UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO

CENTRO UNIVERSITARIO UAEM TEXCOCO

"COMPARATIVA SINTÁCTICA ENTRE LOS LENGUAJES DE PROGRAMACIÓN JAVA Y GROOVY"

T E S I N A

QUE PARA OBTENER EL TÍTULO DE

INGENIERA EN COMPUTACIÓN

PRESENTA

Rosa Isela Zarco Maldonado

DIRECTOR

Dr. en Ed. Joel Ayala de la Vega

REVISORES

M. en I.S.C. Irene Aguilar Juárez

Dr. en C. Oziel Lugo Espinosa

TEXCOCO, EDO DE MÉXICO, 20/03/2015

Texcoco, México, a 5 de Febrero del 2015.

M. EN C. ED. VIRIDIANA BANDA ARZATE
SUBDIRECTORA ACADEMICA DEL
CENTRO UNIVERSITARIO UAEM
TEXCOCO.

COPIA

PRESENTE:

AT'N Lic. en I. A. Cinthya Teresita Islas Rodríguez

RESPONSABLE DEL DEPARTAMENTO DE TITULACIÓN

Con base a las revisiones efectuadas al trabajo escrito titulado "Comparativa Sintáctica entre los Lenguajes de Programación Java y Groovy" que para obtener el título de **Ingeniera en Computación** presenta la sustentante **C. Rosa Isela Zarco Maldonado** con número de cuenta **0823710** respectivamente, se concluye que cumple con los requisitos teórico - metodológicos por lo que se le otorga voto aprobatorio para su sustentación, pudiendo continuar con la etapa de digitalización del trabajo escrito

ATENTAMENTE

M. en I. S. C. Irene Aguilar Juárez

Dr. en C. Oziel Lugo Espinosa

Dr en Ed. Joel Ayala de la Vega

c.p.p. Sustentante: . Rosa Isela Zarco Maldonado
c.p.p. Director.- Dr en Ed. Joel Ayala de la Vega
c.p.p. Titulación.- Lic. en I. A. Cinthya Teresita Islas Rodríguez

TITULACION
RECIBIDO
Por Cinthya
Texcoco, Méx., a 05 de 02 del 2015.

Dedicatoria

La presente tesina la dedico en especial a mi hermosa madre, Rosa Maldonado, mujer que siempre ha estado conmigo, en mis triunfos y fracasos. Siempre me ha dicho que yo puedo lograr lo que deseo y me ha dado ánimos para que continúe en el camino por más difícil que sea la situación. Mi madre, mujer que ha sido el mejor ejemplo de vida para mí, demostrándome que no hay imposibles y que la vida siempre trae situaciones difíciles, pero que con esfuerzo y dedicación, se puede salir adelante. Este logro es más tuyo que mío.

A ti dedico todos mis triunfos, TE AMO Mamá.

Agradecimientos

A mi madre Rosa Maldonado por darme la vida, le agradezco toda su paciencia y confianza, creyó en mí, en mis sueños, me brindó su apoyo y de todo lo necesario de manera incondicional para que yo lograra alcanzar mis metas. Gracias a tus esfuerzos, sacrificios y demás, puedo estar aquí, cerrando e iniciando otra etapa de mi vida, no olvido todo tu amor y todo lo que me has apoyado, te admiro de verdad, todo lo que has sacrificado por mí no ha sido en vano. Gracias por enseñarme a valorar, gracias por tus palabras, gracias por ser la mejor madre, te amo con todo mi corazón.

Agradezco a mis hermanos Mary y Paco, con quienes he compartido mucho, a pesar de los problemas que en ocasiones llegamos a tener no dejo de quererlos, son mis hermanos y pase lo que pase siempre estaré con ellos, los quiero y sé que

cuento con su apoyo como hasta ahora. De igual forma agradezco a mis sobrinitos, Leo, Esme y Edy, por enseñarme que así como ellos siendo niños tienen sueños y ríen día a día, debemos seguir toda la vida, riendo y soñando como cuando se es niño.

