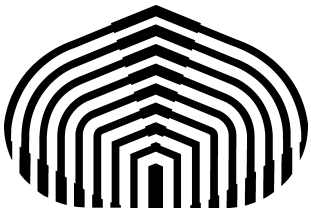

UNIVERSIDAD SIMÓN BOLÍVAR
INGENIERÍA DE LA COMPUTACIÓN
DEPT.COMPUTACIÓN Y TECNOLOGÍA DE LA INFORMACIÓN
CI3641: LENGUAJES DE PROGRAMACIÓN I

Ensayo / Proyecto:
Groovy

PROFESOR:

Ricardo Monascal
.

ALUMNOS:

Andres A. Buelvas V. 13-10184
Miguel C. Canedo R. 13-10214

Índice

Características de Groovy

Alcance

Groovy al igual que la gran mayoría de los lenguajes y siendo Java la base de su desarrollo, posee un **Alcance Estático** o también conocido como Alcance Léxico. Esto quiere decir, que el enlace entre los nombres y los objetos puede ser determinado en tiempo de compilación.

En el siguiente ejemplo, donde usamos unos métodos especiales de Groovy llamados **Closures**, se puede observar que Groovy no solo posee Alcance Estático sino que también posee Asociación Profunda, es decir las clausuras de los Closures se crean al ser pasados como parámetros.

```
u = 1
P = { T, x ->

    def Z = { y ->
        u = x + y
    }

    def u = x + 1

    if (x <= 2){
        T(x*2)
    }
    else{
        P(Z, x-1)
    }
}

// Main
P(P, 4)

assert u == 7
```

Palabra clave '*def*'

El uso de esta palabra es un azúcar sintáctico para scripts básicos de Groovy. Omitir la palabra clave **def** coloca la variable en los enlaces del script actual y Groovy la tratara (en su mayoría) como una variable con ámbito global:

```
x = 1
assert x == 1
assert this.binding.getVariable("x") == 1 // Metodo para observar los
  Enlaces del Script.
```

Pero el uso de la palabra clave **def**, por otro lado, no coloca la variable en los enlaces del script:

```
def y = 2

assert y == 2

try {
    this.binding.getVariable("y")
} catch (groovy.lang.MissingPropertyException e) {
    println "error capturado"
}
```

Esto imprimirá: “error capturado”.

Ademas, si defines un método en tu script, dentro de él no podrás acceder a las variables que hayas declarado con **def** en el cuerpo principal del script ya que dichas variables no estarían solo dentro del alcance del método.

```
x = 1
def y = 2


public bar() {
    assert x == 1

    try {
        assert y == 2
    } catch (groovy.lang.MissingPropertyException e) {
        println "error capturado"
    }
}

bar()
```

Esto también imprimirá: “error capturado”.

Estructuras de Control de Flujo

Secuenciación

La secuenciación es vital cuando se habla de lenguajes imperativos. Es el principal medio para controlar el orden en que ocurren los efectos secundarios, esta se presenta como una sección de código con una o más declaraciones y sentencias. Cuando una declaración sigue a otra en el texto del programa, la primera instrucción se ejecuta antes del segundo.

En el caso de Groovy, sigue la misma estructuración de bloques de código de Java, es decir; puede seguir el mismo principio de encapsulamiento a través del **{...}**:

```
def secuencia(){
    a = 1
    b = 2
    c = 3
    println a
    println b
    println c
    println a + b + c
}
```

Nota: Todo bloque de código que se encuentre en un script, es considerado una clase por Groovy, por lo tanto también sigue la misma sintaxis de **{...}**.

Selección

- **if-else:** Groovy soporta la usual sintaxis *if* - *else* que posee Java, siendo opcional la declaración *else*.

```
def x = false
def y = false

if ( !x ) {
    x = true
}

assert x == true

if ( x ) {
    x = false
} else {
    y = true
}

assert x == y
```

Ademas, Groovy soporta el anidamiento sintáctico de declaraciones *if*, que posee Java, mediante la declaración *else if*.

```
if ( ... ) {
    ...
} else if (...) {
    ...
} else {
    ...
}
```

- **switch / case:** La sentencia *switch* en Groovy es compatible con el código de Java; por lo que puede pasar por casos que comparten el mismo código para múltiples coincidencias. Sin embargo, una diferencia es que la declaración *swith* de Groovy puede manejar cualquier tipo de valor de switch y se pueden realizar diferentes tipos de emparejamientos.

```
def x = 1.23
def result = ""

switch ( x ) {
    case "foo":
        result = "found foo"
        // Sin el 'break', accedera a los siguientes casos.

    case "bar":
        result += "bar"

    case [4, 5, 6, 'inList']:
        result = "list"
        break

    case 12..30:
        result = "range"
        break

    case Integer:
        result = "integer"
        break

    case Number:
        result = "number"
        break

    case ~/fo*/: // Representacion toString() de x coincide con el
   patron?
        result = "foo regex"
        break
```

```
    case { it < 0 }: // o lo que es igual, { x < 0 }
        result = "negative"
        break

    default:
        result = "default"
}

assert result == "number"
```

El *switch* de Groovy admite los siguientes tipos de comparaciones:

- Los valores de *cases* que son Clase coinciden si el valor del *switch* es una instancia de la Clase.
- Los valores de *cases* que son Expresiones Regulares coinciden si la representación toString() del valor del *switch* coincide con la Expresión Regular.
- Los valores de *cases* que son Colecciones coinciden si el valor del *switch* está contenido en la Colección. Esto también incluye rangos (ya que son Listas).
- Los valores de *cases* que son Closures coinciden si la llamada al Closure arroja un resultado que es verdadero de acuerdo con la verdad Groovy.
- Si no se utiliza ninguno de los anteriores, el valor del *case* coincide si el valor del *case* es igual al valor del *switch*.

Cuando se utiliza un Closure en los valores de case, el parámetro por defecto **it** es en realidad el valor del switch.

Repetición

Groovy acepta el estándar de Java / C para la realización de bucles.

- **Indeterminadas:** En esta categoría entran las repeticiones controladas por lógica. En este caso se conseguirá el típico bucle **while** que esta presente en casi todos los lenguajes de programación de alto nivel:

```
def x = 0
def i = 0
while (i < 5) {
    x += i
    i++
}
assert x == 10
```

- **Determinadas:** En esta categoría entran las repeticiones controladas por enumeración.
  - **Clásico for loop**

```
def x = 0
for (int i = 0; i < 5; i++){
    x += i
}
assert x == 10
```

- **for in loop o for each**

```
def x = 0
for ( i in 0..4){
    x += i
}

assert x == 10
```

- **.each()**: Trabaja de manera simular al **for in loop/each**.

```
(1..3).each {
    println "Number ${it}"
}
```

- **iterator**: Se puede usar el operador de subíndices para la clase **Iterator** a través del método **getAt ()**, todo esto se hace a través de la función **iterator()**.

```
def list = ['Groovy', 'Grails', 'Griffon', 'Gradle']
def iterator = list.iterator()

assert 'Groovy' == iterator[0]
assert 'Gradle' == iterator[-1]
assert !iterator[1]  // Iterator is exhausted.

iterator = list.iterator()  // Get new iterator.
def newList = []
(0..<list.size()).each {
 newList << iterator[0]  // Index 0 is next element.
}
assert 'Groovy,Grails,Griffon,Gradle' == newList.join(',')
```

Abstracción Procedural

La llamada a métodos de Groovy admite que un método se llame a si mismo, es decir, Groovy admite recursiones. Un ejemplo básico para visualizar las recursiones en Groovy seria el calcular el Factorial de un número.

```
def factorial(n){
    if(n <= 1){
        return n
    }
    n * factorial(n-1)
}
```

