Revista Iberoamericana de las Ciencias Computacionales e Informática **ISSN: 2007-9915**

Comparativa sintáctica entre los lenguajes de programación java y groovy

Syntactic comparison between programming languages java and groovy

Rosa Isela Zarco Maldonado
Universidad Autónoma del Estado de México
zamrtct@hotmail.com

Joel Ayala de la Vega
Universidad Autónoma del Estado de México
joelayala2001@yahoo.com.mx

Oziel Lugo Espinosa
Universidad Autónoma del Estado de México
ozieluz@hotmail.com

Alfonso Zarco Hidalgo
Universidad Autónoma del Estado de México
azarcox@hotmail.com

Hipólito Gómez Ayala
Universidad Autónoma del Estado de México
escribeme88@gmail.com

Resumen

Uno de los lenguajes que lleva varios años de vida y que permanece como uno de los más importantes debido a sus diversas características que permiten la creación de aplicaciones de software, es el lenguaje Java. Java es un lenguaje que permite el desarrollo para aplicaciones de dispositivos móviles, de escritorio, corporativas y de igual manera para el entorno web. Por otro lado, el área de desarrollo de lenguajes de programación se mantiene en un gran dinamismo. En el 2003 aparece el lenguaje de programación Groovy, este lenguaje conserva una sintaxis familiar a Java pero con características particulares. Tanto Groovy como Java son Lenguajes Orientados a Objetos y se ejecutan sobre una Máquina Virtual. La intención de éste escrito es realizar una comparativa sintáctica de los lenguajes Java y Groovy para observar las particularidades de cada uno, y de esta manera, facilitar a los programadores la implementación de sus proyectos

Palabras clave: Programación Orientada a Objetos, modularidad, métodos, polimorfismo, encapsulamiento, jerarquía, tipificación, concurrencia, persistencia.

Abstract

One of the programing languages that has several years of life and remains one of the most important due to its various features that enable the creation of software applications is the Java language. Java is a language that allows the development of applications for mobile, desktop, corporative and likewise for the web. Moreover, the development of programming languages field remains very strong. In 2003 the Groovy programming language appears, this language retains a familiar syntax to Java but with particular characteristics. Groovy and Java are both object-oriented languages and run on a virtual machine. The intention of this paper is to conduct a comparative syntax of Java and Groovy languages to observe the particularities of each, and thus make it easier for programmers to implement their projects

Keywords: object-oriented programing (OOP), modularity, polymorphism methods, encapsulation, hierarchy, classification, concurrency, persistence.

Fecha recepción: Abril 2015 **Fecha aceptación:** Junio 2015

Introducción

Los lenguajes de programación existen desde hace algunas décadas. Cada nuevo lenguaje proporciona nuevas herramientas para la vida diaria. Uno de los lenguajes que lleva años de vida y que ha sabido permanecer como uno de los más importantes es el lenguaje Java, Java fue desarrollado por Sun Mycrosystems en 1991, con la idea principal de crear un lenguaje para unificar equipos electrónicos de consumo doméstico. En primer momento fue nombrado Oak, pero en 1995 se le cambió el nombre a Java y a partir del año 2009 es propiedad de la corporación Oracle.

Por otro lado se tiene a Groovy, creado por James Strachan y Bob McWhirter en el 2003. Groovy está basado en los lenguajes Smaltalk, Python y Ruby, aunque también se dice que está basado en Perl. Groovy conserva una sintaxis familiar a Java, haciendo de esta manera que los programadores que trabajan Java les sea fácil familiarizarse con Groovy.

Dado que estos dos lenguajes se consideran Orientados a Objetos, para realizar la comparativa se seleccionó como base la definición de Grady Booch acerca del concepto de la Programación Orientada a Objetos:

> *La programación orientada a objetos es un modelo de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia*

Los lenguajes orientados a objetos deben cumplir cuatro elementos fundamentales de éste modelo:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía

Al decir fundamentales, quiere decir que un lenguaje que carezca de cualquiera de estos elementos no es orientado a objetos.

Hay tres elementos secundarios del modelo orientado a objetos:

- Tipos (tipificación)
- Concurrencia
- Persistencia

Por secundarios quiere decirse que cada uno de ellos es una parte útil en el lenguaje orientado a objetos, pero no es esencial (Booch, 1991).

Por lo extenso del lenguaje Java, sólo se escogieron estos puntos a comparar, por lo que a lo largo de este escrito se tratará de realizar una comparativa de estos lenguajes bajo el esquema antes indicado, incluyendo, por supuesto, el análisis de los operadores de control y uso de la máquina virtual

COMPARATIVA

A. MODULARIDAD

Una clase viene a representar la definición de un módulo de programa, y a su vez define métodos y atributos comunes.

Un objeto es una instancia de la clase, ésta instancia contiene atributos y métodos con valores específicos de sus datos. (Rodríguez Echeverría & Prieto Ramos, 2004).

Un método se escribe dentro de una clase y determina cómo tiene que actuar el objeto cuando recibe el mensaje vinculado con ese método. A su vez, un método puede enviar mensajes a otros objetos solicitando una acción o información. Los atributos definidos en la clase permitirán almacenar información para dicho objeto. (Ceballos, 2010).

Cuando se modela pensando en objetos, es necesario tomar las características y propiedades de un ente real, y llevarlo a un objeto. El término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?" (Característica de caja negra) (Di Serio, 2011).

La modularidad dentro de Java y Groovy se organiza de forma lógica en clases y paquetes y de forma física mediante archivos.

Clases:

- Encapsulan los atributos y métodos de un tipo de objetos en un solo compartimiento.
- Ocultan, mediante los especificadores de acceso, los elementos internos que no se pretende publicar al exterior.

Paquetes:

- Son unidades lógicas de agrupación de clases.
    - Las clases públicas forman parte de la interfaz del paquete y son visibles fuera del mismo.
    - Las clases que no son públicas sólo son visibles dentro del propio paquete.

Archivos:

- Dentro de los archivos pueden residir varias clases con ciertas restricciones que más adelante serán vistas. (TutorialesNET, 2014)

El manejo de módulos permite una mejor estructura de los programas, reduce la complejidad de los mismos, permite crear una serie de fronteras bien definidas lo cual aumenta su comprensión.

En Java es conveniente que cada clase se coloque en un archivo. El nombre del archivo tiene el nombre de la clase, si se tiene más de una clase en un archivo, el nombre del archivo será el de la clase que tiene el método *main*.

Únicamente puede haber una clase pública por archivo, el nombre de éste debe coincidir con el de la clase pública. Puede haber más de una clase default en el mismo archivo.

Los nombres de clase deben ser sustantivos, en mayúsculas y minúsculas con la primera letra de cada palabra interna en mayúscula. (Oracle, 2014).

Este mecanismo de nombre es llamado CamelCase[1] (Por ejemplo: NombreClase, CuentaUsuario, Factura).

Groovy, a comparación de Java, permite los scripts. Los scripts son programas, usualmente pequeños o simples, para realizar generalmente tareas muy específicas. Son un conjunto de instrucciones habitualmente almacenadas en un archivo de texto que deben ser interpretados línea a línea en tiempo real para su ejecución; esto los distingue de los programas (compilados), pues éstos deben ser convertidos a un archivo binario ejecutable (por ej: ejecutable .exe, entre otros) para poder correrlos. (Alegsa)

Lo que es lo mismo, no es necesario poner todo el código Groovy en una clase. Por lo que en Groovy, si el código a implementar es de pocas líneas, se puede ejecutar en un simple script, la creación de la clase vendrá de acuerdo al tamaño del programa, en el momento en que se necesiten más variables, instancias, etc., es cuando se necesitará la realización de una clase (Kousen, 2014, pág. 19).

Notar dos diferencias adicionales manejadas en la sintaxis Groovy respecto de Java:

- *Los puntos y comas son opcionales.*
- *Los paréntesis son a menudo opcionales.* No está mal incluirlos si lo desea.

Para Groovy la nomenclatura depende de algunas características en base al manejo de clases, scripts o ambos en un solo archivo. A continuación se muestran las bases para efectuar esta parte:

Archivo para relación de clase Groovy

La relación entre los archivos y las declaraciones de clase no es tan fija como en Java. Archivos Groovy pueden contener cualquier número de declaraciones de clases públicas de acuerdo con las siguientes reglas:

[1] **CamelCase** es un estilo de escritura que se aplica a frases o palabras compuestas. El nombre se debe a que las mayúsculas a lo largo de una palabra en CamelCase se asemejan a las jorobas de un camello. El nombre CamelCase se podría traducir como *Mayúsculas/Minúsculas Camello*.

