

Groovy

Adolfo Sanz De Diego

Abril 2013

1 Acerca de

1.1 Autor

- **Adolfo Sanz De Diego**
 - Correo: asanzdiego@gmail.com
 - Twitter:
[@asanzdiego](<http://twitter.com/asanzdiego>)
 - LinkedIn: <http://www.linkedin.com/in/asanzdiego>
 - Blog: <http://asanzdiego.blogspot.com.es>

1.2 Licencia

- **Este obra está bajo una licencia:**
 - Creative Commons Reconocimiento-CompartirIgual 3.0
- **El código fuente de los programas están bajo una licencia:**
 - GPL 3.0

2 Introducción

2.1 Perspectiva general (I)

- Groovy es un **lenguaje dinámico, orientado a objetos**, muy íntimamente ligado a Java.
- Groovy **simplifica la sintaxis de Java** convirtiendo multitud de tareas en un placer.
- Groovy **comparte con Java** el mismo modelo de objetos, de hilos y de seguridad.
- Groovy puede usarse también de manera dinámica como un lenguaje de **scripting**.

2.2 Perspectiva general (II)

- El 99% del **código Java puede ser compilado con groovy**.
- El 100% del **código Groovy es convertido en bytecode Java**, y ejecutado en la JVM.
- Los programadores Java nos podemos **introducirnos en Groovy poco a poco**.
- Su **curva de aprendizaje es casi plana** para programadores Java.

2.3 HelloWorld.java

```
public class HelloWorld {  
  
    private String nombre;  
  
    public static void main(String[] args) {  
        HelloWorld hw = new HelloWorld();  
        hw.setNombre("Groovy");  
        System.out.println(hw.saluda());  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String saluda() {  
        return "Hola " + nombre;  
    }  
}
```


2.4 HelloWorld.groovy

```
public class HelloWorld {  
  
    private String nombre;  
  
    public static void main(String[] args) {  
        HelloWorld hw = new HelloWorld();  
        hw.setNombre("Groovy");  
        System.out.println(hw.saluda());  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String saluda() {  
        return "Hola " + nombre;  
    }  
}
```

2.5 HelloWorld.groovy (I)

```
public class HelloWorld {  
  
    private String nombre;  
  
    public static void main(String[] args) {  
        HelloWorld hw = new HelloWorld();  
        hw.nombre = "Groovy";  
        System.out.println(hw.saluda());  
    }  
  
    public String saluda() {  
        return "Hola " + nombre;  
    }  
}
```

- **Getters y setters autogenerados.**

2.6 HelloWorld.groovy (II)

```
class HelloWorld {  
  
    String nombre  
  
    static void main(String[] args) {  
        HelloWorld hw = new HelloWorld()  
        hw.nombre = "Groovy"  
        System.out.println(hw.saluda())  
    }  
  
    String saluda() {  
        "Hola " + nombre  
    }  
}
```

- **Clases y métodos public por defecto.**
- **Atributos private por defecto.**
- **punto y coma opcionales.**
- **return opcional** (por defecto la última línea)

2.7 HelloWorld.groovy (III)

```
class HelloWorld {  
  
    def nombre  
  
    static void main(args) {  
        HelloWorld hw = new HelloWorld()  
        hw.nombre = "Groovy"  
        println hw.saluda()  
    }  
  
    def saluda() {  
        "Hola " + nombre  
    }  
}
```

- **tipado dinámico.**
- **println** alias de System.out.println.
- **paréntesis opcionales** cuando hay parámetros.