Hay ciertos casos especiales de recursiones, las cuales tienen a ser mas eficientes siempre y cuando el lenguaje logre detectar que el método posee las características necesarias, estas se llaman **Recursiones de Cola**. Groovy por si solo no logra detectar que un método posea **Recursion de Cola** pero desde la versión 1.8 se da la opción al programador que coloque una etiqueta al método para que el compilador trate al mismo como una **Recursion de Cola**.

```
import groovy.transform.TailRecursive

@TailRecursive
def factorialCola(n, acum = 1){
    if(n != 1){
        return factorialCola(n-1, acum*n)
    }
    acum
}
```

Manejador de Excepciones

Como gran parte de Groovy, el manejo de excepciones también es heredado de Java. Se puede especificar 3 tipos de bloques **try-catch-finally**, **try-catch** o **try-finally**. Es importante acotar que las llaves son importantes para delimitar los cuerpos de los bloques.

```
try {
    'moo'.toLong()    // Esta linea generara una excepcion.
    assert false      // Esta afirmacion a este punto no es alcanzable.
} catch ( e ) {
    assert e in NumberFormatException
}
```

Podemos poner código dentro de una cláusula 'finally' después de una cláusula 'try' coincidente, de modo que independientemente de si el código en la cláusula 'try' arroja una excepción, el código en la cláusula 'finally' siempre se ejecutará:

```
def z
try {
   def i = 7, j = 0
   try {
       def k = i / j
       assert false          // Nunca sera alcanzado por la excepcion de
  la linea previa.
   } finally {
       z = 'reached here'  // Siempre se ejecuta incluso si se lanza una
    excepcion.
   }
} catch ( e ) {
   assert e in ArithmeticException
   assert z == 'reached here'
}
```

Desde la version 2.0 de Groovy, podemos definir varias excepciones para ser atrapadas y tratadas por el mismo bloque catch:

```
try {
   /* ... */
} catch ( IOException | NullPointerException e ) {
   /* un bloque para manejar 2 excepciones */
}
```

Asserts

A pesar de que comparte la palabra clave **assert** con Java, esta tiene un comportamiento diferente dentro de Groovy. En primer lugar, un **assert** en Groovy siempre se ejecuta, independientemente del flag **-ea** de la JVM. Esto hace a los **assert** una elección de primera clase para pruebas unitarias.

Un **assert** dentro de Groovy se descompone en 3 partes:

```
assert [left expression] == [right expression] : (optional message)
```

Si el **assert** evalúa a verdad, entonces no sucederá nada. Pero si llegara a evaluar falso, entonces se provee una representación visual de los valores de cada sub-expresión de la expresión completa dentro del **assert**. Por ejemplo:

```
assert 1+1 == 3
```

Este ejemplo dará lugar a:

```
Caught: Assertion failed:

assert 1+1 == 3
       |   |
       2   false
```

Concurrencia

Como con Groovy se tiene la facilidad de poder usar las librerías que posee Java, esto permite que se puedan trabajar con Hilos como se trabajarían normalmente en Java.

Pero ademas Groovy posee un paquete propio que permite trabajar con concurrencia, dicho paquete es **GPars**. GPars, o como es conocido formalmente *GParallelizer*, ofrece un marco para el manejo simultáneo y asincrónico de tareas mientras salvaguarda valores mutables, utilizando algunos de los mejores conceptos de los lenguajes emergentes y para implementarlos en Groovy.

- **Actores:** Los actores se pueden generar rápidamente para consumir y enviar mensajes entre ellos, incluso entre máquinas distribuidas. Lográndose construir un modelo de simultaneidad basado en mensajes con actores que no estén limitados por la cantidad de subprocesos. Los actores realizan tres operaciones diferentes: enviar mensajes, recibir mensajes y crear nuevos actores. Se crean nuevos actores con el método **actor()** que pasa en el cuerpo del actor como un parámetro de Closure. Dentro del cuerpo del actor se usa **loop()** para iterar, **react()** para recibir mensajes y **reply()** para enviar un mensaje al actor, que ha enviado el mensaje actualmente procesado. A continuación se muestra cómo crear un actor que imprima todos los mensajes que recibe:

```
import static groovyx.gpars.actor.Actors.*

def console = actor {
    loop {
        react {
            println it
        }
    }
}
```

- **Asincronizador:** Una característica principal de GPars es la clase **Asynchronizer**, que ejecuta tareas de forma asincrónica en segundo plano. Dentro de los bloques **Asynchronizer.doParallel()**, se pueden agregar a los Closures métodos asíncronos. **async()** crea una variante del Closure suministrado retornando un **Future** para el valor de retorno potencial cuando sea invocado. **callAsync()** llama a un Closure en Hilo separado que proporciona los argumentos dados y también devuelve un Future para el valor de retorno potencial. Aquí hay un ejemplo de uso de Asynchronizer:

```
Asynchronizer.doParallel() {
    Closure longLastingCalculation = {calculate()}

    Closure fastCalculation = longLastingCalculation.async()

    Future result=fastCalculation()

    println result.get()
}
```

- **Paralelizador:** Finalmente, está el **Parallelizer**, que es un procesador de recolección concurrente. El patrón común para procesar colecciones toma elementos secuencialmente, uno a la vez. Sin embargo, este algoritmo no funcionará bien en hardware multi-core. El método **min()** en un chip de doble núcleo solo puede aprovechar el 50 % de la potencia de cálculo, el 25 % para un núcleo cuádruple. En cambio, GPars usa una estructura similar a un árbol para el procesamiento paralelo. Aquí hay un ejemplo de uso:

```
doParallel {
    def selfPortraits = images.findAllParallel{it.contains me}.
    collectParallel {it.resize()}

    def smallestSelfPortrait = images.parallel.filter{it.contains me
    }.map{it.resize()}.min{it.sizeInMB}
}
```

No Determinismo

Existen pocas herramientas que implementen el No Determinismo dentro Groovy, algunas de ellas son sencillamente el hacer uso de Clases como Random para cuando se necesite escoger un numero de manera arbitraria. Estas herramientas son:

- **Clase Random:** Es una clase que hereda del lenguaje Java, la cual se instancia con la finalidad de generar un conjunto de números pseudoaleatorios. Para obtener dichos números utilizamos el método de instancia **nextInt()**.

```
Random random = new Random()
println random.nextInt()
```

Ademas, existe un Metodo de la clase **Math** el cual realiza el mismo trabajo que el método nextDouble de la clase Random.

```
Math.random()
```

- **Método hashCode():** Este método usado en instancia de la clase Object tiene un comportamiento no determinista, ya que generalmente se implementa convirtiendo la dirección interna del objeto en un entero y las direcciones son No Deterministas.
- **Recolector de Basura:** No es una herramienta de Groovy que podamos usar directamente, pero cabe destacar que el Recolector de Basura que Groovy hereda de Java tiene una naturaleza No Determinista ya que no se puede predecir que direcciones de memoria serán o no liberados en una ejecución.