A toda mi demás hermosa familia mil gracias, siempre he sabido que cuento con ellos y me lo han demostrado, agradezco que siempre estén al pendiente de mí, se han preocupado porque yo salga adelante. En especial agradezco a mi prima Cottí, por el interés, tiempo y paciencia dedicados para que terminara de dar este paso tan importante en mi vida, es una persona a la que admiro por sus logros y la dedicación a su trabajo.

A todos mis amigos con quienes he compartido muchos momentos alegres, gracias por su compañía, su amistad y porque siempre me dan palabras de aliento cuando yo dejo de creer en mí, en mi capacidad.

Gracias a mi director de Tesina, el Dr. Joel Ayala por estar al pendiente de mí en este proceso, gracias por su tiempo dedicado. También agradezco a mis asesores la Maestra Irene Aguilar y el Dr. Oziel Lugo por su tiempo. Gracias a ellos y demás profesores que han sido pilar importante en mi formación académica.

Agradezco a todas aquellas personas que con sus críticas hicieron aferrarme a mis metas, he sabido demostrarles que puedo lograr lo que me propongo, tal vez me cueste muchos sacrificios, desvelos, pero no abandono mis objetivos.

Gracias a todos.

ÍNDICE

INTRODUCCIÓN

Los lenguajes de programación han estado presentes en la vida de las personas desde hace ya algunas décadas atrás, ayudando a dar paso a nuevas herramientas para la vida diaria.

Con el paso de los años y principalmente en base a las necesidades que conlleva la vida cotidiana del ser humano de hacer todo de una manera más fácil y eficiente, ha sido poco satisfactorio lo que se tenía para concebir esas implementaciones, llegando a la necesidad de crear, implementar o mejorar diversos lenguajes de programación.

Uno de los lenguajes que lleva ya varios años de vida y que ha sabido permanecer como uno de los más importantes gracias a sus diversas características que permiten la creación de aplicaciones, es el lenguaje Java.

Java es desarrollado por Sun Mycrosystems en 1991, la necesidad de crearlo fue debido a la poca satisfacción que se tenía con los lenguajes de esa época, por lo que James Gosling, Mike Sheridan y Patrick Naughton entre otros programadores, comenzaron a desarrollarlo, su propósito fue crear un lenguaje para unificar equipos electrónicos de consumo doméstico (lavadoras, tv, etc.). En primer momento fue nombrado como Oak, pero para el año 1995 se le cambió el nombre a Java como actualmente es conocido.

En la actualidad Java es un lenguaje que permite el desarrollo para aplicaciones móviles, de escritorio, corporativas y de igual manera para el entorno web.

Originalmente de Sun Mycrosystems pero a partir del año 2009 ha sido propiedad de Oracle. Java es un lenguaje de Programación Orientado a Objetos, trabaja bajo la JVM (Java Virtual Machine) y es un lenguaje que toma cada vez más importancia.

Por otro lado se tiene a Groovy, creado por James Strachan y Bob McWhirter en el año 2003, basado en los lenguajes Smalltalk, Python y Ruby, aunque también se dice que está basado en Perl, tomando diversas características de dichos

lenguajes como por ejemplo lo dinámico, sólo por mencionar una. Conserva una sintaxis familiar a Java, haciendo de esta manera que los programadores que vienen de trabajar con Java les sea más fácil adaptarse a este lenguaje, de igual manera por su simplicidad de código basado en la utilización de sólo lo necesario, hace que resulte sencillo su aprendizaje.

Groovy, de igual forma que Java, se ejecuta sobre la Máquina Virtual, es un Lenguaje Orientado a Objetos, pero con la diferencia de que el primero es un Lenguaje Orientado a Objetos Puro, al referir que es puro quiere decir que para éste todo es un objeto.

Java es de tipado estático en la declaración de variables primitivas, Groovy es un leguaje dinámico. Groovy es visto como Java minimalista, esto debido a que reduce el código sólo a las líneas necesarias para codificar determinada aplicación.