- Si un archivo Groovy no contiene declaración de la clase, éste se maneja como un script; es decir, que se envuelve de forma transparente en una clase de tipo Script. Esta clase generada automáticamente tiene el mismo nombre que el nombre del archivo script de origen (sin la extensión). El contenido del archivo se envuelve en un método run, y un método *main* adicional se construye fácilmente a partir de la secuencia de comandos.
- Si un archivo Groovy contiene exactamente una declaración de la clase con el mismo nombre que el archivo (sin la extensión), entonces es la misma relación de uno-a-uno como en Java.
- Un archivo Groovy puede contener *múltiples* declaraciones de clase de cualquier visibilidad, y no hay ninguna regla impuesta de que alguno de ellos deba coincidir con el nombre de archivo. El compilador groovyc felizmente crea archivos *.class para todas las clases declaradas en dicho archivo.
- Un archivo Groovy puede *mezclar* declaraciones de clase y código script. En este caso, el código script se convertirá en la clase principal para ser ejecutado (König & Glover, 2007, págs. 188-189).
- Si el archivo contiene una clase junto con un script, y la clase corresponde exactamente al nombre del archivo, el compilador Groovy cortésmente sugiere cambiar el nombre ya sea de script o el nombre de la clase, ya que no puede generar todos los archivos de clases requeridas.

Como regla general, sólo se sugiere utilizar este modo mixto de codificación cuando se están escribiendo scripts independientes. Al escribir código Groovy que se integra en una aplicación más grande, es mejor quedarse con la forma Java de hacer las cosas, de modo que los nombres de archivo de origen correspondan a los nombres de las clases que se implementan como parte de su solicitud. (Dearle, 2010, págs. 34-35). Un punto importante es que el al igual que Java, el nombre del archivo es obligatorio comenzar con una letra mayúscula.

B. MÉTODOS

Las clases están formadas por variables de instancia y métodos. El concepto de método es muy amplio ya que Java les concede una gran potencia y flexibilidad.

En Java los únicos elementos necesarios de una declaración de método son el tipo del método de devolución, nombre, un par de paréntesis "()", y un cuerpo entre llaves "{}".

Más general, las declaraciones de métodos tienen los siguientes componentes:

- Modificadores - como público, privado y otros.
- El tipo de retorno - tipo de datos del valor devuelto por el método, o *void* si el método no devuelve un valor.
- El nombre del método.
- Lista de parámetros entre paréntesis, una lista delimitada por comas de parámetros de entrada, precedida por sus tipos de datos, encerrados entre paréntesis, (). Si no hay parámetros, debe utilizar paréntesis vacíos.
- El cuerpo del método, encerrado entre llaves-código del método, incluida la declaración de variables locales.
- Modificadores, regresan tipos y parámetros. (Oracle)

Respecto a Groovy los modificadores habituales de Java pueden ser utilizados; declarar un tipo de retorno es opcional; y, si no se suministran modificadores o tipo de retorno, la palabra clave *def* llena el agujero. Cuando se utiliza la palabra clave *def*, el tipo de retorno se considerará sin tipo. En este caso, bajo las sábanas, el tipo de retorno será java.lang.Object. La visibilidad por defecto de los métodos es *public*. (König & Glover, 2007, pág. 180)

Se debe utilizar la palabra clave *def* al definir un método que tiene un tipo de retorno dinámico.

En Groovy no se usa la palabra *return* como lo hace Java para retornar un valor o referencia a objeto. Groovy por default siempre retorna la última línea del cuerpo, por lo cual no es necesario especificarlo explícitamente como en Java. Se puede manejar explícitamente sin ningún problema, pero Groovy da la comodidad de omitirlo.

En Java el método principal tiene la siguiente forma:

public static void main(String args[]) { }

En esta línea comienza la ejecución del programa. Todos los programas de Java comienzan la ejecución con la llamada al método *main()*.

El intérprete o máquina virtual de Java llama a *main()* antes de que se haya creado objeto alguno. La palabra clave *void* simplemente indica al compilador que *main()* no devuelve ningún valor. (Schildt, 2009, pág. 23)

En el método *main()* sólo hay un parámetro, aunque complicado. *String args[ ]* declara un parámetro denominado *args*, que es un arreglo de instancias de la clase *String* (los *arreglos* son colecciones de objetos similares). Los objetos del tipo *String* almacenan cadenas de caracteres.

En este caso, *args* recibe los argumentos que estén presentes en la línea de comandos cuando se ejecute el programa.

El último carácter de la línea es *{*. Este carácter señala el comienzo del cuerpo del método *main( )*. Todo el código comprendido en un método debe ir entre la llave de apertura del método y su correspondiente llave de cierre. El método *main( )* es simplemente un lugar de inicio para el programa. Un programa complejo puede tener una gran cantidad de clases, pero sólo es necesario que una de ellas tenga el método *main( )* para que el programa comience. (Schildt, 2009, pág. 24)

Groovy maneja el método principal de una manera más sencilla. Su forma es la siguiente:

static main (args){ }

El método *main* tiene algunos toques interesantes. En primer lugar, el modificador *public* puede ser omitido, ya que el valor es predeterminado. En segundo lugar, *args* generalmente tiene que ser de tipo *String[]* con el fin de hacer el método principal para iniciar la ejecución de la clase. Gracias al método de envío de Groovy, funciona de todos modos, aunque *args* es ahora implícitamente de tipo estático *java.lang.Object*. En tercer lugar, debido a que los tipos de retorno no se utilizan para el envío, es posible omitir la declaración *void*. (König & Glover, 2007, pág. 180)

Parámetros

Tomando en cuenta la Figura 1, se puede observar que Groovy no necesita declarar el tipo de dato junto con la variable que va a llegar, mientras que en Java se debe definir en forma estricta.

Código Groovy:

```
def comer(alimento){
    "le gusta comer ${alimento}"
}
```

Figura 1. Parámetro sin tipo de variable

En la Figura 2 se muestra la forma Java:

```
public String comer(String alimento){
    return "le gusta comer" + alimento;
}
```

Figura 2. Código Java con tipo de parámetro necesario

En el código Groovy sólo es necesario declarar la variable que va a llegar. No olvidando resaltar lo dinámico, en el ejemplo manejado puede llegar tanto un entero como un String sin ningún problema.

C. POLIMORFISMO

El polimorfismo es la manera de invocar una acción a que tenga distintos comportamientos dependiendo del contexto con el que se utiliza. Más simple, "el polimorfismo es la posibilidad que un método (función) pueda tener distinto comportamiento a partir de los parámetros que se envían (*sobrecarga*), o a partir de la manera que se invoca (*sobre escritura*)". (Hdeleon, 2014)

Cualquier objeto que pase el test IS-A[2] puede ser polimórfico.

Sobrecarga de métodos

Es una característica que hace a los programas más legibles. Consiste en volver a declarar un método ya declarado, con distinto número y/o tipo de parámetros. Un método sobrecargado no puede diferir solamente en el tipo de resultado, sino que debe diferir también en el tipo y/o en el número de parámetros formales. (Ceballos, 2010)

Constructores

Para crear un objeto se requiere la palabra clave "new", por ejemplo:

variable = new *nombre_de_clase();*

Resulta más evidente la necesidad de los paréntesis después del nombre de clase. Lo que ocurre realmente es que se está llamando al constructor de la clase.

Un constructor se utiliza en la creación de un objeto que es una instancia de una clase. Normalmente lleva a cabo las operaciones necesarias para inicializar el objeto antes de que los métodos se invoquen o se acceda a los campos. Los constructores nunca se heredan. (Oracle)

El constructor tiene el mismo nombre que la clase en la que reside y, sintácticamente, es similar a un método. Una vez definido, se llama automáticamente al constructor después de crear el objeto y antes de que termine el operador *new*. Los constructores resultan un poco diferentes a los métodos convencionales, porque no devuelven ningún tipo, ni siquiera *void*.

[2] Hace referencia a la herencia de clases o implementación de interfaces. Es como decir "A es un tipo B". La herencia es uni-direccional. Por ejemplo Casa es un Edificio. Pero edificio no es una casa.