Tipos de Datos

Tipos primitivos y Clases envolventes

Tipos primitivos	Wrapper Classes
boolean	Boolean
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double
byte	Byte
	BigInteger
	BigDecimal

```
class Example {
  static void main(String[] args) {
     //Example of a int datatype
     int x = 5;

     //Example of a long datatype
```

```
        long y = 100L;

        //Example of a floating point datatype
        float a = 10.56f;

        //Example of a double datatype
        double b = 10.5e40;

        //Example of a BigInteger datatype
        BigInteger bi = 30g;

        //Example of a BigDecimal datatype
        BigDecimal bd = 3.5g;

        println(x);
        println(y);
        println(a);
        println(b);
        println(bi);
        println(bd);
    }
}
```

Y la salida será:

```
100
10.56
1.05E41
30
3.5
```

Tipos compuestos

- **Listas**: Groovy usa una lista de valores separados por comas y rodeada de corchetes, para denotar listas. Las listas de Groovy son simples JDK **java.util.List**, ya que Groovy no define sus propias clases de colección. La implementación de la lista concreta utilizada al definir los literales de lista es **java.util.ArrayList** de forma predeterminada.

```
def numbers = [1, 2, 3]             // (1)

assert numbers instanceof List      // (2)
assert numbers.size() == 3          // (3)
```

1. Se define un número de lista delimitado por comas y rodeado por corchetes, y asignamos esa lista a una variable.
2. La lista es una instancia de la interfaz **java.util.List** de Java.
3. El tamaño de la lista se puede consultar con el método **size()** y muestra que la lista contiene 3 elementos.

En el ejemplo anterior, se utilizó una lista homogénea, pero también se pueden crear listas que contienen valores de tipos heterogéneos:

```
def heterogeneous = [1, "a", true]
```

Se mencionó que, de manera predeterminada, los literales de lista son en realidad instancias de **java.util.ArrayList**, pero es posible usar un tipo de respaldo diferente para nuestras listas, gracias a la utilización de la **coerción** de tipo con el operador **as**, o con la **declaración de tipo** explícita para sus variables :

```
def arrayList = [1, 2, 3]
assert arrayList instanceof java.util.ArrayList

def linkedList = [2, 3, 4] as LinkedList            //(1)
assert linkedList instanceof java.util.LinkedList

LinkedList otherLinked = [3, 4, 5]                  //(2)
assert otherLinked instanceof java.util.LinkedList
```

1. Se define un número de lista delimitado por comas y rodeado por corchetes, y se asigna esa lista a una variable.
2. Se puede decir que la variable que contiene el literal de la lista es de tipo **java.util.LinkedList**.

   Como las listas pueden ser heterogéneas por naturaleza, las listas también pueden contener otras listas para crear listas multidimensionales:

```
def multi = [[0, 1], [2, 3]]
assert multi[1][0] == 2
```

- **Arreglos:** Groovy reutiliza la notación de lista para matrices, pero para crear matrices de tales literales, se debe definir explícitamente el tipo de matriz mediante **coerción** o **declaración de tipo**.

```
String[] arrStr = ['Ananas', 'Banana', 'Kiwi']          //(1)

assert arrStr instanceof String[]                       //(2)
assert !(arrStr instanceof List)

def numArr = [1, 2, 3] as int[]                         //(3)

assert numArr instanceof int[]                          //(4)
assert numArr.size() == 3
```

1. Se define una matriz de cadenas usando declaración de tipo de variable explícita.
2. Se asegura de que se crea una serie de cadenas de caracteres.
3. Se crea una matriz de entradas con el operador **as**.
4. Se asegura de que se crea una matriz de enteros primitivos.

También se puede crear matrices multidimensionales:

```
def matrix3 = new Integer[3][3]
assert matrix3.size() == 3

Integer[][] matrix2
matrix2 = [[1, 2], [3, 4]]
assert matrix2 instanceof Integer[][]
```

1. Se define los límites de la nueva matriz.
2. O se declara una matriz sin especificar sus límites.

- **Maps:** Llamados diccionarios o matrices asociativas en otros lenguajes de programación, Groovy presenta mapas. Los mapas asocian claves a valores, separando claves y valores con dos puntos, y cada **clave / valor** se empareja con comas, y las claves y valores enteros rodeados de corchetes.

```
def colors = [red: '#FF0000', green: '#00FF00', blue: '#0000FF']

assert colors['red'] == '#FF0000'
assert colors.green  == '#00FF00'

colors['pink'] = '#FF00FF'
colors.yellow  = '#FFFF00'

assert colors.pink == '#FF00FF'
assert colors['yellow'] == '#FFFF00'
```

```
assert colors instanceof java.util.LinkedHashMap
```

1. Se define un mapa de nombres de colores de tipo String, asociados con sus colores html con codificación hexadecimal
2. Se usa la notación del subíndice para verificar el contenido asociado con la tecla roja.
3. También se puede usar la notación de propiedad para afirmar la representación hexadecimal del color verde.
4. Del mismo modo, se puede usar la notación de subíndice para agregar un nuevo par clave / valor.
5. O la notación de propiedad, para agregar el color amarillo.

Nota: Cuando se usa nombres para las claves, realmente se definen claves de tipo String en el mapa. Groovy crea mapas que en realidad son instancias de **java.util.LinkedHashMap**.

- **Sets:** Un Set en groovy es una colección no ordenada, sin duplicados, de elementos, como en Java. Implementa la interfaz **java.util.Set**. Por defecto, un set en groovy es un **java.util.HashSet**. Si se quiere especificar otro tipo basta con instanciarlo de la siguiente forma: **def miSet = new TreeSet()**

```
def emptySet = [] as Set
println emptySet.class.name // java.util.HashSet
println emptySet.size() // 0

def set = ["Lia","Lia"] as Set
println set  // {"Lia"} No hay duplicados
```

Como en los anteriores tipos compuestos, también es posible cambiar el tipo del mismo.

```
def list = set as List
println list.class.name // java.util.ArrayList
println set.asList().class.name // java.util.ArrayList
println set.toList().class.name // java.util.ArrayList
```

Casting

Los tipos de casting que Groovy puede hacer de un tipo a otro podrán ser visualizados en la sección de **Diferencias con respecto a Java** en la subsección **Conversiones**.

```
// Un ejemplo de casting
def number = (Integer)'1'
def number = '1'.toInteger()
def number = '1' as Integer
```

Subrutinas

Al ser Groovy un lenguaje orientado a Objetos, todas las subrutinas que se manejen en sus scripts son consideradas como Métodos, ya que al ser definida estará ligada a una clase.

Un Método en Groovy se define con un tipo de retorno o con la palabra clave **def**. Los métodos pueden recibir cualquier cantidad de argumentos. No es necesario que los tipos estén explícitamente definidos al definir los argumentos. Se pueden agregar modificadores como **public**, **private** y **protected**. De forma predeterminada, si no se proporciona un modificador de visibilidad, *el método sera público*. Ademas los métodos también pueden devolver valores al programa que realiza la llamada, mediante la palabra clave **return**.

```
def nombreMetodo(parametro1, parametro2, parametro3) {
  // Codigo del metodo
}
```

También hay una disposición en Groovy para especificar valores predeterminados para los parámetros dentro de los métodos. Si no se pasan valores al método para los parámetros, se utilizan los predeterminados. Si se utilizan tanto los parámetros predeterminados como los no predeterminados, se debe tener en cuenta que los parámetros predeterminados se deben definir al final de la lista de parámetros.

```
def algunMetodo(parametro1, parametro2 = 0, parametro3 = 0) {
  // Codigo del metodo
}
```

Pasaje de Parámetros

Groovy, al igual que su lenguaje predecesor Java, posee un Pasaje de Parámetros peculiar. Se tiende a pensar que Groovy trata las primitivas y los objetos de forma diferente al pasarlas como parámetros, de forma que las primitivas se pasan por valor y los objeto por referencia. Pero esto es una idea, en realidad el Pasaje de Parámetros de Groovy es completamente **Pasaje por Valor**.