La intención de éste lenguaje respecto a Java no es sustituirlo, ya que toda su funcionalidad se encuentra disponible en Groovy, sino más bien, es saber cuándo es conveniente utilizar uno respecto del otro, lo anterior con la intención de facilitar a los programadores la implementación de sus proyectos.

CAPÍTULO I. Programación Orientada a Objetos

Introducción

En este capítulo se hace la definición de algunos conceptos básicos para entender lo que es la programación, se hablará sobre la diferencia entre clases y objetos, siendo que algunas veces suele causar un poco de confusión y entender ambos como lo mismo, se conocerán estas y otras definiciones más para dar pie a tratar y entender un poco sobre ambos lenguajes.

Además se explicará qué es y cuáles son los elementos básicos del paradigma Orientado a Objetos, aplicado a los Lenguajes de Programación que serán tratados en el presente trabajo, refiriéndose específicamente a Java y Groovy.

¿Qué es la Programación Orientada a Objetos?

La programación orientada a objetos (POO) es un modelo de programación que utiliza objetos, ligados mediante mensajes para la solución de problemas. La idea central es simple: organizar los programas a imagen y semejanza de la organización de los objetos en el mundo real. (Ceballos, 2010, pág. 33)

Es una forma especial de programar, más cercana a cómo expresaríamos las cosas en la vida real que otros tipos de programación.

Con la POO tenemos que aprender a pensar las cosas de una manera distinta, para escribir nuestros programas en términos de objetos, propiedades, métodos, entre otras cosas. (Álvarez, 2014)

La tecnología orientada a objetos se apoya en los sólidos fundamentos de la ingeniería, cuyos elementos reciben el nombre global de modelo de objetos. El modelo de objetos abarca los principios de abstracción, encapsulación, modularidad, herencia, entre otros conceptos más.

Definición de POO

> *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.* (Booch, 1991)

Hay tres partes importantes en esta definición:

1. Utiliza objetos.
2. Cada objeto es una instancia de alguna clase.
3. Las clases están relacionadas con otras clases por medio de relaciones de herencia.

Un programa puede parecer orientado a objetos, pero si falta alguno de estos elementos, no lo es. (Booch, 1991)

Clases y objetos

A la hora de tratar un problema, se puede descomponer en subgrupos de partes relacionadas. Estos subgrupos pueden traducirse en unidades auto contenidas llamadas *objetos*. Antes de la creación de un objeto, se debe definir en primer lugar su formato general, su plantilla, que recibe el nombre de *clase*. (Rodríguez Echeverría & Prieto Ramos, 2004)

> *Clase: Viene a representar la definición de un módulo de programa, y a su vez define métodos y atributos comunes.*
>
> *Objeto: Cada uno de los objetos es una instancia de la clase, esta instancia contiene atributos y métodos con valores específicos de sus datos.*

En la siguiente figura se muestran los conceptos anteriores:

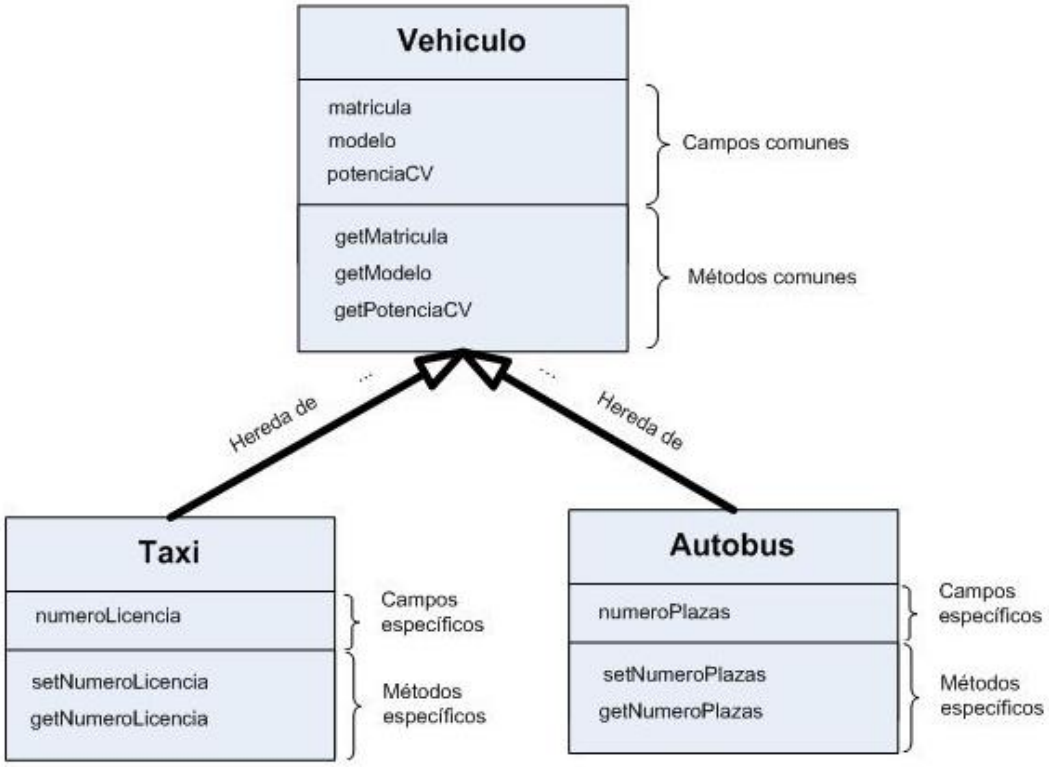

Figura 1. Clases y Objetos

Métodos

Cada uno de los objetos es una entidad que tiene propiedades particulares, atributos, y formas de operar sobre ellos, los métodos.

Un método se escribe en una clase de objetos 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)

Elementos del Modelo Orientado a Objetos

Grady Booch en su libro Análisis y Diseño orientado a objetos menciona lo siguiente al respecto:

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 de 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.

En lo que resta del capítulo se definirán estos elementos del paradigma para ambos lenguajes que se pretenden tratar en el presente trabajo.

Abstracción

Es la capacidad de concentrar las propiedades y comportamientos necesarios para la correcta representación del objeto dentro del sistema, otra definición es que consiste en el aislamiento conceptual de una propiedad de un objeto.

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)

Se centra en las características esenciales de algún objeto; en relación a la perspectiva del observador.

Encapsulamiento

Es la propiedad que permite asegurar que el contenido de la información de un objeto está oculto al mundo exterior, es decir, un objeto *A* no conoce lo que hace un objeto *B*, y viceversa. La encapsulación también es conocida como ocultación de la información.

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 sólo 3 modificadores de acceso. Las clases únicamente pueden ser public o default. (Horna, 2010)

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*. (Judd & Faisal Nusairat, 2008, pág. 23)

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

```java
public class AccesoJ {
    public static void main(String[] args){
        System.out.println("Hola");
        ClaseBJ b = new ClaseBJ();
        b.imprimir();
    }
}

class ClaseBJ{
    public void imprimir(){
        System.out.println("El acceso de la clase principal debe " +
                "ser public y las demás default");
    }
}
```

Figura 2. Clase Java

Transcribiendo el código anterior en lenguaje Groovy ambas clases son por default *public*, no importando cuál tenga el método *main*. Ver la siguiente Figura:

```groovy
class AccesoG {
    static main(args){
        println "Hola"
        def b = new ClaseB()
        b.imprimir()
    }
}

class ClaseB{
    def imprimir(){
        println """Ambas clases son publicas por default,
no importando que la clase AccesoG tenga el main."""
    }
}
```

Figura 3. Clase Groovy

Modularidad

Los módulos sirven como contenedores físicos en los que se declaran las clases y objetos del diseño lógico realizado.