Cuando no se define explícitamente un constructor de clase, Java crea un constructor de clase por defecto.

Para clases sencillas, resulta suficiente utilizar el constructor por defecto, pero no para clases más sofisticadas. Una vez definido el propio constructor, el constructor por omisión ya no se utiliza. (Schildt, 2009, págs. 117-119). Lo mencionado anteriormente se hace referenciado a Java y Groovy.

La declaración de un constructor diferente del constructor por defecto, obliga a que se le asigne el mismo identificador que la clase y que no se indique de forma explícita un tipo de valor de retorno. La existencia o no de parámetros es opcional. Por otro lado, la *sobrecarga* permite que puedan declararse varios constructores (con el mismo identificador que el de la clase), siempre y cuando tengan un tipo y/o número de parámetros distintos. (García Beltrán & Arranz)

En Groovy, al igual que los métodos, el constructor es *public* por defecto. Se puede llamar al constructor de tres maneras diferentes: la forma habitual de Java (uso normal de constructor de la Figura 3), con el tipo de restricción forzada mediante el uso de la palabra clave *as*, y con tipo de restricción implícita.

```
class VendorWithCtor {
    String name, product

    VendorWithCtor(name, product) {
        this.name    = name
        this.product = product
    }
}

def first = new VendorWithCtor('Canoo','ULC')

def second = ['Canoo','ULC'] as VendorWithCtor

VendorWithCtor third = ['Canoo','ULC']
```

Definición de constructor

Uso normal de constructor

❶ Resticción con as

❷ Restricción en asignación

Figura 3. Llamada de constructores con parámetros posicionales

La restricción en los números 1 y 2 pueden ser llamativas. Cuando Groovy ve la necesidad de restringir una lista a algún otro tipo, trata de llamar al constructor del tipo con todos los argumentos suministrados por la lista, en el orden de la lista. Esta necesidad de restringir se puede hacer cumplir con la palabra clave *as* o puede surgir de las asignaciones a las referencias de tipos estáticos.

as: Permite cambiar el tipo de objeto.

Los parámetros con nombre en constructores son útiles. Un caso de uso que surge con frecuencia es la creación de *clases inmutables* que tienen algunos parámetros que son opcionales. El uso de los parámetros de posición se convertiría rápidamente engorroso porque se tendría que tener constructores que permiten todas las combinaciones de los parámetros opcionales.

Por ejemplo, si se desean dos constructores con un argumento tipo cadena de caracteres, no se podrían tener constructores con un solo argumento, porque no se distinguiría si va a poner *name* o el atributo *product* (ambas son cadenas). Se solicitaría un argumento extra para la distinción, o se tendrían que escribir fuertemente los parámetros. Para evitar lo anterior, Groovy se apoya en parámetros con nombre. En la figura 4 se muestra cómo utilizar parámetros con nombre con una versión simplificada de la clase *Vendor*. Se basa en el constructor predeterminado implícito. (König & Glover, 2007, págs. 186-187)

```
class Vendor {
    String name, product
}

new Vendor()
new Vendor(name:    'Canoo')
new Vendor(product:'ULC')
new Vendor(name:    'Canoo', product:'ULC')

def vendor = new Vendor(name: 'Canoo')
assert 'Canoo' == vendor.name
```

Figura 4. Parámetros nombrados

El ejemplo de la Figura 4 muestra cómo los parámetros con nombre son flexibles para sus constructores. (König & Glover, 2007, págs. 186-187)

De igual manera permite pasar un mapa al constructor de un bean que contiene los nombres de las propiedades, junto con un valor de inicialización asociado:

```
map = [id: 1, name: "Barney, Rubble"]
customer1 = new Customer( map )
customer2 = new Customer( id: 2, name: "Fred, Flintstone")
```

Figura 5. Pasar un mapa a un constructor

Al pasar "map" directamente al constructor Customer, se permite omitir "map" del paréntesis, como se muestra en la inicialización de customer2.

Cada GroovyBean[3] cuenta por defecto con este constructor incorporado *Map*. Este constructor trabaja iterando el objeto *map* y llama a la propiedad correspondiente *setter*[4] para cada entrada en el *map*. Cualquier entrada de *map* que no corresponde a una propiedad real del bean causará una excepción al ser lanzado. (Dearle, 2010, pág. 41).

Groovy permite declarar 2 constructores con el mismo tipo de argumentos, y se ayuda gracias a que los parámetros son nombrados para no causar error. Además se puede percatar de que es posible omitir la definición del constructor.

En Java, un *JavaBean* es una clase que implementa métodos *getters* y *setters* para todos o algunos de sus campos de instancia. Groovy genera automáticamente *getters* y *setters* de campos de instancia de una clase y tienen la visibilidad predeterminada *public*. También genera el constructor por defecto. Campos de instancia que tienen getters y setters generados automáticamente son conocidos en Groovy como *propiedades*, y se refiere a estas clases como *GroovyBeans*, o por el coloquial *POGO* (Plain Old Groovy Object).

Los *closures* son fragmentos de código anónimos que se pueden asignar a una variable y tienen características que los hacen parecer como un método en la medida en que permite pasar parámetros a ellos, y ellos pueden devolver un valor. Sin embargo, a diferencia de los métodos, los *closures* son anónimos, no están pegados a un objeto. Un *closure* es sólo un fragmento de código que se puede asignar a una variable y ejecutar más tarde. (Dearle, 2010)

No necesariamente deben ser declarados dentro de las clases, se pueden declarar en cualquier lugar.

Un closure Groovy es código envuelto como un objeto de tipo groovy.lang.Closure, definido y reconocido por llaves *{// código aquí}*. Un *closure* tiene gran similitud con los métodos de Java, el manejo de parámetros lo puede realizar con tipos dinámicos. Esta característica de Groovy es una de las más relevantes del lenguaje, la cual marca una diferencia en cuanto a funcionalidad respecto a Java. Un *closure* puede ser pasado como parámetro a otro *closure*, opción que Java permite al trabajar con sus respectivos métodos, cabe resaltar que cuando un *closure* es pasado como parámetro entre los paréntesis, éste debe ir declarado como último argumento.

[3] Un **Bean** es un componente software que tiene la particularidad de ser reutilizable y así evitar la tediosa tarea de programar los distintos componentes uno a uno.

[4] Los getters y setters son métodos de acceso lo que indica que son siempre declarados públicos. Los setters nos sirve para asignar un valor inicial a un atributo, pero de forma explícita, además el Setter nunca retorna nada. Los getters nos sirve para obtener (recuperar o acceder) el valor ya asignado a un atributo y utilizarlo para cierto método

Es posible determinar si el *closure* ha sido proporcionado. De lo contrario, se puede decidir utilizar una implementación por defecto para manejar el caso. (Subramaniam, 2013)

```
3  class CDinamico {
4      static main(args){
5          doSomeThing() { println "Use specialized implementation" }
6          doSomeThing({ println "Use specialized implementation" })
7          doSomeThing()
8      }
9      def static doSomeThing(closure) {
10         if (closure) {
11             closure()
12         } else {
13             println "Using default implementation"
14         }
15     }
16 }
```

Figura 6. Closure dinámico

El código de la Figura 6 muestra lo anterior donde se manda a llamar el closure 3 veces, la primera y segunda con un valor, y la tercera sin ello, el *if* recibirá la llamada de las líneas 5 y 6, mientras que el *else* ejecutará la llamada de la línea 7 puesto que no manda algún valor.

En la línea 5 pareciera que se está implementando el cuerpo de un método, pero no lo es, la línea 6 pasa el valor entre los paréntesis, ambas líneas son equivalentes.

D. ENCAPSULACIÓN

La encapsulación permite controlar la forma en que se utilizan los datos y los métodos. Puede utilizar modificadores de acceso para evitar que los métodos externos ejecuten métodos de clase o lean y modifiquen sus atributos. Para permitir que otros objetos consulten o modifiquen los atributos de los objetos, las clases suelen presentar métodos de acceso. (UNAM, 2013)

Se tienen los siguientes niveles de acceso (siendo de menor a mayor):

- **private:** acceso únicamente desde la propia clase.
- **default:** acceso desde la propia clase y desde el mismo paquete.
- **protected:** acceso desde la propia clase, paquete y subclase.
- **public:** acceso desde cualquier paquete.