Esto significa, que Groovy al pasar un objeto como parámetro no pasa precisamente la referencia al mismo sino pasa una copia de la referencia, es decir, el valor de la referencia. Así que si se hacen cambios sobre el objeto usando la referencia pasada el objeto fuera del método recibirá dichos cambios pero si se intenta modificar la referencia como tal, esa modificación no se veran reflejada fuera del método.

```
def wat(a) {
   a[0] = 42
}

def b = new int[5]
wat(b)

assert b[0] == 42
```

En el ejemplo anterior se puede apreciar como al hacer un cambio a un objeto dentro del método usando la referencia pasada por parámetro, dicho cambio surte efecto al salir del metodo pero si vemos el siguente caso:

```
def wat(a) {
   a = new int[10]
}

def b = new int[5]
wat(b)

assert b.length == 5
```

En este caso, se puede notar como los cambios al objeto no se reflejaron al salir del método, ya que se redefinió la referencia pasada como parámetro y la misma es solo una copia de la referencia al objeto. Con esto se puede apreciar que incluso para objetos, el Pasaje de Parámetros de Groovy es por valor.

Métodos Genéricos

Los Métodos Genéricos permiten que los tipos de datos sean parámetros al definir un método, permitiendo reutilizar el mismo código con diferentes entradas. Uno de los

usos mas comunes para los Métodos Genéricos son en el uso de Colecciones, así como las Listas, los Conjuntos, los Arreglos, entre otros.

Además en Groovy, también sus clases pueden ser generalizadas haciéndolas mas flexibles en aceptar cualquier tipo y trabajar correctamente con estos.

```
def list = new ArrayList<String>()
```

Más detalles de implementación

POO (POGO's)

POGO (Plain Old Groovy Objects) tiene un significado equivalente a los POJO (Plain Old Java Object) de Java, es decir; son clases que no heredan ni implementan ninguna clase/interface de un API en concreto y externo a una aplicación, y que por tanto pueden ser reusados una y otra vez entre aplicaciones. A efectos prácticos, un POGO es una clase normal y corriente:

```
class Libro {
    String titulo
    String autor
    int numPaginas
}
```

La clase del código anterior define un libro con tres atributos que pueden ser leídos/escritos directamente, ya que Groovy generará los métodos getter/setter en tiempo de compilación. Es más, a pesar de que dichos métodos estarán disponibles cuando se compile y llame a la clase, se puede llamar a los atributos directamente por su nombre:

```
def libro = new Libro()
libro.titulo = "Effective Java 2nd Edition"
libro.autor = "Joshua Bloch"
libro.numPaginas = 346
```

- **Getters y Setters:** En ciertas ocasiones, sin embargo, se debe controlar como se almacenan y leen los valores de los atributos. Nada tan sencillo como sobreescribir el getter/setter correspondiente:

  ```
  class Libro {
      String titulo
      String autor
      int numPaginas
  ```

```
    String getTitulo() {
        return titulo.toUpperCase()
    }
}

def libro = new Libro()
libro.titulo = "Introduccion a Groovy"
libro.autor = "David Marco"
libro.numPaginas = 0

println libro.titulo
```

- **Constructores:** Groovy ofrece una sintaxis especial para crear con POGO en una única linea:

```
def libro = new Libro(titulo:'>Effective Java 2nd Edition', autor:'
  Joshua Bloch', numPaginas:346)
```

Otra forma de colocar colocar constructores es de la siguiente forma:

```
def mapa = [titulo:'Effective Java 2nd Edition', autor:'Joshua Bloch
  ', numPaginas:346]
def libroGrails = new Libro(mapa)

println mapa.titulo
```

- **Niveles de acceso:** En Groovy se aprecia, en cierto modo, el concepto de **Convención sobre Configuración**; esto es, que la sintaxis más simple refleja el comportamiento mas común de un componente. Este método se aplica de la siguiente manera a los niveles de acceso por defecto en POGO's:

  - Todas las clases son públicas por defecto.
  - Todos los métodos son públicos por defecto.
  - Todos los atributos son privados por defecto.

```
class Libro {
   String titulo
   String autor
   private int numPaginas
}
```

En el ejemplo anterior, se ha definido explícitamente el atributo **numPaginas** como privado. Esto indica al compilador que no debe generar métodos getter/-setter para este atributo, de manera que el atributo no será visible de ninguna manera para un cliente Java. Pero esto solo sucede en Java, en Groovy se puede seguir accediendo al atributo, para leerlo y escribirlo. Esto es debido a que Groovy tiene ciertos privilegios al usar otro código Groovy, se puede llamar a esto **Despotismo Groovy**. Incluso aunque se proporcione métodos setter/getter privados para ese atributo, Groovy podrá seguir accediendo a el.

Un privilegio extra de Groovy es que puede leer y escribir un atributo directamente, sin pasar por el método getter/setter correspondiente (sea este generado implícitamente por el compilador o explícitamente por nosotros):

```
println libro.@titulo
```

En la sentencia anterior se ha utilizado el caracter **@** para indicar que se debe acceder a la variable titulo directamente.

Trait

Los traits son una construcción estructural del lenguaje que permite:

- Composición de comportamientos.
- Implementación en tiempo de ejecución de las interfaces.
- Comportamiento predominante.
- Compatibilidad con la comprobación / compilación de tipos estáticos.

Se pueden ver como interfaces que llevan implementaciones y estado predeterminados. Un trait se define usando la palabra clave de **trait**:

```
trait FlyingAbility {                    //1
      String fly() { "I'm flying!" }     //2
}
```

1. Declaración de un trait.
2. Declaración de un método dentro de un trait.

Luego se puede usar como una interfaz normal usando la palabra clave **implements**:

```
class Bird implements FlyingAbility {}
def b = new Bird()
assert b.fly() == "I'm flying!"
```

1. Agrega el atributo **FlyingAbility** a las capacidades de la clase **Bird**.
2. Instancia un nuevo **Bird**.
3. La clase **Bird** obtiene automáticamente el comportamiento del rasgo de **FlyingAbility**

Closures

Siendo una de las características mas potentes del Groovy, los Closures son bloques de código anónimos por naturaleza (fuera de cualquier clase) que puede ser definido y usado en puntos distintos de un programa. Los mismos pueden ser llamados como una **Llamada Normal de Método** o usando **call()**. Ya que los Closures pueden ser asignados a variables, pasados como parámetros y pueden ser retornados como valores de un método, se consideran **Valores de Primera Clase**.

```
def saludar = { println 'Hola Mundo de los Closures!' }

saludar()
saludar.call()
```

Además, a los Closures también pueden obtener parámetros, ya sean explícitos o implícitos, otorgándoles una mayor utilidad. Para el uso de parámetro implícito debe usarse la palabra clave **it** para referirse a dicho parámetro dentro del Closure.

```
def saludar2 = {                                    // Parametro explicito
    println "Hola ${it}!"
}

def saludar3 = { nombre, apellido ->                // Parametro implicito
    println "Hola ${nombre} ${apellido}!"
}
```

Currying

Groovy ademas, nos permite pre-cargar valores en los parametros de un Closure, esto se conoce como Currying. Pero no se corresponde con el concepto real de Currying dentro de la Programación Funcional debido a las diferentes reglas de alcance que Groovy

aplica en los Closures.