Una definición es la siguiente (Booch, 1991):

> Modularidad es la propiedad de un sistema que ha sido descompuesto a una cantidad de módulos altamente cohesivos y débilmente acoplados.

Los conceptos de acoplamiento y cohesión se definen como sigue:

- Acoplamiento: El acoplamiento indica el nivel de dependencia entre las clases de un sistema, es decir, el grado en que una clase puede funcionar sin recurrir a otras.
- Cohesión: se refiere al grado en que una clase tiene un único y bien definido papel o responsabilidad.

Por lo que "*Hay que hacer lo posible por construir módulos cohesivos (agrupando abstracciones que guarden cierta relación lógica) y débilmente acoplados (minimizando las dependencias entre módulos).*"

Así los principios de abstracción, encapsulación y modularidad son sinérgicos[1]. 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)

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.

[1] Perteneciente o relativo a la sinergia. Sinergia: Acción de dos o más causas cuyo efecto es superior a la suma de los efectos individuales.

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.

Para el presente trabajo, la codificación para ambos lenguajes se ha realizado sobre el IDE Eclipse, dicho IDE y otros más como NetBeans, permiten que las clases puedan ser agrupadas dentro de paquetes lo cual da paso a la modularidad.

La siguiente figura muestra la estructura de paquetes en el IDE Eclipse.

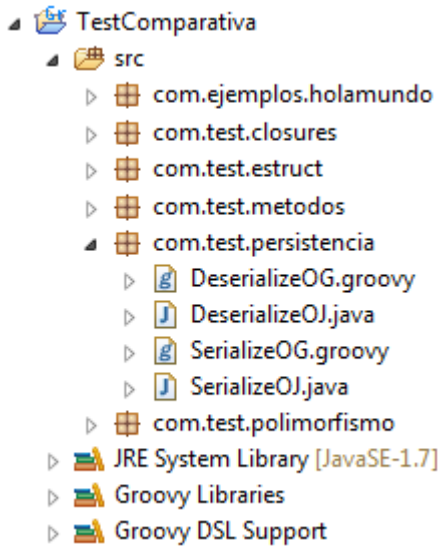

Figura 4. Modularidad

Se pueden observar una serie de paquetes cuyo nombre delimita a un grupo de clases pertenecientes a determinado tema en particular.

Jerarquía

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)

Se crean nuevos objetos a partir de los existentes de forma que heredan las propiedades y comportamientos de sus ancestros. Existen dos clases de herencia: simple y múltiple.

La herencia impone una relación jerárquica entre clases en la cual una clase hija hereda de su clase padre. Si una clase sólo puede recibir características de otra clase base, la herencia se denomina herencia simple.

Si una clase recibe propiedades de más de una clase base, la herencia se denomina herencia múltiple. (Di Serio, 2011)

La herencia es una de las piedras angulares de la programación orientada a objetos, ya que permite la creación de clasificaciones jerárquicas. Mediante la herencia se puede crear una clase general que define rasgos generales para un conjunto de términos relacionados. Esta clase puede ser heredada por otras clases más específicas, cada una de las cuales añadirá aquellos elementos que la distinguen. 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.

Herencia simple

Cada clase puede tener una superclase (o clase base) lo que se denomina herencia simple, o dicho de otra forma, una clase únicamente puede heredar de otra.

Fundamentos de la herencia

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

A continuación se tienen los códigos para una clase en ambos lenguajes.

```java
package com.test.c1;

class Animal{
    String nombre, alimento;
    int edad;

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

    public void setNobre(String nombre){..
    public String getNombre(){..
    public void setEdad(int edad){..
    public int getEdad(){..
    public void setAlimento(String alimento){..
    public String getAlimento(){..
    public String toString(){
        return "El animal es un " + nombre +", tiene " + edad +" años de edad";
    }
}

class Gato extends Animal{
    public Gato(String nombre, int edad){
        super.setNobre(nombre);
        super.setEdad(edad);
    }
}

public class TestAnimal {
    public static void main(String[] args){
        Gato g = new Gato("Gato", 4);
        System.out.println(g +" y " + g.comer(" atún"));
    }
}
```