2.8 HelloWorld.groovy (IV)

```
HelloWorld hw = new HelloWorld()
hw.setNombre "Groovy"
println hw.saluda()

class HelloWorld {

    def nombre

    def saluda() {
        "Hola ${nombre}"
    }
}
```

- **declaración de classes y método main opcional en scripts.**

2.9 Otros aspectos (I)

- **imports por defecto** (`java.io.`, *java.lang.*, `java.math.BigDecimal`, `java.math.BigInteger`, `java.net.`, *java.util.*, `groovy.lang.`, *groovy.util.*)
- Todas las **excepciones son de tipo `RuntimeException`** (ni declararlas, ni capturarlas)
- **Sobrecarga de operadores** (plus, minus, multiply, div, mod, or, and, next, previous, ...) [ver documentación](#)

2.10 Otros aspectos (II)

- Todos **los tipos primitivos son tratados como objetos** (autoboxing)
- Todos **los decimales son tratados como `BigDecimal`** para evitar la inexactitud de las clases `Float` y `Double`.
- Siempre que se evalúe un **valor cero, null, un `String` vacío, una colección vacía, un array de longitud cero o un `StringBuilder/StringBuffer` vacío, se obtendrá `false`**. En cualquier otra situación, se obtendrá `true`.

2.11 Otros aspectos (III)

- El **operador ==** está **sobrecargado** en el método `equals()`
- El `==` de Java se puede usar con el método `is()`
- Uso de la **anotación @PackageScope** para la **visibilidad package-scope**.

2.12 Otros aspectos (IV)

- Uso de **maps** en los constructores.

```
class Persona {  
  def nombre  
  def edad  
}  
  
def persona = new Person(nombre: "Alba", edad: 5)
```

2.13 Otros aspectos (V)

- **Operador referencia segura (?.)** que se pone antes del punto y que si el objeto es null devuelve null y en caso contrario devuelve lo que sigue al punto.

```
if (order != null) {  
    if (order.getCustomer() != null) {  
        if (order.getCustomer().getAddress() != null) {  
            System.out.println(order.getCustomer().getAddress());  
        }  
    }  
}
```

- El código anterior se simplifica:

```
println order?.customer?.address
```

2.14 Otros aspectos (VI)

- **Operador Elvis (?:)** para valores por defecto.

```
def result = name != null ? name : "Unknown"
```

- El código anterior se simplifica:

```
def result = name ?: "Unknown"
```

2.15 Otros aspectos (VII)

- Uso intensivo de **asserts**.

```
def check(String name) {  
    // name non-null, non-empty and size > 3  
    assert name?.size() > 3  
}
```

3 Instalación y configuración

3.1 JDK

1. Descargar.
2. Instalar/Descomprimir.
3. Variable de entorno y añadir al path.

```
export JAVA_HOME=~/.Java/jdk"  
export PATH=$PATH:"$JAVA_HOME/bin"
```

3.2 Groovy-SDK

1. Descargar.
2. Instalar/Descomprimir.
3. Variable de entorno y añadir al path.

```
export GROOVY_HOME=~/.Java/groovy  
export PATH=$PATH:$GROOVY_HOME/bin
```

3.3 Probando

```
$ groovy --version  
Groovy Version: 2.1.2  
JVM: 1.7.0_21 Vendor: Oracle Corporation OS: Linux
```


4 Ejecución

4.1 Scripts

- **Ejecución directa** (compila a .class y ejecuta directamente)

```
$ groovy HelloWorld.groovy
```

4.2 Groovy Shell

- Se abre una **shell de groovy**, con historial de comandos.

```
$ groovysh
```

4.3 Groovy Console

- Funciona en **modo gráfico** y permite opciones mas potentes que la shell, como guardar y cargar archivos, opciones de edición de texto, etc.

```
$ groovyConsole
```

4.4 Compilación

```
$ groovyc helloworld.groovy
```

- Luego se puede ejecutar directamente el .class

```
$ groovy helloworld
```

5 Cadenas de texto

5.1 Strings

- Similares a las de Java pero **podemos usar tanto comillas dobles como simples.**
- Las comillas simples no interpreta las variables, las dobles sí

5.2 GStrings

- Contienen **expresiones embebidas**.
- Las expresiones se introducen con `${}` y son **evaluadas en tiempo de ejecución**.

```
def saldo = 1821.14  
def mensaje = "El saldo a fecha ${new Date()} es de ${saldo} euros"  
println mensaje
```