El Currying en Groovy se divide en tres tipos:

- **Left Currying:** es el hecho de establecer el parámetro más a la izquierda de un Closure (**curry()**).

```
def nCopies = { int n, String str -> str*n }

def twice = nCopies.curry(2)                    // Left Currying

assert twice('bla') == 'blabla'
assert twice('bla') == nCopies(2, 'bla')
```

- **Right Currying:** Similar al Left Currying, establece el parámetro más a la derecha de un Closure (**rcurry()**).

```
def nCopies = { int n, String str -> str*n }

def blah = nCopies.rcurry('bla')          // Right Currying

assert blah(2) == 'blabla'
assert blah(2) == nCopies(2, 'bla')
```

- **Index Currying:** En caso de que un Closure acepte más de 2 parámetros, establece cualquiera de los parámetros indicando su posición (**ncurry()**).

```
def volume = { double l, double w, double h -> l*w*h }

def fixedWidthVolume = volume.ncurry(1, 2.0)      // Index Currying

assert volume(3.0, 2.0, 4.0) == fixedWidthVolume(3.0, 4.0)

def fixedWidthAndHeight = volume.ncurry(1, 2.0, 4.0)    // Multiple
  Index Currying

assert volume(3.0, 2.0, 4.0) == fixedWidthAndHeight(3.0)
```

Estático vs Dinámico

Groovy proporciona verificación de tipos opcional gracias a la anotación **@TypeChecked**. El comprobador de tipos se ejecuta en tiempo de compilación y realiza un

análisis estático de código dinámico. El programa se comportará exactamente igual si la verificación de tipo ha sido habilitada o no. Esto significa que la anotación **@TypeChecked** es neutral con respecto a la semántica de un programa. Aunque puede ser necesario agregar información de tipo en las fuentes para que el programa se considere seguro, al final, la semántica del programa es la misma.

Si bien esto puede sonar bien, en realidad hay un problema con esto: la comprobación de tipo de código dinámico, realizada en tiempo de compilación, es por definición solo correcta si no se produce un comportamiento específico de tiempo de ejecución. Por ejemplo, el siguiente programa pasa la comprobación de tipos:

```
class Computer {
    int compute(String str) {
        str.length()
    }
    String compute(int x) {
        String.valueOf(x)
    }
}

@groovy.transform.TypeChecked
void test() {
    def computer = new Computer()
    computer.with {
        assert compute(compute('foobar')) =='6'
    }
}
```

Hay dos métodos de cálculo. Uno acepta un **String** y devuelve una **int**, la otra acepta una **int** y devuelve un **String**. Si compila esto, se considera seguro de tipo: la llamada de cálculo interno **('foobar')** devolverá un **int**, y el cálculo de llamada en este **int** a su vez devolverá un **String**.

Ahora, antes de llamar a *prueba()*, se considera agregar la siguiente línea:

```
Computer.metaClass.compute = {String str -> new Date ()}
```

Al utilizar la Metaprogramación en tiempo de ejecución, en realidad se está modificando el comportamiento del método de cómputo **(String)**, de modo que en lugar de devolver la longitud del argumento proporcionado, se devolverá un **Date**. Si ejecuta el programa, fallará en el tiempo de ejecución. Como ésta línea se puede agregar desde cualquier lugar, en cualquier hilo, no hay absolutamente ninguna manera para que el verificador de tipos se asegure estáticamente de que eso no ocurra. Este es solo un ejemplo, pero esto ilustra el concepto de que hacer un análisis estático de un programa

dinámico es intrínsecamente incorrecto.

El lenguaje Groovy proporciona una anotación alternativa a **@TypeChecked** que en realidad asegurará que los métodos que se deducen como llamados se invocarán eficazmente en el tiempo de ejecución. Esta anotación convierte el compilador de Groovy en un compilador estático, donde todas las llamadas de método se resuelven en tiempo de compilación y el bytecode generado se asegura de que esto suceda: la anotación es **@groovy.transform.CompileStatic**.

La anotación **@CompileStatic** se puede agregar en cualquier lugar donde se pueda usar la anotación **@TypeChecked**, es decir; en una clase o un método. No es necesario agregar **@TypeChecked** y **@CompileStatic**, ya que **@CompileStatic** realiza todo lo que **@TypeChecked** hace, pero además activa la compilación estática.

Con el ejemplo anterior que falló, se reemplaza la anotación **@TypeChecked** con **@CompileStatic**:

```
class Computer {
    int compute(String str) {
        str.length()
    }
    String compute(int x) {
        String.valueOf(x)
    }
}

@groovy.transform.CompileStatic
void test() {
    def computer = new Computer()
    computer.with {
        assert compute(compute('foobar')) == '6'
    }
}
Computer.metaClass.compute = { String str -> new Date() }
test()
```

Esta es la única diferencia. Si se ejecuta este programa, esta vez, no hay error de tiempo de ejecución. El método de prueba se volvió inmune a los **monkey patching**, porque los métodos de cálculo que se llaman en su cuerpo están vinculados en tiempo de compilación, por lo que incluso si la metaclase de **Computer** cambia, el programa aún se comporta como espera el verificador de tipos.

Hay varios beneficios de usar **@CompileStatic** en un código:

- Tipo de seguridad.

- Inmunidad al Monkey Patching.
- Mejoras de rendimiento.

Autoboxing

En Groovy, todo es un objeto. Incluso los tipos primitivos son tratados como objetos. Ejemplo:

```
println 2.floatValue()
```

El código anterior llama al método **floatValue()**, este método forma parte de la clase **Integer**. Esto sucede debido a que Groovy convierte (cuando es necesario) cualquier variable de tipo primitivo en su correspondiente clase **wrapper**.

Prácticas opcionales

- **Punto y coma:** Como se ha visto en varios ejemplos anteriores, el carácter de punto y coma **(;)** al final de cada línea es opcional. Sólo debe ser usado al escribir varias sentencias en una única línea:

```
println "hola"; println "Mundo"
```

- **Paréntesis:** El carácter de paréntesis para llamar a un método con argumentos también es opcional.

```
void metodo(String s1, String s2) {
    println s1 + s2
}
metodo "uno", "dos"
```

Los paréntesis si son obligatorios cuando se llama a un método sin argumentos, ya que el compilador no podría saber si es un método o una variable a lo que se está refiriendo.

- **return:** La sentencia **return** al final de un método también es opcional en Groovy. Si se omite, el resultado devuelto por su última línea será implícitamente su valor de retorno:

```
void cuadrado(int numero) {
    println "devolviendo cuadrado de " + numero
    numero*numero
```

```
}
```

El método anterior, cuando es ejecutado, escribe un mensaje por la salida estándar y devuelve el cuadrado del número pasado como argumento. La sentencia **return** solo es necesario cuando, por ejemplo, se necesita devolver un valor dentro de un **bucle** o un bloque **switch**. En caso de que la última sentencia de un método o script no devuelva ningún valor (por ejemplo una sentencia **println** o la llamada a un método **void**), se devolverá **null**.

- **Declaración de variables:** En Groovy no se tiene que declarar el tipo de una variable cuando se le asigna un valor. Esto es debido a la naturaleza dinámica de Groovy.

```
x = "Esto es un String"
println x.class
x = 12
println x.class
```

La variable x, que no ha sido declarada previamente en el código, es inicializada con un String primero y con un entero después. Las sentencias **println** demuestran esto mostrando por pantalla el tipo actual de la variable después de cada asignación. Esta naturaleza dinámica es conocida como **duck typing**. En este, si una variable contiene un String y se comporta como tal, entonces es un String (aunque no se haya definido su tipo). Formalmente, esta característica se denomina **inferencia de tipos**.

El código anterior, que funciona bien en scripts, no puede ser usado dentro de una clase: o se declara el tipo explícitamente (String o int) o se declara la variable con **def**:

```
class MiClase {
    def x = "Esto es un String"

    void metodo() {
        println x.class
        x = 12
        println x.class
    }
}

new MiClase().metodo()
```

Uso y Limitaciones de Groovy

Usos

1. Uno de los principales usos de Groovy es debido a su expresividad y sintaxis parecida a la de lenguajes dinámicos como Ruby o Python. Groovy fue pensado de forma que el código fuese visualmente más amigable al desarrollador, pero sin descuidar el potencial y las herramientas de Java.