Figura 5. Herencia clase Java

A continuación se tiene el código en Groovy.

```groovy
package com.test.c1

class TestAnimalG {
    static main(args){
        def g = ["Gato ", 4] as GatoG
        println "${g} y  ${g.comer('atún')}"
    }
}

class GatoG extends AnimalG{
    GatoG(String nombre, int edad){
        super(nombre, edad)
    }
}

class AnimalG{
    def nombre, alimento
    def edad

    AnimalG(String nombre, int edad){
        this.nombre = nombre
        this.edad = edad
    }
    def comer(alimento){
        "le gusta comer ${alimento}"
    }
    String toString(){
        "El animal es un ${nombre}, tiene ${edad} años de edad"
    }
}
```

Figura 6. Herencia clase Groovy

Como se puede ver en la clase anterior no hay nada diferente comparándolo con el código Java, se tienen los mismos resultados en ambas clases.

En los códigos anteriores se hace uso de la palabra *super*, más adelante se hablará al respecto.

Herencia Múltiple

Debido a que Java no soporta el concepto de herencia múltiple como otros lenguajes Orientados a Objetos, este 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)

El ejemplo es el siguiente:

```
1  package com.test.herencia;
2
3  interface A{
4      public void rodar();
5  }
6
7  interface B{
8      public void correr(int n);
9  }
10
11 public class HerenciaMultJ implements A, B{
12     public static void main(String[] args){
13         HerenciaMultJ t = new HerenciaMultJ();
14         t.brincar();
15         t.rodar();
16         t.correr(3);
17     }
18     public void rodar(){
19         System.out.println("Rodar");
20     }
21     public void brincar(){
22         System.out.println("Brincar");
23     }
24     public void correr(int n){
25         System.out.println("Saltar " + n + " metros");
26     }
27 }
```

Figura 7. Simulación de Herencia Múltiple en Java

La ejecución del código es la siguiente:

```
Brincar
Rodar
Saltar 3 metros
```

Groovy para manejar el concepto 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)

```
1  package com.ejemplos.herenciam
2
3  @Mixin([Ag, Bg])
4  class HerenciaMultG {
5      static main(args){
6          def t = new HerenciaMultG()
7          t.rodar()
8          t.brincar()
9          t.correr(3)
10     }
11     def brincar(){
12         println "Brincar in Groovy"
13     }
14 }
15 class Ag{
16     def rodar(){
17         println "Rodar in Groovy"
18     }
19 }
20 class Bg{
21     def correr(n) {
22         println "Correr in Groovy $n metros"
23     }
24 }
```

Figura 8. Herencia Múltiple en Groovy

La ejecución del programa arroja lo siguiente:

```
Rodar in Groovy
Brincar in Groovy
Correr in Groovy 3 metros
```

Lo interesante al respecto es que Groovy permite manejo de herencia de una manera más clara respecto a que sí hace el manejo de clases, Java hace uso de interfaces pero únicamente estas permiten la declaración de métodos mas no la implementación como lo hace el otro lenguaje.

La palabra clave *super*

Siempre que una subclase necesite referirse a su superclase inmediata, se utilizará la palabra clave *super*.

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. A continuación se examina cada uno de estos usos.

super para llamar a constructores de superclase

Una subclase puede llamar a un método, constructor definido por su superclase utilizando la siguiente forma de *super*:

super (lista de parámetros);

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)

Esta manera fue la utilizada en los ejemplos anteriores (Figura 5 y 6).

Segundo uso de super

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. Ejemplo:

```
Public class vehiculo {
    double velocidad;
    …..
    public void acelerar (double cantidad){
        velocidad += cantidad;
    }
}
Public class coche extends vehiculo {
    double gasolina;
    public void acelerar(double cantidad) {
        super.acelerar(cantidad);
        gasolina *= 0.9;
    }
}
```

Figura 9. Uso de super en métodos

En el ejemplo anterior, la llamada *super.acelerar(cantidad)* llama al método acelerar de la clase vehículo (el cual acelerará la marcha).