Nota: En Java, hay 4 niveles de acceso, pero solo 3 modificadores de acceso. Las clases únicamente pueden ser public o default. (Horna, 2010)

Dentro del código Java únicamente la clase principal debe ser *public* cuando hay más de una clase dentro del mismo archivo.

Un objeto proporciona una frontera bien definida alrededor de una sola abstracción, y tanto el encapsulamiento como la modularidad proporcionan barreras que rodean a esa abstracción. (Booch, 1991)

Java requiere siempre expresar explícitamente la visibilidad pública de una clase. De forma predeterminada, a menos que especifique lo contrario, por defecto en Groovy todas las clases, propiedades y métodos son de acceso *public* no importando cual tenga el método *main*. (Judd & Faisal Nusairat, 2008, pág. 23) .

E. JERARQUIA

Frecuentemente un conjunto de abstracciones forma una jerarquía, y la identificación de esas jerarquías en el diseño simplifica en gran medida la comprensión del problema.

(Booch, 1991) Define el término de la siguiente manera:

> *La jerarquía es una clasificación u ordenación de abstracciones.*

La herencia es la jerarquía de clases más importante y es un elemento esencial de los sistemas orientados a objetos. (Booch, 1991)

Si una clase sólo puede recibir características de otra clase base, la herencia se denomina herencia simple. Pero si una clase recibe propiedades de más de una clase base, la herencia se denomina herencia múltiple. (Di Serio, 2011).

En la terminología de Java, una clase que es heredada se denomina *superclase*. La clase que hereda se denomina *subclase*. Por lo tanto, una subclase es una versión especializada de una superclase, que hereda todas las variables de instancia y métodos definidos por la superclase y añade sus propios elementos.

Para heredar una clase, simplemente se incorpora la definición de una clase dentro de la otra usando la palabra clave **extends**. (Schildt, 2009, pág. 157)

La palabra clave *super* tiene dos formas generales. La primera llama al constructor de la superclase. La segunda se usa para acceder a un miembro de la superclase que ha sido escondido por un miembro de una subclase.

La *lista de parámetros* especifica cualquier parámetro que el constructor necesite en la superclase. *super()* debe ser siempre la primera sentencia que se ejecute dentro de un constructor de la subclase. (Schildt, 2009, págs. 162-163)

A veces se requiere llamar a un método de la superclase. Eso se realiza con la palabra reservada *super*. Si *this* hace referencia a la clase actual, *super* hace referencia a la superclase respecto a la clase actual, con lo que es un método imprescindible para poder acceder a métodos anulados por herencia.

Por defecto Java realiza estas acciones:

- Si la primera instrucción de un constructor de una subclase es una sentencia que no es ni *super* ni *this*, Java añade de forma invisible e implícita una llamada *super()* al constructor por defecto de la superclase, luego inicia las variables de la subclase y luego sigue con la ejecución normal.
- Si se usa *super(…)* en la primera instrucción, entonces se llama al constructor seleccionado de la superclase, luego inicia las propiedades de la subclase y luego sigue con el resto de sentencias del constructor.

Finalmente, si esa primera instrucción es *this(...)*, entonces se llama al constructor seleccionado por medio de *this*, y después continúa con las sentencias del constructor. La inicialización de variables la habrá realizado el constructor al que se llamó mediante *this*. (Marin, 2012)

Todas las características de herencia de Java (incluyendo clases abstractas) están disponibles en Groovy.

Debido a que Java no soporta el concepto de herencia múltiple como otros lenguajes Orientados a Objetos, éste hace una simulación, presenta el concepto de interfaces como una alternativa a la herencia múltiple, son bastante diferentes, a pesar de que las interfaces pueden resolver problemas similares. En particular:

- Desde una interfaz, una clase sólo hereda constantes.
- Desde una interfaz, una clase no puede heredar definiciones de métodos.
- La jerarquía de interfaces es independiente de la jerarquía de clases. Varias clases pueden implementar la misma interfaz y no pertenecer a la misma jerarquía de clases. En cambio, cuando se habla de herencia múltiple, todas las clases pertenecen a la misma jerarquía. (Ceballos, 2010)

En Groovy, para manejar la de Herencia Múltiple hace uso de un concepto llamado "*Mixins*". Los mixins se usan para poder inyectar el comportamiento (métodos) de una o más clases en otra. Normalmente se utilizan con la anotación @Mixin. (emalvino, 2013)

Sobre escritura de métodos

Cuando utilizamos la herencia, al heredar de una clase podemos utilizar sus métodos, y puede haber ocasiones en las cuales el método del padre no sea de nuestra utilidad y debamos crear uno nuevo con el mismo nombre, para ello utilizamos la *Sobreecritura***.** (Hdeleon, 2014)

Dentro de Java al reemplazar un método, es posible que desee utilizar la anotación *@Override* que instruye al compilador que tiene la intención de sustituir un método en la superclase. Si, por alguna razón, el compilador detecta que el método no existe en una de las superclases, entonces se generará un error. (Oracle, 2014)

A diferencia de Java, Groovy no maneja la anotación *@Override* para sobrescribir, este utiliza la propiedad *MetaClass y el operador "="*. (EduSanz, 2010) Usando la propiedad MetaClass es posible añadir nuevos métodos o reemplazar los métodos existentes de la clase. Si se desea añadir nuevos métodos que tienen el mismo nombre, pero con diferentes argumentos, se puede utilizar una notación de acceso directo. Groovy utiliza para esto leftShift() (<<). (Klein, 2009)

F. TIPOS (TIPIFICACIÓN)

Los lenguajes O.O. hacen (por sencillez) que no haya prácticamente ninguna diferencia entre tipos y clases, no obstante, formalmente la diferenciación que se suele hacer subraya que un tipo especifica una semántica[5] y una estructura y una clase es una implementación concreta de un tipo.

En la P.O.O. las clases suelen definir un tipo, y aparte existen los tipos básicos. En Java, para que podamos tratarlos como clases cuando lo necesitamos (por ejemplo, para poder meterlos donde se requiere un Object), se definen clases contenedoras (del inglés wrapper classes). (Castro Souto, 2001)

Las ventajas de ser estrictos con los tipos son:

- Fiabilidad, pues al detectar los errores en tiempo de compilación se corrigen antes.

[5] La semántica es la forma correcta en que se escribe una instrucción en un lenguaje de programación.

- Legibilidad, ya que proporciona cierta "documentación" al código.
- Eficiencia, pueden hacerse optimizaciones. (Castro Souto, 2001)

En base al concepto de tipos, se manejan un par de conceptos más que son de apoyo para clasificar los lenguajes. Estos son, comprobación y ligadura de tipos, los cuales se definirán a continuación:

En la *comprobación de tipos* se asegura de que el tipo de una construcción coincida con el previsto en su contexto. Hay dos maneras de manejar la comprobación de tipos:

- Comprobación estricta de tipos: Requiere que todas las expresiones de tipos sean consistentes en tiempo de compilación. Los lenguajes que manejan este tipo son Java, C++, Object Pascal. (Castro Souto, 2001)
- Comprobación débil de tipos: Todas las comprobaciones de tipos se hacen en tiempo de ejecución (se dan, por tanto, un mayor número de excepciones). Es decir, la flexibilidad es mucho mayor, pero nos topamos con muchos más problemas a la hora de ejecutar. Ejemplo: Smalltalk. (Castro Souto, 2001)

Existen varios beneficios importantes que se derivan del uso de lenguajes con tipos estrictos.

- "Sin la comprobación de tipos, un programa puede estallar de forma misteriosa en la ejecución.
- En la mayoría de los sistemas, el ciclo editar-compilar-depurar es tan tedioso que la detección temprana de errores es indispensable.
- La declaración de tipos ayuda a documentar los programas.
- La mayoría de los compiladores pueden generar un código más eficiente si se han declarado los tipos." (Tesler, 1981)

Con lo citado anteriormente, y en base a conocimientos previos en la manera como trabajan Java y Groovy este aspecto, se puede concluir que *Java es un lenguaje de comprobación estricta de tipos*, puesto que al ir codificando, y en caso de que se le llegase a asignar a un tipo un valor diferente respecto al tipo, de inmediato el compilador marca un error, advirtiendo que algo está mal y que debe revisarse o de lo contrario la ejecución no será posible.