5.3 Heredocs

- Se forma con **tres comillas simples o dobles**.
- Nos permiten cadenas de **texto multilinea**.
- Nos permiten también **mezclar comillas simples y dobles en su interior**.

```
def multilinea = """  
Primera linea  
Segunda linea  
Tercera linea con "comillas dobles" y 'comillas simples'  
"""  
println multilinea
```

6 Closures

6.1 Definición

- **Bloque de código autónomo** que puede usarse en distintos sitios.

```
def saludar = { nombre, apellido ->
  println "¡Hola ${nombre} ${apellido}!"
}
saludar "Alba", "Sanz"
```

6.2 Curry

- Nos permite **pre-cargar valores** que serán siempre los mismos para una determinada función.

```
def multiplicar = { valor1, valor2 ->
  valor1 * valor2
}

def doble = multiplicar.curry(2)
def triple = multiplicar.curry(3)

println doble(7)
println triple(7)
```

6.3 Parámetros

- Las closures pueden ser utilizadas como **argumentos de una función.**

```
def repetirClosure(int numRepeticiones, Closure closure) {  
    for(int i = 0; i < numRepeticiones; i++) {  
        closure.call(i)  
    }  
}  
  
def closure = { println it }  
repetirClosure(5, closure)
```

7 Rangos

7.1 Numéricos

- Imprime 5 números del 1 al 5 inclusive.

```
(1..5).each {  
  println it  
}
```

- Imprime 4 números del 1 al 4.

```
(1..<5).each {  
  println it  
}
```

7.2 Otros

- Fechas

```
def hoy = new Date()
def dentroDeSieteDias = hoy + 7
(hoy..dentroDeSieteDias).each { dia ->
    println dia
}
```

- Letras

```
('a'..'z').each { letra ->
    println letra
}
```


7.3 Propiedades

- Algunos atributos y métodos:

```
def rango = 5..10  
println rango.from  
println rango.to  
println rango.contains(4)  
println rango.size()  
println rango.get(3)  
println rango[3]
```

7.4 Switchs

```
def sueldo = 1700;  
switch(sueldo) {  
  case 600..<1200:  
    println 'nivel 1'  
    break  
  case 1200..<1800:  
    println 'nivel 2'  
    break  
}
```

8 Listas

8.1 Añadir

- Se añaden por **índice**:

```
def paises = ["España", "Mexico"]  
paises << "Argentina"  
paises.add("Ecuador")
```

8.2 Recuperar/Modificar

- Se recuperan también por **índice**:

```
países[3] = "Colombia"  
países[6] = "Ecuador"  
  
println países[0]  
println países.getAt(1)  
println países
```

8.3 Eliminar

- Lanza un **NullPointerException** si no existe nada en el índice 6.

```
def eliminado1 = paises.remove(6)
```

- Devuelve un **null** si no hay ningún objeto coincidente.

```
def eliminado2 = paises.remove("Ecuador")
```

- Elimina el objeto con el **índice más alto**.

```
def eliminado = paises.pop()
```

8.4 Iterar

- **Iterar:**

```
países.each {  
  println it.toUpperCase()  
}
```

- **Iterar con índice:**

```
países.eachWithIndex { país, índice ->  
  println "${país} se encuentra en la posición ${índice}"  
}
```

- **Iterar sobre cada elemento y devolver otra lista:**

```
def paísesMayusculas = países.collect { país ->  
  país.toUpperCase()  
}
```

8.5 Ordenar

- **Ordenar** la lista original:

```
países.sort()
```

- Devuelve una lista **invertida**, sin modificar la original:

```
def paísesInvertidos = países.reverse()
```


8.6 Operadores + y -

- El operador `+=` y el operador `-=`

```
def pares = [2, 4, 6, 8]
def impares = [1, 3, 5, 7, 9]

pares += impares
pares.sort()
println pares

pares -= impares
println pares
```