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)

Tipos (Tipificación)

Para comenzar a desarrollar este tema, se abordarán un par de definiciones sobre tipos. (Booch, 1991)

> *"Un tipo es una caracterización precisa de las propiedades estructurales o de comportamiento (atributos, métodos) que comparten una serie de entidades."*

Otra definición es la siguiente:

> *"Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no puedan intercambiarse o, como mucho, puedan intercambiarse sólo de formas muy restringidas."*

Los tipos permiten expresar las abstracciones de manera que el lenguaje de programación en el que se implantan puede utilizarse para apoyar las decisiones de diseño. (Booch, 1991)

En este tema se manejan los términos tipo y clase de manera muy similar, pero ¿Cuál es su relación?

Los lenguajes O.O. hacen (por sencillez) que no haya prácticamente ninguna diferencia, no obstante, formalmente la diferenciación que se suele hacer subraya que un tipo especifica una semántica[2] y una estructura y una clase es una implementación concreta de un tipo.

En los L.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)

Ventajas de la tipificación

Las ventajas de ser estrictos con los tipos son:

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

Comprobación de tipos

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:

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

- 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 ejecución en la mayoría de los lenguajes.
- 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 a la manera de trabajar Java y Groovy este aspecto, se tiene la conclusión de 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.

Los Lenguajes Orientados a Objetos tienen la característica de poder ejecutar ligadura tardía.

A partir de lo anterior se abre paso para hablar de un tema de suma importancia dentro del Modelo Orientados a Objetos, el tema referido es el “Polimorfismo”.

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, tema que será tratado más adelante.

Una de las características más sobresalientes que marcan la diferencia del lenguaje Groovy respecto de Java es que el primero maneja tipificación dinámica en cuanto a variables, es decir, no es necesario definir un determinado tipo de

dato a una variable como en Java, es permitido para el que le plazca manejarlo explícitamente, pero este lenguaje brinda la oportunidad de no hacerlo, proponiendo con esto un diseño más flexible y con menos código que su homólogo, de lo anterior se hablará en las siguientes páginas.

Tipado 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)

En Groovy todo es un Objeto.

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 como siempre 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 automático y unboxing 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)

No importa cómo las literales (números, cadenas, etc.) aparecen en el código Groovy, son siempre objetos.

Por lo anterior señalado, Groovy es un Lenguaje Orientado a Objetos Puro.

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 (*sobreescritura*). (Hdeleon, 2014)

Cualquier objeto que pase el test IS-A[3] 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)

Sobreescritura 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

[3] 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.

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 sobreescribir, 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)

Lo anterior se verá en ejemplos posteriores y se hablará al respecto.

```
1  package com.test.polimorfismo;
2
3  public class AutoJ2 extends CarroJ2{
4      public static void main(String[] args){
5          CarroJ2 c= new CarroJ2();
6          CarroJ2 a = new AutoJ2();
7          c.correr();
8          a.correr(200);
9      }
10     @Override
11     public void correr(){
12         System.out.println("El Auto corre");
13     }
14     public void correr(int i){
15         System.out.println("El Auto corre a " + i + " km/h");
16     }
17 }
18
19 class CarroJ2 {
20     public void correr(){
21         System.out.println("El carro corre a gran velocidad");
22     }
23
24     public void correr(int i){
25         System.out.println("Corre a " + i + " km/h");
26     }
27 }
```

Figura 10. Polimorfismo Java

```groovy
package com.test.polimorfismo

class CarroG3 {
    def correr(){
        println "El carro corre a gran velocidad"
    }
}

class AutoG33 extends CarroG3{
    def correr(int i){
        println "El auto corre a $i km/h"
    }
}

AutoG33.metaClass.correr = {->
    println "El Auto corre"
}

def c = new CarroG3()
CarroG3 a = new AutoG33()
c.correr()
a.correr(5)
```

Figura 11. Polimorfismo Groovy

En los ejemplos anteriores se maneja la sobrecarga y sobreescritura de métodos, pero comparando ambos códigos se pueden distinguir ciertas diferencias para manejar tales características, las cuales se señalarán más adelante.