Groovy, por el contrario, es un lenguaje con comprobación débil de tipos, éste al asignar un valor diferente del tipo asignado respectivamente, no advierte nada y llegado el momento de ejecución es cuando envía los errores.

En pocas palabras, mientras que Java advierte los errores respecto a tipos en tiempo de compilación, Groovy lo hace en tiempo de ejecución.

Ligadura de tipos

Ligadura es el proceso de asociar un atributo a un nombre en el caso de funciones, el término ligadura (binding) se refiere a la conexión o enlace entre una llamada a la función y el código real ejecutado como resultado de la llamada. (Joyanes Aguilar, 1996)

La ligadura se clasifica en dos categorías: ligadura estática y ligadura dinámica. (Booch, 1991)

- Ligadura Estática: la asignación estática de tipos -también conocida como ligadura estática o ligadura temprana- se refiere al momento en el que los nombres se ligan con sus tipos. La ligadura estática significa que se fijan los tipos de todas las variables y expresiones en tiempo de compilación.
- Ligadura Dinámica: también llamada ligadura tardía, significa que los tipos de las variables y expresiones no se conocen hasta el tiempo de ejecución.

Existe el polimorfismo cuando interactúan las características de la herencia y el enlace dinámico. Es quizás la característica más potente de los lenguajes orientados a objetos después de su capacidad para soportar la abstracción, y es lo que distingue la programación orientada a objetos de otra programación más tradicional con tipos abstractos de datos. (Booch, 1991)

Dicho lo anterior, tanto Java como Groovy son lenguajes con ligadura tardía, ambos soportan el polimorfismo.

Tipeado estático

En está tipificación, cada variable debe ser declarada con un tipo asociado a ella.

Java es un lenguaje fuertemente tipificado. Parte de su seguridad y robustez se debe a este hecho.

En primer lugar, cada variable y cada expresión tienen un tipo, y cada tipo está definido estrictamente. En segundo lugar, en todas las asignaciones, ya sean explícitas o por medio del paso de parámetros en la llamada a un método, se comprueba la compatibilidad de tipos. En Java no existen conversiones automáticas de tipos incompatibles como en algunos otros lenguajes. El compilador de Java revisa todas las expresiones y parámetros para asegurarse de que los tipos son compatibles.

Java define ocho tipos *primitivos*: *byte, short, int, long, char, float, double* y *boolean*. Los tipos primitivos son llamados también tipos *simples*. (Schildt, 2009)

Cualquier variable utilizada en un código java exige declararle un tipo de dato, de lo contrario el compilador hará su trabajo resaltando ese error al codificar.

Tipado dinámico

Este tipo de tipificación no requiere que a la variable se le asigne algún tipo de dato como podría ser int, String, etc., tal como lo haría Java, simplemente basta con asignar valores a estas variables y su tipo de dato va a ser considerado dependiendo del valor que se asigne a la variable.

SEGURIDAD EN GROOVY

Independientemente de si el tipo de una variable se declara explícitamente o no, el sistema es de tipo seguro. A diferencia de lenguajes sin tipo, Groovy no permite el tratamiento de un objeto de un tipo como una instancia de un tipo diferente, sin ser una conversión bien definida disponible. Nunca se podría tratar un java.lang.String con valor "1", como si se tratara de un java.lang.Number. Ese tipo de comportamiento sería peligroso - es la razón por la que Groovy no permite esto más de lo que hace Java. (König & Glover, 2007)

Para Groovy definir un tipo de dato primitivo a una variable como lo hace Java es muy válido, sin en cambio, para hacer todo más sencillo, Groovy facilita su manejo, sin olvidar el tema principal referido a la manera en que este lenguaje opera todo como un objeto.

Los diseñadores de Groovy decidieron acabar con tipos primitivos. Cuando Groovy necesita almacenar valores que han utilizado los tipos primitivos de Java, Groovy utiliza las clases wrapper ya proporcionadas por la plataforma Java.

Cada vez que se vea lo que parece ser un valor primitivo (por ejemplo, el número 5 en el código fuente Groovy, ésta es una referencia a una instancia de la clase wrapper apropiada). Por el bien de la brevedad y la familiaridad, Groovy permite declarar las variables como si fueran variables de tipo primitivo.

La conversión de un valor simple en una instancia de un tipo de wrapper se llama boxing en Java y otros lenguajes que soportan la misma noción. La acción inversa - teniendo una instancia de un wrapper y recuperar el valor simple - se llama unboxing. Groovy realiza estas

operaciones de forma automática en caso necesario. Este boxing y unboxing automático se conoce como autoboxing.

Todo esto es transparente – no se tiene que hacer nada en el código Groovy para habilitarlo.

Debido a lo anterior se puede afirmar que Groovy es más orientado a Objetos que Java. (König & Glover, 2007)

G. CONCURRENCIA

Para ciertos tipos de problema, un sistema automatizado puede tener que manejar diferentes eventos en forma simultánea. Otros problemas pueden implicar tantos cálculos que excedan la capacidad de cualquier procesador individual. En ambos casos, es natural considerar el uso de un conjunto distribuido de computadores para la implantación que se persigue o utilizar procesadores capaces de realizar multitarea. Un solo proceso -denominado *hilo de control*- es la raíz a partir de la cual se producen acciones dinámicas independientes dentro del sistema.

Los sistemas que se ejecutan en múltiples CPU's permiten hilos de control verdaderamente concurrentes, mientras que los sistemas que se ejecutan en una sola CPU sólo pueden conseguir la ilusión de hilos de control concurrentes.

Existen 2 tipos de concurrencia, concurrencia pesada y ligera. Un proceso pesado es el manejado de forma independiente por el sistema operativo de destino y abarca su propio espacio de direcciones. Un proceso ligero suele existir dentro de un solo proceso del sistema operativo en compañía de otros procesos ligeros, que comparten el mismo espacio de direcciones y suelen involucrar datos compartidos.

Mientras que la programación orientada a objetos se centra en la abstracción de datos, encapsulamiento y herencia, la concurrencia se centra en la abstracción de procesos y la sincronización. El objeto es un concepto que unifica estos dos puntos de vista distintos: cada objeto (extraído de una abstracción del mundo real) puede representar un hilo separado de control (una abstracción de un proceso). (Booch, 1991)

En base a lo anterior Grady Booch define a la concurrencia como sigue:

> *"La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo."*

Como se puede observar, se manejan los conceptos *hilo* y *proceso*, que a continuación serán definidos para entender mejor el tema.

- Hilo: conocido también como proceso ligero o thread. Son pequeños procesos o piezas independientes de un gran proceso. También se puede decir, que un hilo es un flujo único de ejecución dentro de un proceso.
- Proceso: es un programa ejecutándose dentro de su propio espacio de direcciones. (Cisneros, 2010)

Java da soporte al concepto de *Thread* desde el propio lenguaje, con algunas clases e interfaces definidas en el paquete *java.lang* y con métodos específicos para la manipulación de *Threads* en la clase *Object*.

Se puede definir e instanciar un hilo (thread) de dos maneras:

- Extendiendo de la clase java.lang.Thread
- Implementando la interfaz Runnable

Desde Groovy se pueden utilizar todas las facilidades normales que ofrece Java para la concurrencia, sólo que Groovy facilita el trabajo como se verá enseguida. (König & Glover, 2007)

La primera y principal característica de Groovy para apoyo multihilo es que Closure es implementado en Runnable. Esto permite las definiciones simples de hilos como:

```
t = new Thread() { /* Closure body */ }
t.start()
```

Esto incluso se puede simplificar con nuevos métodos estáticos en la clase Thread:

```
Thread.start { /* Closure body */ }
```

Java tiene el concepto de un hilo demonio, y por lo tanto también lo hace Groovy. Un hilo demonio se puede iniciar a través de:

```
Thread.startDaemon { /* Closure body */ }
```

H. PERSISTENCIA

La persistencia es la propiedad de un objeto a través del cual su existencia trasciende en el tiempo y/o espacio. Esto significa que un objeto persistente sigue existiendo después de que ha finalizado el programa que le dio origen y que además puede ser movido de la localidad de memoria en la que fue creado. (Ortiz, 2014)

Se sugiere que hay un espacio continuo de existencia del objeto, que va desde los objetos transitorios que surgen en la evaluación de una expresión hasta los objetos de una base de datos que sobreviven a la ejecución de un único programa. Este aspecto de persistencia abarca lo siguiente:

- Resultados temporales en la evaluación de expresiones.
- Variables locales en la activación de procedimientos.
- Variables propias, variables globales y elementos del montículo (heap) cuya duración difiere de su espacio.
- Datos que existen entre ejecuciones de un programa.
- Datos que existen entre varias versiones de un programa.
- Datos que sobreviven al programa.