8.7 Max y min

- Valores **máximo y mínimo**:

```
println letras.max()  
println letras.min()
```

8.8 Aplanar

- La función **flatten** aplanar una lista anidada:

```
['a', ['c', 'd'], 'f'].flatten() == ['a', 'c', 'd', 'f']
```

8.9 Join y disjoint

- La función **join** convierte la lista en el String a-b-c

```
def letras = ['a', 'b', 'c']  
println letras.join("-")
```

- La función **disjoint** nos devuelve **true** si las 2 listas son **disjuntas**:

```
['a', 'c', 'd'].disjoint(['b', 'e', 'f']) == true
```

8.10 Intersección

- La función **intersect** nos devuelve los **elementos comunes** entre 2 listas:

```
['a', 'c', 'd'].disjoint(['b', 'c', 'd']) == ['c', 'd']
```

8.11 Unicidad

- La función **unique** quita los duplicados:

```
['a', 'c', 'a', 'd'].unique() == ['a', 'c', 'd']
```

8.12 Búsqueda

- La función **find**, que admite una closure, devuelve el primer elemento encontrado:

```
[1, 2, 3, 4].find { it % 2 == 0 } == 2
```

- La función **findAll**, que admite una closure, devuelve todos los elementos encontrados:

```
[1, 2, 3, 4].findAll { it % 2 == 0 } == [2, 4]
```

8.13 Sumatorio

- La función **sum**, que admite una closure, devuelve la suma de los elementos:

```
[1, 2, 3, 4].sum() == 10  
[1, 2, 3, 4].sum { it % 2 == 0 } == 6
```

- El operador *****, que ejecuta un método del objeto para todos los objetos de la lista:

```
class Persona  
  def nombre  
  def imprimir() {  
    println nombre  
  }  
}  
  
def personas = [new Persona(nombre:"Alba"), new Persona(nombre:"Laura")]  
personas*.imprimir()
```


9 Mapas

9.1 Añadir

- Se añaden pares **clave-valor**:

```
def capitales = ['Madrid':'España', 'Mexico D.F.': 'Mexico']  
capitales.put('Buenos Aires', 'Argentina')
```

9.2 Recuperar/modificar

- Se modificar/recuperar los pares **clave-valor**:

```
capitales.get('Madrid')  
capitales.Madrid  
capitales['Madrid']  
capitales['Buenos Aires'] = Argentina  
capitales['Buenos Aires'] = 'Argentina'
```

9.3 Eliminar

- **Eliminar:**

```
capitales.remove('Buenos Aires')
```

9.4 Iterar

- **Iterar:**

```
capitales.each { capital, pais ->  
  println "La capital de ${pais} es ${capital}"  
}
```

9.5 Operadores + y -

- Operador +=:

```
def angloParlantes = ['Washington':'EEUU', 'Londres':'Reino Unido']  
capitales += angloParlantes
```

- El operador -= no está soportado, en su caso:

```
angloParlantes.each {  
  capitales.remove(it.key)  
}
```

9.6 Keys y Values

- Las funciones **keySet()** y **values()** (similares Java):

```
def claves = capitales.keySet()  
def valores = capitales.values()
```

- Las funciones **containsKey()** y **containsValue()**:

```
println capitales.containsKey('Madrid')  
println capitales.containsValue('España')
```

10 Meta Programación

10.1 ¿Qué es y para qué sirve?

- Mediante metaprogramación podemos escribir **código que genera o modifica otro código o incluso a si mismo en tiempo de ejecución.**
- Esto nos permite, entre otras cosas, manejar situaciones que no estaban previstas cuando se escribió el código, sin necesidad de recompilar.

