una aplicación
- Se definen con un límite inferior y uno superior

```
//Ambos valores están incluidos en el rango
limiteInferior .. limiteSuperior

El límite superior no está incluido en el rango
limiteInferior ..< limiteSuperior
```

Rangos

```
//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

```
//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'
```

Rangos

```
//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'
```

Rangos

- Los rangos son objetos y pueden ser pasados como parámetros a funciones
- Rangos como filtrado de datos

Rangos

```
//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]
```

Rangos

- Cualquier tipo de dato puede ser utilizado en un rango
  - El tipo debe implementar los métodos next() y previous()
  - El tipo debe implementar *java.lang.Comparable* y el método *compareTo()*

Rangos

```
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]}
}
```

Rangos

```
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 '
```

Listas

- Los arrays en Java no son fáciles de tratar
- Sin embargo, permite el acceso a elementos por su posición mediante el operador []
- Groovy aprovecha lo bueno de Java y mejora lo malo
- En Groovy las listas se definen utilizando los corchetes

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

Listas

```
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
```

Listas

```
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
```

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']
```

Listas

```
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']
```

Listas

```
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'
```

Mapas

- Prácticamente idénticos a las listas
- Se referencian a partir de una clave única
- Se definen igual que las listas, pero incluyendo la clave única

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

- Los mapas son del tipo *java.util.HashMap*

Mapas

```
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
```

Mapas

```
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
assert miMapa.get('a',0) == 1 //Si no existe la clave, devuelve un valor por
defecto, en este caso 0

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

Mapas

- Los métodos de los mapas están disponible en el API de la clase *java.util.Map*
- Groovy añade dos métodos en forma de closure
  - *any()*, al menos uno de los elementos cumplen la condición pasada por parámetro
  - *every()*, todos los elementos cumplen la condición pasada por parámetro

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, '
```

Mapas

```
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 '
```

Mapas

- Groovy añade cuatro métodos en forma de closure para la gestión de los mapas
  - *subMap()*, crea un submapa a partir de algunas claves
  - *findAll()*, encuentra todos los elementos que cumplen una condición
  - *find(),* encuentra un elemento de un mapa que cumpla una condición
  - *collect(),* realiza operaciones sobre los elementos de un mapa

Mapas

```
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}
assert miMapaDoble.every { item -> item % 2 == 0 }
```

Estructuras de control

- La sentencia if
- El operador ternario ?:
- La sentencia switch
- El bucle while
- El bucle for
- La sentencia return

La sentencia if

- Igual que en Java

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

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

La sentencia if

- Tipos de datos utilizados en la condición

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 es no nulo

El operador ternario ?:

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

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

La sentencia switch

- En Java es muy restrictiva y sólo se puede utilizar con *int, byte, char y short*
- Groovy permite un amplio abanico de tipos de datos

La sentencia switch

```
switch (14){
        case 0:                          assert false; break
        case 0..13:                      assert false; break
        case [1,4,12]:                   assert false; break
        case Float:                      assert false; break
        case { it%3 == 0}:               assert false; break
        case ~/../:                      assert true; break
        default:                  assert false; break
}
```

El bucle while

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

El bucle for

- Podemos utilizar tanto
  - *for(int i=0;i<10;i++) print i*
- Como
  - *for (variable in iterable) {cuerpo}*

```
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'
```

La sentencia return

- Su utilización en los métodos es opcional
- En caso de que no se utilice, se devolverá el resultado de la última operación realizada