2. Aprender Groovy requiere una curva de aprendizaje mucho menor que cambiarse a un nuevo lenguaje como Python o Ruby (si se es programador Java), y esto es muy común verlo debido a que Java es el lenguaje más demandado de la actualidad.

3. Groovy es utilizado como un lenguaje de **scripting** para la plataforma Java, por lo tanto tendrá todos los beneficios de un lenguaje de tipo scripting orientado a Java, sin embargo, a partir de la versión 2, los scripts de Groovy se pueden compilar estáticamente proporcionando un rendimiento de código casi Java.

4. Es imposible que al hablar de Groovy, no se le relacione con Java, por lo tanto, Groovy también es muy utilizado, debido a que provee métodos adicionales sobre Java que hacen que el código sea más legible y compacto, como soporte nativo de expresiones regulares, XML, JSON, trabajo simplificado con archivos, etc.

5. El 99 % del código de Java funcionará en Groovy.

6. Groovy es genial para las secuencias de **comandos del sistema**, del mismo modo que lo son Ruby y Python. No todo debe escribirse dentro de una clase para que pueda escribir secuencias de comandos en poco tiempo.

7. Las pruebas unitarias son mucho más fáciles de escribir dado que el código es mucho más expresivo, y por lo tanto, hay menos oportunidades de errores, especialmente gracias a los DSL.

8. Groovy se puede usar para casi todo, como desarrollar aplicaciones de escritorio, escribir software de sistema integrado, desarrollar aplicaciones web usando frameworks como **Grails**, aplicaciones móviles, hacking, etc.

9. Uno de los grandes usos de Groovy es por su conocido framework de desarrollo, **Grails**. Este framework es muy parecido a **Hibernate / Spring** pero mejor, es decir; se reutilizaron tecnologías probadas como Spring e Hibernate y se creó una convención dinámica sobre configuración capa de abstracción encima de ellos para optimizar los procesos. También, Hiberante/Spring se encuentran debajo de Grails dentro de la su estructura lo que quiere decir que se aceptan todos los módulos de Spring, y es interesante ya que se sabe que Spring puede integrarse a casi todas las bases de datos de manera óptima.

Limitaciones

1. Las llamadas al código que no es de JVM pasan por JNI, que es lento y aveces con un gran costo.
2. Falta de liberación determinística de recursos.
3. Técnicamente hablando, es más lento que Java.
4. En general, posee las limitaciones de un lenguaje interpretado común.

Comparación con Java

Importaciones

Todos estos paquetes y clases se importan de manera predeterminada, es decir; no es necesario escribir en código **import** y el nombre del paquete para usarlos.

```
java.io.*
java.lang.*
java.math.BigDecimal
java.math.BigInteger
java.net.*
java.util.*
groovy.lang.*
groovy.util.*
```

Múltiples métodos

En Groovy, los métodos que se invocarán se eligen en tiempo de ejecución. En Java, esto es lo opuesto: los métodos se eligen en tiempo de compilación, según los tipos declarados.

```
int metodo(String arg){
   return 1;
}

int metodo(Object arg){
   return 2;
}

Object o = "Objeto";
int resultado = metodo(o);
```

En Java se tendrá:

```
assertEquals(2, resultado);
```

Mientras que en Groovy se tendra:

```
assertEquals(1, resultado);
```

Esto se debe a que Java utilizará el tipo de información estática, que es que o se declara como un Objeto, mientras que Groovy lo seleccionará en tiempo de ejecución, cuando realmente se llama al método. Como se llama con un String, se llama a la versión String.

Inicializando matrices

En Groovy, el **{ ... }** bloque está reservado para cierres. Eso significa que no se puede crear literales de una matriz con la misma sintaxis de Java, ahora la forma de **inicializar** matrices es de la siguiente forma:

```
int[] arreglo = [1,2,3]
```

Funciones Lambdas

Java 8 admite lambdas y referencias de métodos:

```
Runnable run = () -> System.out.println("Run");
list.forEach(System.out::println);
```

Java 8 lambdas pueden considerarse más o menos como clases internas anónimas. Groovy no es compatible con esa sintaxis, pero posee los cierres en su lugar:

```
Runnable run = { println 'run' }
list.each { println it } // or list.each(this.&println)
```

Primitivas y Wrappers

Debido a que Groovy usa **Object** para todo, empaqueta automáticamente referencias a **primitivos**. Debido a esto, no sigue el comportamiento de **ensanchamiento** de

Java teniendo prioridad sobre el **boxing**.

Un ejemplo a continuación:

```
int i
m(i)

void m(long l){
    println "in m(long)"        //1.
}

void m(Integer i){
    println "in m(Integer)"     //2.
}
```

1. Es el método que Java llamaría, ya que la ampliación tiene prioridad sobre el **desempaquetado**.
2. Es el método que realmente llama Groovy, ya que todas las referencias primitivas usan su clase contenedora.

Comportamiento del ==

En Java == significa igualdad de tipos primitivos o identidad para objetos. En Groovy el == se traduce de la siguiente manera: se supone a y b dos variables cualquieras, entonces **a.compareTo(b) == 0**, retornará True o False, en este caso, ambas variables deben ser comparables, **a.equals(b)** retornará True o False si son o no iguales. Para verificar la identidad, está la palabra reservada **is**. Por ejemplo: **a.is(b)**

Palabras reservadas

Hay algunas palabras reservadas más en Groovy que en Java.

```
as
def
in
trait
```

Conversiones

Se visualiza como maneja Java y Groovy las conversiones.

De \ A	boolean	byte	short	char	int	long	float	double
boolean	-	N	N	N	N	N	N	N
byte	N	-	Y	C	Y	Y	Y	Y
short	N	C	-	C	Y	Y	Y	Y
char	N	C	C	-	Y	Y	Y	Y
int	N	C	C	C	-	Y	T	Y
long	N	C	C	C	C	-	T	T
float	N	C	C	C	C	C	-	Y
double	N	C	C	C	C	C	C	-

Tabla 1: Conversiones en Java

'Y' indica una conversión que Java puede hacer, **'C'** indica una conversión que Java puede hacer siempre y cuando haya un casting explicito, **'T'** indica una conversión que Java puede hacer, pero los datos se truncan, **'N'** indica una conversión que Java no puede hacer.