10.2 Reflection

- Mediante **reflection** podemos acceder a los miembros de una clase:

```
println String.class
String.interfaces.each { println it }
String.constructors.each { println it }
String.methods.each { println it }

def s = new String("cadena de texto")
s.properties.each { propiedad ->
  println propiedad
}
```

10.3 Expandos

- Un Expando es como un **objeto en blanco**, al cual podemos añadir métodos y propiedades.

```
def posicion = new Expando()  
posicion.latitud = 15.47  
posicion.longitud = -3.11  
posicion.mover = { deltaLatitud, deltaLongitud ->  
    posicion.latitud += deltaLatitud  
    posicion.longitud += deltaLongitud  
}
```

10.4 Propiedades

- **metaClass.hasProperty()** nos permite comprobar si dispone de una propiedad concreta:

```
def boligrafo = new Artículo(  
    descripcion:"Boligrafo negro", precio:0.45)  
  
if(boligrafo.metaClass.hasProperty(boligrafo, "precio")) {  
    // hacer algo  
}
```

10.5 Propiedades dinámicas

- Para poder añadir propiedades a un objeto (que no sea un `expando`) de forma dinámica se hace con las funciones **`setProperty()`** y **`getProperty()`**

```
class Artículo {
    String descripcion
    double precio
    def propiedades = [:]

    void setProperty(String nombre, Object valor) {
        propiedades[nombre] = valor
    }

    Object getProperty(String nombre) {
        propiedades[nombre]
    }
}

def articulo = new Artículo()
articulo.codigoEAN = 84123445593
println articulo.codigoEAN
```

10.6 Punteros

- Punteros a **propiedades** con el **operador @**:

```
println boligrafo.@precio
```

- Punteros a **métodos** con el **operador &**:

```
def lista = []  
def insertar = lista.&add  
insertar "valor1"  
insertar "valor2"
```

10.7 Categorías

- Groovy nos permite **usar métodos de una categoría** dentro de una clase.

```
class Artículo {  
    String descripcion  
    double precio  
}  
  
class ArtículoExtras {  
    // importante el static y la clase Artículo como parámetro  
    static double conImpuestos(Artículo artículo) {  
        artículo.precio * 1.18  
    }  
}  
  
Artículo artículo = new Artículo(descripción:'Grapadora', precio:4.50)  
use(ArtículoExtras) {  
    artículo.conImpuestos()  
}
```

10.8 Métodos

- **metaClass.respondsTo()** nos permite comprobar la existencia de un método:

```
if(boligrafo.metaClass.respondsTo(boligrafo, "getDescripcion")) {  
    // hacer algo  
}
```


10.9 Ejecutando con GStrings

- Podemos **ejecutar métodos mediante GStrings**:

```
def nombreDelMetodo = "getPrecio"  
boligrafo."${nombreDelMetodo}"()
```

- Esto nos permite ejecutar en tiempo de ejecución métodos no creados en tiempo de compilación.

10.10 Interceptando métodos (I)

- Para poder añadir métodos a un objeto (que no sea un expando) de forma dinámica se hace con la función **invokeMethod()**

```
class Artículo {  
    String descripcion  
    double precio  
  
    Object invokeMethod(String nombre, Object args) {  
        println "Invocado método ${nombre}() con los argumentos ${args}"  
    }  
}  
  
def articulo = new Artículo()  
articulo.operacionInexistente('abc', 123, true)
```

10.11 Interceptando métodos (II)

- El caso anterior sólo intercepta los métodos no definidos.
- Si lo que queremos es interceptar todos los métodos, la clase tiene que implementar la Interfaz **GroovyInterceptable**
- Esto nos permite Programación Orientada a Aspectos.

```
class Artículo implements GroovyInterceptable {  
    String descripcion  
    double precio  
  
    Object invokeMethod(String nombre, Object args) {  
        def metaMetodo = Artículo.metaClass.getMetaMethod(nombre, args)  
        metaMetodo.invoke(this, args)  
    }  
}
```

10.12 Métodos dinámicos

- Groovy nos permite **añadir métodos** a una clase ya creada:

```
Integer.metaClass.numeroAleatorio = {  
    def random = new Random()  
    random.nextInt(delegate.intValue())  
}  
  
50.numeroAleatorio()
```

- Tened en cuenta que **delegate** hace referencia al objeto 'delegado', el objeto que estará disponible en tiempo de ejecución.