Los lenguajes de programación tradicionales suelen tratar solamente los tres primeros tipos de persistencia de objetos; la persistencia de los tres últimos tipos pertenece típicamente al dominio de las tecnologías de las bases de datos. (Booch, 1991)

El atributo de persistencia solamente debe estar presente en aquellos objetos que una aplicación requiera mantener entre corridas, de otra forma se estarían almacenando una cantidad probablemente enorme de objetos innecesarios. La persistencia se logra almacenando en un dispositivo de almacenamiento secundario (disco duro, memoria flash) la información necesaria de un objeto para poder restaurarlo posteriormente. Típicamente la persistencia ha sido dominio de la tecnología de base de datos, por lo que esta propiedad no ha sido sino hasta recientemente que se ha incorporado en la arquitectura básica de los lenguajes orientados a objetos.

El lenguaje de programación Java permite serializar objetos en un flujo de bytes. Dicho flujo puede ser escrito a un archivo en disco y posteriormente leído y deserializado para reconstruir el objeto original. Con esto se logra lo que se llama "persistencia ligera" (*lightwigth persistence* en inglés). (Ortiz, 2014)

La *serialización* de un objeto consiste en obtener una secuencia de bytes que represente el estado de dicho objeto. Esta secuencia puede utilizarse de varias maneras (puede enviarse a través de la red, guardarse en un fichero para su uso posterior, utilizarse para recomponer el objeto original, etc.).

Objeto serializable

Un objeto serializable es un objeto que se puede convertir en una secuencia de bytes. Para que un objeto sea serializable, debe implementar la interfaz *java.io.Serializable*. Esta interfaz no define ningún método. Simplemente se usa para 'marcar' aquellas clases cuyas instancias pueden ser convertidas a secuencias de bytes (y posteriormente reconstruidas). Objetos tan comunes como String, Vector o ArrayList implementan Serializable, de modo que pueden ser serializados y reconstruidos más tarde. (Miedes, 2014)

La serialización y la deserialización se pueden realizar tanto en Java como en Groovy.

En Java, el manejo de archivos se hace a través de la clase *File* usando el paquete *java.io*, el cual proporciona soporte a través de métodos para las operaciones de Entrada/Salida.

El enfoque sobre la manera en que Groovy hace el manejo de archivos, comparado con Java, no hace mucha diferencia. En primer lenguaje hace uso del mismo paquete con el que trabaja Java, sólo que agrega varios métodos de conveniencia y como siempre, hace reducción de código a las líneas necesarias para trabajar.

El JDK de Java, aborda esta necesidad con sus paquetes java.io y java.net. Proporciona apoyo elaborado con las clases File, URL y numerosas versiones de flujos, readers y writers.

Más sin en cambio, Groovy extiende el JDK de Java con su propio GDK (Groovy Development Kit). El GDK tiene una serie de métodos que hacen la magia para trabajar de una manera más fácil.

En Groovy las características que se pueden observar a simple vista son las siguientes:

- No se necesita hacer el manejo de excepciones.
- El paquete *java.io* es importado automáticamente por el GDK, no haciendo necesario su uso explícito.
- No es necesario el manejo del *BufferedReader* y el *FileReader*.

- Sólo se utiliza la clase *File*, la cual es únicamente la representación de un archivo y rutas de directorio.

La eliminación de las características anteriores reduce en varias líneas el código, permitiendo una codificación más rápida y un código más legible.

El JDK Groovy se puede consultar en su sitio oficial. (Groovy)

I. MANEJO DE EXCEPCIONES

Una excepción es un evento que se produce durante la ejecución de un programa e interrumpe el flujo normal de las instrucciones del mismo. (Oracle)

La gestión de excepciones en Java se lleva a cabo mediante cinco palabras clave: *try, catch, throw, throws* y *finally*. A continuación se describe brevemente su funcionamiento: Las sentencias del programa que se quieran monitorear, se incluyen en un bloque *try*. Si una excepción ocurre dentro del bloque *try*, ésta es lanzada. El código puede capturar esta excepción, utilizando *catch*, y gestionarla de forma racional. Las excepciones generadas por el sistema son automáticamente enviadas por el intérprete Java. Para enviar manualmente una excepción se utiliza la palabra clave *throw*. Se debe especificar mediante la cláusula *throws* cualquier excepción que se envíe desde un método al método exterior que lo llamó. Se debe poner cualquier código que el programador desee que se ejecute siempre, después de que un bloque *try* se complete, en el bloque de la sentencia *finally*. (Ceballos, 2010).

En Groovy, el manejo de excepciones es exactamente el mismo que en Java y sigue la misma lógica. Puede especificar una completa secuencia de bloques *try-catch-finally*, o simplemente *try-catch*, o simplemente *try-finally*. Tenga en cuenta que a diferencia de otras estructuras de control, se requieren llaves alrededor de los cuerpos de bloque si contienen o no más de una sentencia. La única diferencia entre Java y Groovy en términos de excepciones es que las declaraciones de excepciones en la firma del método son opcionales, incluso para las excepciones comprobadas. (König & Glover, 2007, pág. 171)

Groovy no exige controlar las excepciones que no se desean manejar o que son inapropiados en el nivel actual de código. Cualquier excepción no manejada se pasa automáticamente a un nivel superior (Subramaniam, 2013, pág. 17) .

En general todas las excepciones en Groovy son *no comprobadas*, o más bien su manejo es opcional.

J. PALABRAS RESERVADAS

Java define una serie de palabras propias del lenguaje para la identificación de operaciones, métodos, clases etc., con la finalidad de que el compilador pueda entender los procesos que se están desarrollando. Estas palabras no pueden ser usadas por el desarrollador para nombrar a métodos, variables o clases, pues como se mencionó, cada una tiene un objetivo dentro del lenguaje.

En caso de definir a un identificador con alguna de las palabras reservadas el compilador advierte ese error para que el programador haga algo al respecto.

La Tabla 1 muestra las palabras reservadas de Java:

Tabla 1. Palabras reservadas Java (Schildt, 2009)

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

El lenguaje Groovy hace la implementación de nuevas palabras clave, las cuales ayudan a que maneje ciertas características de una manera más sencilla, haciendo el código más fácil de escribir, leer y entender.

def e *in* se encuentran entre las nuevas palabras clave de Groovy. *def* define los métodos, propiedades y variables locales. *in* se utiliza en los bucles para especificar el rango de un bucle, como en *for(i in 1..10).*

El uso de estas palabras clave como nombres de variable o nombres de método puede dar lugar a problemas, especialmente cuando se utiliza código Java existente como código Groovy.

Tampoco es una buena idea definir una variable llamada *it.* Aunque Groovy no se quejará, si se utiliza un campo con ese nombre dentro de un cierre, el nombre se refiere al parámetro de cierre y no un campo dentro de la clase. (Subramaniam, 2013)

Otra palabra nueva que trae Groovy es *as*, la cual permite cambiar el tipo de objeto, el ejemplo práctico se vio en el tema Constructores.

K. ESTRUCTURAS DE CONTROL

Groovy soporta casi las mismas estructuras de control lógicas que Java, haciendo excepción respecto a los parámetros aceptados o implementaciones de las mismas. Dentro de las condicionales evalúan una prueba de Booleanos y toman una decisión en base a si el resultado era verdadero o falso. Ninguna de estas estructuras debería ser una experiencia completamente nueva para cualquier desarrollador de Java, pero por supuesto Groovy añade algunos toques propios.

En Groovy la expresión de una prueba booleana puede ser de cualquier tipo (no void). Se puede aplicar a cualquier objeto. Groovy decide si se debe considerar la expresión como verdadera o falsa mediante la aplicación de las reglas que se muestran en la Figura 8 (König & Glover, 2007):

Tipo	Criterio de evaluación requerido a true
Boolean	Correspondiente valor booleano es true
Matcher	El comparador cuenta con una coincidencia
Collection	La colección no está vacía
Map	El mapa no está vacío
String, Gstring	La cadena no está vacía
Number, Character	El valor es distinto de cero
Ninguna de las anteriores	La referencia del objeto no es null

Figura 8. Secuencia de reglas utilizadas para evaluar un prueba booleana.

if/ if-else

Actúa exactamente de la misma manera en Groovy a como lo hace en Java.

