



Groovy & Grails: Desarrollo rápido de aplicaciones

Sesión 3: Aspectos avanzados en Groovy



Aspectos avanzados en Groovy

- Closures
- Groovy como lenguaje orientado a objetos



Closures

- Definición de closure
- Declarando closures
- Los closures como objetos
- Usos de los closures
- Más métodos de los closures
- Valores devueltos en los closures



Closures

- **Definición**

“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”



Closures

- Los closures son objetos
- Se pasan como parámetro a funciones
- Aportan agilidad al programador



Declarando closures

- Los closures son bloques de código encerrado entre llaves { }

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

imprimeNombre()

nombre = "Yolanda"
imprimeNombre()
```



Declarando closures

- Closures con parámetros

```
def imprimeNombre = { nombre -> println "Mi nombre es ${nombre}"}  
  
imprimeNombre("Juan")  
imprimeNombre "Yolanda" //Los paréntesis son opcionales
```



Declarando closures

- Closures con múltiples parámetros

```
//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"
```




Declarando closures

- Closures con un parámetro implícito

```
def imprimeNombre = { println "Mi nombre es $it" }  
  
imprimeNombre("Juan")  
imprimeNombre "Yolanda"
```



Declarando closures

- Closures a partir de métodos
- Operador referencia &



Declarando closures

```
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)
```



Declarando closures

```
Closure primerClosure = primero.&validar  
  
def palabras = ["cadena larga", "mediana", "corta"]  
  
assert "mediana" == palabras.find(primerClosure)  
assert "corta" == palabras.find(segundo.&validar)
```



Declarando closures

- Closures multimétodo

```
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)
```



Closures como objetos

- Método *each()*

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

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



Usos de los closures

- ¿Cómo invocar un closure x?
 - `x.call()`
 - `x()`

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



Usos de los closures

- Closures como parámetros

```
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
```




Usos de los closures

- Valores por defecto para los parámetros

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



Más métodos de los closures

- Los closures son instancias de la clase *groovy.lang.Closure*
- Método *call()* para invocar a los closures
- Método *each()* para iterar por los elementos de determinados tipos de datos



Más métodos de los closures

- Método *getParameterTypes()* para saber el número de parámetros pasados al closure

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

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



Más métodos de los closures

- Método *curry()* para implementar la técnica *currying* de programación
- Consiste en crear una función a partir de otra sin su último parámetro

```
def suma = { x, y -> x + y }  
def sumaUno = suma.curry(1)  
  
assert suma(4,3) == 7  
assert sumaUno(5) == 6
```



Valores devueltos en los closures

- Dos formas de devolver valores
 - De forma implícita

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

- De forma explícita

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



Valores devueltos en los closures

- Podemos salir de un closure de forma prematura con la sentencia return

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



Groovy como lenguaje orientado a objetos

- Clases y scripts
- Organizando nuestras clases y scripts
- Características avanzadas del modelo orientado a objetos
- GroovyBeans
- Otras características interesantes de Groovy



Groovy como lenguaje orientado a objetos

- Groovy se ha subido al carro de lenguajes altamente productivos como Perl, Python o Ruby
- Ofrece nuevas características del modelo orientado a objetos
- Tipos de datos referencia, rangos, closures y nuevas estructuras de control



Clases y scripts

- Las **clases** se declaran como en Java con la palabra reservada *class* y pueden tener *campos*, *constructores*, *inicializadores* y *métodos*.
- Los **scripts** pueden contener la definición de *variables* y *métodos*, así como la declaración de *clases*



Clases y scripts

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



Clases y scripts

- Definición del tipo de la variable es opcional
- Si no lo indicamos, debemos utilizar la palabra reservada *def* antes del nombre de la variable
- Es imposible asignar valores a variables que no coincidan en el tipo
- Autoboxing cuando sea posible



Clases y scripts

- Asignación de valores a las propiedades de las clases
 - *objeto.campo*
 - *objeto['campo']*



Clases y scripts

```
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
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
```



Clases y scripts

- La declaración de los métodos siguen los mismos criterios que las variables
- Se pueden utilizar los modificadores Java
- Es opcional utilizar la sentencia *return*
- Con la palabra reservada *def* poder evitar especificar el tipo de dato devuelto
- Por defecto, la visibilidad de los métodos es *public*



Clases y scripts

```
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(){ }
}
```



Clases y scripts

- No es necesario el método *main()*
- Utilizaremos el método *main()* para pasarle parámetros
- No es necesario especificar que el método es *public*
- No es necesario que los argumento sean del tipo *String[]*
- Se puede obviar la etiqueta *void* puesto que el método no va a devolver nada



Clases y scripts

- Método *main()* en Java

```
public static void main (String[] args)
```

- Método *main()* en Groovy

```
static main (args)
```



Clases y scripts

- Comprobación de errores: *NullPointerException*

```
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
```



Clases y scripts

- Tres tipos de protección:
 - Bloque *if* con comprobación en cortocircuito
 - Bloque *try/catch*
 - Operador *?.*



Clases y scripts

- Parámetros de los constructores
 - Por posición
 - Por nombre de la propiedad
 - Con la palabra reservada *as* y una lista de parámetros



Clases y scripts

```
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')
```



Clases y scripts

```
//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'
```



Clases y scripts

- Podemos ahorrar 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'
```



Organizando nuestras clases y scripts

- Relación entre ficheros fuentes y las clases

Fichero fuente	Archivo generado
.groovy sin declaración de clase	.script
.groovy con declaración de una clase	.class
.groovy con declaración de varias clases	.class para cada clase

- Un archivo .groovy puede tener varias clases



Organizando nuestras clases y scripts

- Organización jerárquica de **paquetes** de la misma forma que se hace en Java
- Al ejecutar no sólo se buscan los archivos `.class` sino también los `.groovy`, puesto que no es necesario compilarlos previamente
- En caso de conflicto, se opta por el más reciente



Organizando nuestras clases y scripts

- Declaración de paquetes

```
package negocio

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

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



Organizando nuestras clases y scripts

- Importar paquetes

```
import negocio.*

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

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



Organizando nuestras clases y scripts

- Alias de paquetes

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

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



Características avanzadas del modelo orientado a objetos

- **Herencia:** igual que en Java. Una clase Groovy puede extender una clase Java y viceversa
- **Interfaces:** igual que en Java. Lo más parecido a la herencia múltiple
- **Multimétodos:** Groovy elige el tipo de datos de forma dinámica al pasarlo como parámetros a los métodos de nuestras clases



Características avanzadas del modelo orientado a objetos

- Ejemplo de múltimétodos

```
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'
```



GroovyBeans

- Mismo concepto que los JavaBeans
- Convenciones en cuanto a los nombres que permiten a varias clases comunicarse entre si
- Los GroovyBeans mejoran el concepto de los JavaBeans



GroovyBeans

- Ejemplo de JavaBean

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

    public String getTitulo(){
        return titulo;
    }

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




GroovyBeans

- Ejemplo de GroovyBean

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



GroovyBeans

- Ventajas de los GroovyBeans
 - Ahorro en código fuente
 - Más fácilmente modificable
 - Posibilidad de sobrescribir los métodos *setters* y *getters*
 - No crea setters para las variables de tipo *final*
 - Posibilidad de acceder a las propiedades con el operador `.`



GroovyBeans

```
class Persona {
    String nombre, apellidos

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

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

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



GroovyBeans

- Sobreescibir los métodos *setters* y *getters*

```
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
```

```
assert 300 == doble.@valor
```



Otras características interesantes de Groovy

- Operador spread *

```
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)
```