11 Ficheros

11.1 Listados

- Podemos **iterar** de forma sencilla sobre los ficheros y directorios:

```
def directorio = new File(".")

// imprimimos todo
directorio.listFiles { println it }

// imprimimos los subdirectorios
directorio.listFiles { println it }

// imprimimos los subdirectorios recursivamente
directorio.listFilesRecursively { println it }

// imprimimos los subdirectorios que contengan b
directorio.listFiles { println it }
```

11.2 Escritura

- **Sobreescribe** el fichero:

```
def file = new File('datos.dat')
file.write """
Hola
Todo bien?
Adios
"""
```

- **Añade** al final del fichero:

```
def file = new File('datos.dat')
file << "P.D. Un beso"
file.append "Otro beso"
```

11.3 Lectura

- Lee **todo el texto**:

```
def file = new File('datos.dat')  
println file.text
```

- Lee el texto **línea a línea**:

```
def file = new File('datos.dat')  
file.eachLine { println "->$it" }
```


11.4 Tamaños

- Para sacar el tamaño **de los ficheros y de las particiones** es parecido a Java:

```
// tamaño en bytes  
println file.size()  
  
// bytes libres en la partición actual  
println file.getFreeSpace()  
  
// bytes disponibles en la máquina virtual  
println file.getUsableSpace()  
  
// tamaño total en bytes de la partición actual  
println file.getTotalSpace()
```

11.5 Propiedades

- Al igual que en Java podemos acceder a las **propiedades de los archivos**:

```
file.exists()  
file.isFile()  
file.canRead()  
file.canWrite()  
file.isDirectory()  
file.isHidden()
```

11.6 Creación

- Crear un **fichero**:

```
def file = new File("kkk.txt")  
file.createNewFile()
```

- Crear un **fichero temporal**:

```
File.createTempFile("kkk", "txt")
```

- Crear **directorios**:

```
def dir = new File("kk1/kk2")  
dir.mkdirs()
```

11.7 Borrado

- Para borrar **tanto ficheros como directorios**:

```
file.delete()
```

12 XML

12.1 Builders

- Groovy utiliza las listas y los mapas para **parsear datos** de forma sencilla.
- Aunque podemos crear los nuestros, Groovy viene ya con varios Builders:
 - **NodeBuilder** - navegación mediante XPath
 - **DOMBuilder** - navegación mediante DOM
 - **SAXBuilder** - navegación mediante SAX
 - **MarkupBuilder** - documentos de XML / HTML
 - **AntBuilder** - tareas Ant
 - **SwingBuilder** - interfaces Swing

12.2 Escritura XML (I)

- Utilizamos **MarkupBuilder**:

```
def writer = new StringWriter()
def builder = new MarkupBuilder(writer)
builder.setDoubleQuotes true
builder.personas{
  persona(id:"1"){
    nombre "Adolfo"
    edad 35
  }
  persona(id:"2"){
    nombre "Alba"
    edad 25
  }
}
println writer.toString()
```

12.3 Escritura XML (II)

- El fichero generado:

```
<personas>  
  <persona id="1">  
    <nombre>Adolfo</nombre>  
    <edad>35</edad>  
  </persona>  
  <persona id="2">  
    <nombre>Alba</nombre>  
    <edad>25</edad>  
  </persona>  
</personas>
```


12.4 Lectura XML

- **XmlParser** lee todo el documento y genera en memoria una estructura parecida al DOM.
- Es más cómodo y rápido una vez leído, pero necesita más memoria RAM.

```
def personas=new XmlParser().parse("personas.xml")  
personas.each { println it }
```