De \ A	boolean	byte	short	char	int	long	float	double
boolean	-	N	N	N	N	N	N	N
Boolean	B	N	N	N	N	N	N	N
byte	T	-	Y	Y	Y	Y	Y	Y
Byte	T	B	Y	Y	Y	Y	Y	Y
short	T	D	-	Y	Y	Y	Y	Y
Short	T	D	B	Y	Y	Y	Y	Y
char	T	Y	Y	-	Y	Y	Y	Y
Character	T	D	D	D	D	D	D	D
int	T	D	D	Y	-	Y	Y	Y
Integer	T	D	D	Y	B	Y	Y	Y
long	T	D	D	Y	D	-	T	T
Long	T	D	D	Y	D	B	T	T
BigInteger	T	D	D	D	D	D	D	D
float	T	D	D	T	D	D	-	Y
Float	T	D	D	T	D	D	B	Y
double	T	D	D	T	D	D	D	-
Double	T	D	D	T	D	D	D	B
BigDecimal	T	D	D	D	D	D	T	T

Tabla 2: Conversiones en Groovy

De \ A	Boolean	Byte	Short	Character	Integer	Long	BigInteger	Float	Double	BigDecimal
boolean	B	N	N	N	N	N	N	N	N	N
Boolean	-	N	N	N	N		N	N	N	N
byte	T	B	Y	D	Y	Y	Y	Y	Y	Y
Byte	T	-	Y	D	Y	Y	Y	Y	Y	Y
short	T	D	B	D	Y	Y	Y	Y	Y	Y
Short	T	T	-	D	Y	Y	Y	Y	Y	Y
char	T	D	D	D	D	D	D	D	D	D
Character	T	D	D	-	D	D	D	D	D	D
int	T	D	D	D	B	Y	Y	Y	Y	Y
Integer	T	D	D	D	-	Y	Y	Y	Y	Y
long	T	D	D	D	D	B	Y	T	T	Y
Long	T	D	T	D	T	-	Y	T	T	Y
BigInteger	T	D	D	D	D	D	-	D	D	T
float	T	D	D	D	D	D	D	B	Y	Y
Float	T	T	T	D	T	T	D	-	Y	Y
double	T	D	D	D	D	D	D	D	B	Y
Double	T	T	T	D	T	T	D	T	-	Y
BigDecimal	T	D	D	D	D	D	D	D	D	-

Tabla 3: Más conversiones en Groovy

'Y' indica una conversión que Groovy puede hacer, **'D'** indica una conversión que Groovy puede hacer cuando se compila dinámicamente o si se especifica un casting explícito, **'T'** indica una conversión que Groovy puede hacer pero los datos se truncan, **'B'** indica una operación de boxing/unboxing, **'N'** indica una conversión que Groovy no puede realizar.

Bloques ARM

El bloque ARM (Automatic Resource Management) de Java 7 no es compatible con Groovy. En cambio, Groovy proporciona varios métodos que dependen de cierres, que tienen el mismo efecto y son más compactos.

Se visualiza el siguiente ejemplo:

```
Path file = Paths.get("/path/to/file");
Charset charset = Charset.forName("UTF-8");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

Así puede escribirse en Groovy:

```
new File('/path/to/file').eachLine('UTF-8') {
  println it
}
```

GString

Groovy permite crear instancias de objetos de **java.lang.String**, así como de **GStrings (groovy.lang.GString)** que también se denominan **Strings interpolados** en otros lenguajes de programación.

Como los literales de cadena doble **("... ")** se interpretan como valores de GString (si existe interpolación en el literal), Groovy puede fallar con un error de compilación o producir un código sutilmente diferente si una clase literal de String que contiene un carácter de dólar se compila con el compilador de Groovy y Java.

Por lo general, Groovy realizará un casting automático entre GString y String si una API declara el tipo de un parámetro, se debe tener cuidado con las API de Java que aceptan un parámetro **Object** que luego verifique el tipo real.

Visibilidad del alcance de los paquetes

En Groovy, omitir un modificador en un campo no da como resultado un campo **package-private** como en Java:

```
class Person {
   String name
}
```

En cambio, si es usado para crear una propiedad, es decir; un campo privado, con su **getter** y **setter** asociado, entonces es posible crear un campo package-private al anotarlo con **@PackageScope**:

```
class Person {
   @PackageScope String name
}
```

Proyectos desarrollados con Groovy

Grails

Grails (anteriormente conocido como "Groovy on Grails") es un framework de programación basado en Groovy e inspirado en Ruby on Rails . Posee dos características sobresalientes para los desarrolladores: la **codificación por convención** y el **DRY** (Don't Repeat Yourself). Grails corre en JDK, por lo tanto, también se tiene acceso al ecosistema completo de Java. Grails fue construido sobre los conocimientos y herramientas de Spring MVC, Hibernate y Site mesh.

Grails provee de muchas técnicas y herramientas, se hará mención de alguna de ellas:

- **Scaffolding:** plantillas que permiten de manera dinámica o estática generar pantallas que administren los objetos de dominio de las aplicaciones .
- **Internacionalización:** En este aspecto Grails usa el soporte de **java.util.ResourceBundle** y **java.util.Locale**, lo cual es una tónica habitual de todos los frameworks MVC del mercado.

- **Capa de servicio:** La capa de servicio en Grails se basa en el mecanismo de **autowiring** de Spring, con lo que para inyectar un servicio dentro de un controlador, éste se realiza mediante técnicas de **autowiring by name**, sin necesidad por tanto de preocuparse de establecer las dependencias explícitamente.
- **Configuración basada en entornos:** Grails presenta una solución muy elegante para disponer de distintos entornos y no tener que realizar labores de parametrización intrusivas dentro de la aplicación.
- Pruebas unitarias y de integración: Gracias a Groovy es fácil la realización de pruebas unitarias gracias a la librería de Spock.
- Entre otras más.

Consideramos que el uso de Groovy en este framework de desarrollo se debe principalmente a la idea de diseñar un framework que adoptara todas las características y librerías de Java, pero con la intención de hacer la programación más digerible, esto debido a que la sintaxis de Groovy tiende ser más agradable a la vista, además de que se requiere de un aprendizaje corto (si ya eres un programador Java). Otro uso que tiene Groovy es la realización de proyectos y su testeo a través de pruebas unitarias utilizando Spock (más adelante se hablará sobre esta herramienta).

Gradle

Gradle es una herramienta de compilación con un enfoque en la automatización de compilación y soporte para el desarrollo de múltiples lenguajes. Si se está creando, probando, publicando e implementando software en cualquier plataforma, Gradle ofrecerá un modelo flexible que puede admitir todo el ciclo de vida de desarrollo, desde la compilación y el código de empaquetado hasta la publicación de sitios web. Gradle ha sido diseñado para admitir la automatización de compilación en múltiples lenguajes y plataformas, incluyendo Java, Scala, Android, C / C ++ y Groovy, y está estrechamente integrado con herramientas de desarrollo y servidores de integración continua, incluidos Eclipse, IntelliJ y Jenkins.

Gradle proporciona un lenguaje específico de dominio, o DSL, para describir construcciones. Este lenguaje de compilación está basado en Groovy, con algunas adiciones para que sea más fácil describir una compilación.

Groovy a nuestro punto de vista es importante en Gradle, debido a que las ventajas de un DSL interno sobre XML son muchas cuando se trata de scripts de compilación. Aunque Gradle es una herramienta de creación general en su núcleo, éste está basado principalmente en proyectos Java. Creemos que aunque Java es el lenguaje de compilación del momento, este posee algunas limitaciones que lenguajes expresivo, agradable y poderoso como Groovy, Python o Ruby podrían lograr evitar. Se decide por Groovy por

la razón más obvia: la curva de aprendizaje, como se mencionó los proyectos Java son la base de Gradle por lo tanto usar Groovy supone muchas ventajas a los desarrolladores.

Spock

Spock es un marco de pruebas unitarias que utiliza, en gran medida, la sintaxis de Groovy, haciendo que sus pruebas sean comprensibles y fáciles de entender. Aunque es una tecnología Groovy, puede usarse para probar clases de Java también. Gracias a JUnit, Spock es compatible con la mayoría de los IDE, herramientas de compilación y servidores de integración continua. Spock está inspirado en JUnit, jMock, RSpec, Groovy, Scala, Vulcans, entre otros.

Spock nació con la idea de hacer las pruebas típicas de TDD de JUnit más expresivas, eficientes, y desarrolladas en menor tiempo. Opinamos que Gracias a la sintaxis de Groovy se puede mejorar la claridad de pruebas unitarias mediante el uso de sus cierres, uso de mapas; también beneficia como lenguaje propio, debido a su interoperabilidad con Java, dinamismo y en como aborda la metaprogramación. Con Groovy es fácil realizar pruebas basadas en datos, la técnica mocking y aserciones. Gracias a que Groovy es un lenguaje de tipado dinámico permite que la realización de pruebas unitarias sean más cortas y menos propensa a errores.

Asgard

Asgard es una interfaz web para implementaciones de aplicaciones y administración de la nube en Amazon Web Services (AWS). Asgard era el hogar del dios nórdico de los truenos y los relámpagos, y es donde controlas las cosas en la nube.

Asgard maneja un entorno dinámico a gran escala. Proporciona una vista única y clara de la nube de Netflix, permite reversiones y proporciona seguridad. También hay muchas otras características: soporte para rolling pushes, aplicación de convenciones, integración de Jenkins CI, monitoreo de estado, seguimiento de métricas, etc.

Asgard fue desarrollado con Groovy y con su herramienta predilecto de desarrollo de aplicaciones webs, Grails. La primera versión de Asgard se inició en Groovy y Grails porque Netflix era principalmente una tienda de Java y como ya hemos dejado en claro en este informe, Groovy es como Java, pero más poderoso. Groovy a nuestro punto de vista, provee más dinamismo y transparencia a los desarolladores. Groovy se beneficia de expertos internos en el ajuste JVM y un amplio ecosistema de librerías de Java, reduce o elimina la monotonía de los archivos de configuración XML, ganancias de productividad para escribir código, se mejoran las pruebas y la implementación. Además la característica minimalista de Groovy y su curva de aprendizaje extremadamente baja

(cuando nos referimos a desarrolladores Java) son cualidades que se buscan hoy en día. Es menester compartir que compañías como IBM, eBay y TRUSTe hacer uso de algunas de sus capacidades.

Proyecto: 8-Puzzle

Enunciado

Introducción

El objetivo de este proyecto es el de resolver el problema del 8-Puzzle con un algoritmo de búsqueda A*. El algoritmo A* es un algoritmo informado de búsqueda de caminos de costo mínimo. Para este proyecto debe implementar la versión del algoritmo A* basada en el Modelo General de Etiquetamiento.

Descripción del Problema

El problema de 8-Puzzle consiste en un tablero 3x3, el cual contiene 8 bloques cuadrados numerados (desde el 1 hasta el 8) y un cuadrado vació (en nuestra representación sera numerado con 0). Los cuadrados adyacentes al cuadrado vació pueden moverse a ese espacio, y entonces dejar el espacio que ocupaba como cuadrado vació. El objetivo es mover los cuadrados a partir de una configuración inicial del tablero, que llamamos estado inicial, hasta alcanzar una configuración especifica que llamaremos estado meta.

La formulación del 8-Puzzle contiene los siguientes elementos:

- **Estados:** Un estado que indica la posición de cada uno de los nueve cuadrados dentro del tablero.
- **Estado inicial:** Un estado con en el que se inicia el problema.
- **Estado meta:** Un estado que se quiere alcanzar a partir de un estado inicial.
- **Acciones:** Dado un estado, las acciones posibles a ejecutar son las de mover un cuadrado hacia el espacio vacío, aplicando un movimiento del cuadrado en alguna de las siguientes direcciones: izquierda, derecha, arriba y abajo.
- **Función para obtener sucesores:** A partir de un estado se generan los estados factibles o validos que resultan de las cuatros posibles acciones indicadas anteriormente.
- **Grafo de estados:** Es un grafo en el cual los vértices son estados. Existe un lado entre un estado A y un estado B, si es posible llegar desde el estado A hasta el estado B. Esto es, el estado B es un estado sucesor del estado A que se obtiene por medio de una de aplicar la función para obtener sucesores al estado A.
- **Costo del camino:** Cada movimiento o paso en el tablero tiene un costo de uno. Se puede construir un camino en el Grafo de estados que representa cada uno de

los movimientos que se efectúan desde un estado inicial hasta un estado nal. En costo de ese camino es igual al numero de pasos (o movimientos) que se dan para alcanzar el estado meta.

- **Función de prueba de meta:** Determina si un estado es el estado meta.

El **objetivo** en este proyecto es *encontrar el camino mas corto desde el estado inicial, hasta el estado meta usando el algoritmo A*.*

Detalles de Diseño y de Implementación

- **Grafo a utilizar:** El grafo a usar para resolver este problema es un *grafo implícito*. Esto es, el *Grafo de estados* se va construyendo a partir del *estado inicial*, con el cual se obtienen los estados sucesores, con la *función para obtener sucesores*, hasta que se alcanza el *estado meta*. Es decir, no se guarda el grafo completo, que es todas las posibles combinaciones del tablero, en memoria; por el contrario se van almacenando los estados que se van observando a medida que se expande un estado dado.

- **Heurística a utilizar:** La heurística a usar para la implementación del algoritmo de búsqueda A* será la *Distancia Manhattan*, la cual se basa en sumar de las distancias, verticales y horizontales, de cada uno de los bloques, a su posición en el estado meta.

- **Evitando explorar estados visitados anteriormente:** Al resolver este problema con el algoritmo A*, tenemos que es posible visitar el mismo estado o tablero varias veces. Para prevenir la exploración innecesaria de estados, cuando se considere los estados sucesores de un estado actual, se debe evitar generar un estado que sea igual que el estado antecesor, al estado actual o algún otro estado que ya sea parte del *Grafo de estados*.

- **Detección de puzzles que no se pueden resolver:** Desde cualquier tablero inicial no siempre es posible alcanzar el estado meta. El programa debe detectar si un tablero se puede resolver o no.

Entrada y Salida de Datos

- **Entrada de datos:** Debe hacer un programa llamado Puzzle8.groovy que se podrá ejecutar desde la consola con el siguiente comando:

```
$ groovy Puzzle8.groovy <instancia>
```

Se tiene que **instancia** es el nombre de un archivo con los datos del tablero, el cual es representado con tres las y tres columnas de números desde el 0 hasta el 8, separados por espacio. Si en la entrada no se indica una instancia, entonces se debe dar un mensaje de error al usuario.

- **Salida de datos:** Si el tablero no se puede resolver se debe imprimir por la salida estándar la frase **"No hay solucion."**. En caso contrario, se debe imprimir por salida estándar la secuencia de los tableros que resuelven el *puzzle* y que componen el camino de costo mínimo.

Explicación

Se consideró desarrollar este proyecto ya que al ser Groovy un Lenguaje Orientado a Objetos se otorga mayor facilidad para la creación y manipulación de Grafos, en este caso, para la creación paso a paso y manipulación del **Grafo Implícito de Estados**. Junto a esto, permite el encapsulamiento del comportamiento de dichos Estados del tablero definiendolos genéricamente como una clase. Además, Groovy posee una manera sencilla de leer archivos y manipular la información obtenida, dicha ventaja se logró explotar haciendo la lectura del archivo de entrada.

Por último, cabe destacar que este proyecto pudo haber sido escrito en cualquier lenguaje orientado a objetos pero lo que hace a Groovy una opción más llamativa es que si ya se tiene un conocimiento previo de como es programar en Java, ¿Por qué no programarlo en un lenguaje que ya sabes pero simplificando a gran manera la cantidad de código que se escriben? Y efectivamente Groovy garantiza esa facilidad de escribir un código de Java, donde un programa equivalente puede incluso duplicar la cantidad de código que se necesita.

Enlace al repositorio

En el siguiente enlace podrán acceder al repositorio donde se encuentra el código fuente para la resolución de este proyecto:

https://github.com/migcanedo/8Puzzle