Al igual que en Java, la expresión de prueba booleana debe ir entre paréntesis. El bloque condicional normalmente se encierra entre llaves. Estos apoyos son opcionales si el bloque se compone de una sola sentencia.

La única diferencia está en cómo interpreta Groovy estas condiciones. Groovy puede promover una serie de condiciones no booleanas a true o false. Así, por ejemplo, un número distinto de cero es siempre *true*. (Dearle, 2010)

Operador ternario y Elvis (?:)

En Java, el operador ternario se puede decir que es un if-else abreviado, como lo maneja el lenguaje de programación C. Groovy es compatible con la manera estándar de Java al manejarlo, pero tiene un operador similar llamado Elvis, el cual destaca más practicidad al programar, su estructura es *(a ?: b)* siendo el equivalente ternario *(a ? a : b)*. (Dearle, 2010). Los símbolos "**?:**" no deben ir con un espacio entre ellos, de lo contario marcará error.

Elvis trabaja mediante el mantenimiento de una variable local oculta, la cual almacena el resultado inicial. Si ese resultado se da de acuerdo a las reglas de la Figura 8, entonces ese valor se devuelve, si no, se utiliza el valor alternativo. (Dearle, 2010)

Sentencia switch

En Java, al igual que en el lenguaje C, la sentencia *switch* permite a determinada variable ser probada por una lista de condiciones manejadas en los *case*.

Groovy, por el contrario, es más amigable con los tipos de datos dentro de los *case*. Los elementos *case* no son del mismo tipo que recibe el *switch*, puede manejar al mismo tiempo enteros, listas, cadenas, rangos, etc. (Dearle, 2010)

Bucle while

Este bucle, respecto a la sintaxis y estructura lógica, es igual en ambos lenguajes.

Cabe destacar que Groovy no acepta el bucle do{} while(), desconoce totalmente qué signifique esa sentencia si alguien la intenta usar.

Bucle for

Cuando se creó Groovy, y durante algún tiempo después, él no apoyó el estándar Java para el bucle:

for (int i = 0; i < 5; i++) { ... }

En la versión 1.6, sin embargo, se decidió que era más importante que admita construcciones Java, que tratar de mantener un lenguaje libre de esa sintaxis poco incómoda que Java heredó de sus predecesores (Como el lenguaje C).

De igual manera soporta el *for-each* de Java. (Kousen, 2014)

El for-each es utilizado para iterar sobre los elementos de colecciones que sean arrays, ArrayList, HashMap,…

Sin embargo, ninguno de esos bucles es la forma más común de iterar en Groovy. En lugar de escribir un bucle explícito, como en los ejemplos anteriores, Groovy prefiere una aplicación más directa del diseño de patrón iterador. Groovy añade el método *each*, que toma un cierre como argumento para colecciones.

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

El método *each* es la construcción del bucle más común en Groovy.

Otro bucle manejado por Groovy es el for-in, se puede utilizar la expresión "*in*" para repetir cualquier tipo de colección. (Dearle, 2010)

L. LA JAVA VIRTUAL MACHINE (JVM)

Uno de los puntos interesantes que tiene Groovy es que hace uso de la máquina virtual de Java para ejecutar los códigos. Como bien es sabido, Java es un lenguaje compilado e interpretado, mientras que Groovy es interpretado, aunque también es posible utilizar el compilador para él.

La especificación JVM provee definiciones concretas para la implementación de lo siguiente: un conjunto de instrucciones (equivalente a la de una unidad central de procesamiento [CPU]), un conjunto de registros, el formato de archivo de clase, una pila de ejecución, pila garbage-collector, un área de memoria, mecanismo de informes de error fatal, y soporte de temporización de alta precisión.

El formato del código de la máquina JVM consiste en *bytecodes* compactos y eficientes. Programas representados por *bytecodes* JVM deben mantener una disciplina de tipo adecuado. La mayor parte de la verificación de tipos se realiza en tiempo de compilación.

Cualquier tecnología compatible intérprete de Java debe ser capaz de ejecutar cualquier programa con archivos de clases que cumplen con el formato de archivo de clase especificado en la Especificación de la Máquina Virtual de Java. (Oracle and/or its affiliates, 2010)

La filosofía de la máquina virtual es la siguiente: el código fuente se compila, detectando los errores sintácticos, y se genera una especie de ejecutable, con un código máquina dirigido a una

máquina imaginaria, con una CPU imaginaria. A esta especie de código máquina se le denomina *código intermedio*, o a veces también *lenguaje intermedio*, *p-code*, o *byte-code*.

Como esa máquina imaginaria no existe, para poder ejecutar ese ejecutable, se construye un intérprete. Este intérprete es capaz de leer cada una de las instrucciones de código máquina imaginario y ejecutarlas en la plataforma real. A este intérprete se le denomina el *intérprete de la máquina virtual*. (Vic, 2013)

Actualmente, como es bien sabido, la JVM ya no es sólo para Java, existen otros lenguajes que se pueden compilar en la máquina virtual y se encuentran disponibles para su uso. Estos lenguajes compilan a bytecode en archivos clase, los cuales pueden ser ejecutados por la JVM.

Groovy es un lenguaje de programación orientado a objetos que se ejecuta en la JVM. También conserva plena interoperabilidad con el lenguaje Java.

Se menciona lo siguiente respecto a la ejecución de clases o scripts Groovy, concluyendo el autor de dicho escrito que éste es un lenguaje compilado:

"*En Java se compila con javac y ejecuta los bytecodes resultantes con java. En Groovy puede compilar con groovyc y ejecutar con groovy, pero en realidad no tiene que compilar primero. El comando groovy puede funcionar con un argumento de código fuente, y compilará primero y luego lo ejecutará. Groovy es un lenguaje compilado, pero usted no tiene que separar los pasos, aunque la mayoría de la gente lo hace. Cuando se utiliza un IDE, por ejemplo, cada vez que guarde un script o clase Groovy, se compila*". (Kousen, 2014)

Conclusiones

Se pudo corroborar que los dos son lenguajes que cumplen satisfactoriamente con los elementos necesarios para ser considerados Orientados a Objetos, recordando que sólo son cuatros los elementos primarios a considerar, pero de la misma forma cumplen con los secundarios, tales elementos forman parte del Modelo Orientado a Objetos propuesto por Graddy Booch.

Ambos lenguajes tienen un gran parecido sintáctico, Groovy hace el uso de scripts para programas pequeños, mientras que para un sistema de mayor tamaño es recomendable el uso de clases al igual que lo hace Java, simplemente para tener un mejor desarrollo y este permita ser modificado o mejorado de una manera simple.

Una de las características más notorias es la simplicidad que maneja en código el nuevo lenguaje, es más fácil programar en Groovy, viniendo de trabajar con Java resulta más cómodo codificar, recordando que el 99 % de código Java lo reconoce Groovy, pero además éste último sólo utiliza lo necesario para trabajar. Lo que Java podría trabajar con 5 líneas en el simple "hola mundo", Groovy lo realiza en sólo una utilizando scripts, o con las mismas 5 líneas que usa Java pero omitiendo parte de esa sintaxis forzosa que maneja, punto que ha dado pie al desarrollo de este trabajo.

De manera sencilla se pueden destacar lo siguientes puntos que maneja Groovy en cuanto a sintaxis comparándolo con Java:

- Las clases, variables y métodos son public por defecto.
- Groovy maneja herencia múltiple respetando jerarquía de clases.
- Se omiten los puntos y comas al finalizar una sentencia.
- Los paréntesis son opcionales.
- Groovy no requiere definir un tipo de dato a las variables, puesto que uno de sus fuertes es que es un lenguaje dinámico.
- Algunas sentencias las acorta.
- Hace importaciones de bibliotecas y no requiere que se expresen explícitamente.
- La palabra return es opcional.
- Para la implementación de Strings ofrece nuevas formas de hacerlo además de trabajar con la manera tradicional de Java.
- Reduce código al crear getters y setters de forma implícita a variables public por default.
- Respecto a estructuras de control maneja las mismas que Java excepto por el do-while que no lo reconoce, pero todas las demás las trabaja con la misma lógica, e incluso hace algunas implementaciones de las mismas.
- Trabaja una característica llamada Closure, la cual ofrece la misma funcionalidad que un método, pero la diferencia es que los Closures son anónimos, es decir no están ligados a un objeto.
- El manejo del bloque try-catch y todo lo que implica el manejo de excepciones lo hace opcional.
- Groovy es Orientado a Objetos Puro, aparentemente trabaja con variables primitivas como Java, pero bajo las sábanas recurre a los wrappers para convertirlos a Objetos.