- **XmlSlurper** hace lectura directa y es más rápido en la primera lectura.
- Viene bien para hacer búsquedas en ficheros grandes.

```
def personas=new XmlSlurper().parse("personas.xml")  
personas.each { println it }
```

13 Plantillas

13.1 La plantilla

- En Groovy podemos usar **plantillas** de este estilo:

```
<html>
<head>
  <title>Informe del ${String.format("%tA",fecha)}</title>
</head>
<body>
  Estimado ${usuario?.nombre} ${usuario?.apellidos}
  bla,bla,bla,...
</body>
</html>
```

13.2 Parseo

- Y podemos **parsearlas** de forma sencilla:

```
def plantilla=this.class.getResource("plantillaEmail.gtpl")
def datos=[
  "usuario": new Usuario(nombre:"pepe", apellidos:"perez"),
  "fecha":new Date()]
def procesador=new SimpleTemplateEngine()
def correo=procesador.createTemplate(plantilla).make(datos);
println correo.toString()
```

14 Expresiones regulares

14.1 Uso

- Las expresiones regulares se encierran con la barra (/).
- Con la virgulilla (~) se hacen las comparaciones.

```
a?    -> 0 o 1 'a'  
a*    -> 0 o muchas 'a'  
a+    -> 1 0 muchas 'a'  
a|b   -> 'a' o 'b'  
.  
[1-9] -> cualquier número  
[^13] -> cualquier número excepto el '1' y el '3'  
^a    -> empieza por 'a'  
a$    -> termina por 'a'
```

14.2 Ejemplos I

- Ejemplos **sencillos**:

```
// igual a abc
```

```
assert "abc" ==~ /abc/
```

```
// empieza por ab
```

```
assert "abcdef" ==~ /^ab.*/
```

```
// termina por ef
```

```
assert "abcdef" ==~ /.ef$/
```

14.3 Ejemplos II

- Ejemplos algo **más elaborados**:

```
// empieza por a termina por d y tiene en medio una b o una c
```

```
assert "abd" ==~ /^a[b|c]d$/
```

```
// empieza por a termina por d y tiene en medio cualquier carácter
```

```
assert "acd" ==~ /^a.?d$/
```

```
// una o varias a y luego b
```

```
assert "aab" ==~ /a+b/
```


15 Fechas

15.1 Hoy

- Cuando creamos un objeto se crea con **la fecha y la hora actual**:

```
def today = new Date()
```

15.2 Sumar/Restar

- Podemos **añadir y sustraer** fechas de forma sencilla:

```
def tomorrow = today + 1,  
  dayAfter  = today + 2,  
  yesterday = today - 1,  
  dayBefore = today - 2  
println "dayBefore = $dayBefore"  
println "yesterday = $yesterday"  
println "today      = $today"  
println "tomorrow  = $tomorrow"  
println "dayAfter  = $dayAfter"
```

15.3 Comparaciones

- Podemos hacer **comparaciones**:

```
println "tomorrow.after(today) = " + tomorrow.after(today)
println "yesterday.before(today) = " + yesterday.before(today)
println "tomorrow.compareTo(today) = " + tomorrow.compareTo(today)
println "tomorrow.compareTo(dayAfter) = " + tomorrow.compareTo(dayAfter)
println "dayBefore.compareTo(dayBefore) = " + dayBefore.compareTo(dayBefore)
```

15.4 Formateo

- Podemos **formatear fechas** de forma sencilla:

```
// YYYY/mm/dd  
println String.format('Hoy es %tY/%<tm/%<td', today)  
  
// HH:MM:SS.LLL  
println String.format('La hora es %tH:%<tM:%<tS.%<tL', today)
```

15.5 Parseo

- Tambien podemos **parsear cadenas** en fechas de forma sencilla:

```
def date = Date.parse("yyyy/MM/dd HH:mm:ss", "2013/05/01 11:12:13")
```