Tanto Java como Groovy ejecutan sus procesos sobre una Máquina Virtual, ésta trabaja con un compilador y un intérprete, el primer lenguaje es compilado e interpretado, mientras que el segundo es interpretado, pero también se le permite compilar.

Cabe reconocer que Java sigue sobresaliendo y ha logrado ser uno de los mejores lenguajes debido a que ofrece mayor número de herramientas para cumplir con las necesidades de los programadores, permitiendo a éstos satisfacer los requerimientos de los usuarios. A Groovy le falta abordar algunos puntos que permitan su empleo de manera más satisfactoria, al menos para trabajar a la altura de su homólogo, pero no olvidar que poco a poco éste mismo ha mejorado y además se han ido creando herramientas que lo ayudan a cubrir con la demanda. Java, de la misma manera va mejorando aspectos, mostrando que no es un lenguaje fácil de remplazar o igualar, se le ha visto tener fallas pero esto le ha servido para trabajar en ello y mejorar.

Sería interesante más adelante retomar el tema aquí tratado y observar cómo han ido avanzando ambos lenguajes, tener muy presente que el desarrollo es un tema de actualidad con gran porvenir, habrá nuevos lenguajes, otros desaparecerán tal vez y muchos más irán mejorando sus características como ya se ha visto.

> *"El éxito de Groovy se debe principalmente a su familiaridad con Java. Los desarrolladores quieren hacer las cosas, desean dominar el lenguaje rápidamente, quieren características de potencia. Groovy es un lenguaje más productivo que Java, pero aún con una sintaxis de Java similar que ya es bien conocida. Y encima de eso, Groovy simplifica las tareas cotidianas que solían ser complejas para desarrollar."*
> (White & Maple, 2014)

Bibliografía

Booch, G. (1991). *Object Oriented Design with Aplications.* Redwood City, California 94065: The Benjamin/cummings Publishing Company, Inc.

Castro Souto, L. (2001). *Programación Orientada a Objetos.* Retrieved Enero 15, 2015, from Programación Orientada a Objetos: http://quegrande.org/apuntes/EI/OPT/POO/teoria/00-01/apuntes_completos.pdf

Ceballos, F. J. (2010). *Java 2, Curso de Programación.* México: RA-MA.

Cisneros, O. (2010, Agosto). *Tópicos Selectos de Programación*. Retrieved Enero 17, 2015, from Programación concurrente multihilo: http://topicos-selectosdeprogramacion-itiz.blogspot.mx/p/unidad-3-programacion-concurrente.html

Dearle, F. (2010). *Groovy for Domain-Specific Languages.* Birmingham: Packt Publishing.

Di Serio, A. (2011, Marzo 18). *Programación Orientada a Objetos.* Retrieved Octubre 28, 2014, from LDC: Noticias: http://ldc.usb.ve/~adiserio/Telematica/HerramientasProgr/ProgramacionOONotes.pdf

EduSanz. (2010). *Añadiendo o sobreescribiendo métodos*. Retrieved Febrero 15, 2015, from Groovy & Grails: http://beginninggroovyandgrails.blogspot.mx/p/groovy.html

Egiluz, R. (2011, Septiembre 27). Groovy hacia un JVM políglota.

emalvino. (2013, Mayo 21). *Hexata Architecture Team*. Retrieved Febrero 3, 2015, from Hexata Architecture Team: http://hat.hexacta.com/agregando-funcionalidad-a-objetos-de-dominio-en-grails/

ESCET. (2009, Agosto 18). *Escuela Superior de Ciencias Experimentales y Tecnología.* Retrieved Octubre 23, 2014, from Introducción a POO y Java: http://www.escet.urjc.es/~emartin/docencia0809/poo/1.-Introduccion-4h.pdf

Groovy. (n.d.). *Class File*. Retrieved Diciembre 2014, from Method Summary: http://groovy.codehaus.org/groovy-jdk/java/io/File.html

Hdeleon. (2014, Mayo 12). *5.- POLIMORFISMO – CURSO DE PROGRAMACIÓN ORIENTADA A OBJETOS.* Retrieved Diciembre 4, 2014, from Curso Programación Orientada a Objetos POO: http://hdeleon.net/5-polimorfismo-curso-de-programacion-orientada-objetos-en-10-minutos-5/

Horna, M. (2010). *Resumen de objetivos para el SCJP 6.0.*

Joyanes Aguilar, L. (1996). *Programación Orientada a Objetos.* España: McGraw-Hill.

Judd, C. M., & Faisal Nusairat, J. (2008). *Beginning Groovy and Grails.* United States of America: Apress.

Klein, H. (2009, Diciembre 22). *Groovy Goodness: Implementing MetaClass Methods with Same Name but Different Arguments*. Retrieved Febrero 15, 2015, from HaKi: http://mrhaki.blogspot.mx/2009/12/groovy-goodness-implementing-metaclass.html

König, D., & Glover, A. (2007). *Groovy in Action.* United States of America: Manning.

Kousen, K. A. (2014). *Making Java Groovy.* United States of America: Manning.

Marin, F. (2012). *Herencia y Polimorfismo JAVA.* Retrieved Septiembre 25, 2014, from issuu: http://issuu.com/felixmarin/docs/herencia_y_polimorfismo_java

Miedes, E. (2014). *Serialización de objetos en Java .* Retrieved Diciembre 3, 2014, from javaHispano: http://www.javahispano.org/storage/contenidos/serializacion.pdf

Oracle. (2014, Septiembre 16). *Naming Convertions*. Retrieved from Oracle Technology network-Java: http://www.oracle.com/technetwork/java/codeconventions-135099.html

Oracle. (2014). *Overriding and Hiding Methods*. Retrieved Diciembre 10, 2014, from Java Documentation: https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Oracle and/or its affiliates. (2010, Junio). *Java Programming Language, Java SE 6.* United States of America.

Oracle. (n.d.). *What Is an Exception?* Retrieved Septiembre 27, 2014, from Java Documentation: http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Ortiz, A. (2014). *Persistencia en Java*. Retrieved Diciembre 3, 2014, from Estructura de datos: http://webcem01.cem.itesm.mx:8005/apps/s201411/tc1018/notas_persistencia/

Rodríguez Echeverría, R., & Prieto Ramos, Á. (2004). *Programación Orientada a Objetos.*

Schildt, H. (2009). *Java, Manual de Referencia (7a ed.).* México: McGraw-Hill.

Subramaniam, V. (2013). *Programming Groovy 2.* United States of America: The Pragmatic Bookshelf.

Tesler, A. (1981). *The Smalltalk Environment.*

TutorialesNET. (2014, Abril 16). *Java - 33: Modularidad.* Retrieved Enero 9, 2015, from TutorialesNet: http://tutorialesnet.net/cursos/curso-de-java-7

UNAM. (2013, Febrero 12). *Análisis Orientado a Objetos.* Retrieved Octubre 28, 2014, from Repositorio digital de la Facultad de Ingeniería - UNAM: http://www.ptolomeo.unam.mx:8080/xmlui/bitstream/handle/132.248.52.100/175/A6%20Cap%C3%ADtulo%203.pdf?sequence=6

Vic. (2013, Febrero 28). *La tecla de ESCAPE.* Retrieved Octubre 15, 2014, from La tecla de ESCAPE: http://latecladeescape.com/t/Compiladores,+int%C3%A9rpretes+y+m%C3%A1quinas+virtuales

White, O., & Maple, S. (2014). *10 Kickass Technologies Modern Developers Love.* Retrieved Octubre 30, 2014, from RebelLabs: http://pages.zeroturnaround.com/Kickass-Technologies.html?utm_source=10%20Kickass%20Technologies&utm_medium=reportDL&utm_campaign=kick-ass-tech&utm_rebellabsid=89