Se puede percatar que para manejar el polimorfismo se usa el test IS-A, puesto que se está haciendo la obtención de características de otra clase utilizando el concepto de herencia. Lo que se quiere decir es, "*Auto es un Carro*".

Se hace el llamado en el método principal de un método de la clase padre y otro de la clase hija, pero en Java la clase padre maneja un método y se sobrecarga, puesto que la clase hija lo ocupa y lo requiere manejar sobrecargado en la clase *CarroJ2*, a comparación de Groovy, que utiliza la propiedad metaClass para sobreescribir, eso se puede ver en las líneas 15 a la 17 de la Figura 11 y posteriormente en la clase AutoG33 se sobrecarga el método.

El siguiente código muestra cómo hace Groovy la sobrecarga de métodos.

```groovy
package com.test.polimorfismo

class AutoG44 {
    def frenar(){
        println "El auto frena"
    }
}

AutoG44.metaClass.correr << {->
    println "El Auto corre"
} << {int i ->
    println "El auto corre a $i km/h"
} << {s, s2 ->
    println "El auto es ${s} con ${s2}"
}

AutoG44 a = new AutoG44()
a.correr(5)
a.correr("gris", "negro")
a.frenar()
```

Figura 12. Sobrecarga de métodos en Groovy

Se puede observar que se tiene una clase AutoG44 con un método, con la propiedad *metaClass* se agrega uno nuevo y allí mismo se sobrecarga con los símbolos “<<”.

El resultado de correr el script anterior es el siguiente:

```
El auto corre a 5 km/h
El auto es gris con negro
El auto frena
```

Concurrencia

Para ciertos tipos de problema, un sistema automatizado puede tener que manejar muchos eventos diferentes simultáneamente. 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 CPUs permiten hilos de control verdaderamente concurrentes, mientras que los sistemas que se ejecutan en una sola CPU sólo pueden conseguir la ilusión de hilos concurrentes de control.

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."*

Una segunda definición es la siguiente:

> *Es la capacidad de un sistema de ejecutar múltiples procesos al mismo tiempo y dado el caso que estos procesos puedan interactuar entre sí.*

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

Extendendiendo de java.lang.Thread

- Se logra extendiendo de la clase java.lang.Thread y
- Sobreescribiendo el método run().

Ejemplo:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Important job running in MyThread");
    }
}
```

Implementando la interface java.lang.Runnable

Ejemplo:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Important job running in MyRunnable");
    }
}
```

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 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 */ }*

Se presentan algunos ejemplos en ambos lenguajes para revisar lo anterior.

En el ejemplo Java se sobreescribe el método run(). Al ejecutar el programa se pasa un nombre a cada hilo y se imprime.

```java
package com.ejemplos.concurrencia;

public class Languages2J extends Thread{
    public void run(){
        for(int n=0; n<=3; n++){
            System.out.println("Thread " + Thread.currentThread().getName()+ " n es: " +n);
        }
    }
    public static void main(String[] args){
        Thread t1 = new Languages2J();
        Thread t2 = new Languages2J();
        t1.setName("Java");
        t2.setName("Groovy");
        t1.start();
        t2.start();
    }
}
```

Figura 13. Thread Java

La salida es la siguiente:

```
Thread Groovy n es: 0
Thread Java n es: 0
Thread Groovy n es: 1
Thread Java n es: 1
Thread Groovy n es: 2
Thread Java n es: 2
Thread Java n es: 3
Thread Groovy n es: 3
```

En el ejemplo de la siguiente figura respecto a Groovy se hace el uso de ciclos anidados, el primero para decir que se va a ejecutar 4 veces y el segundo es para pasarle el nombre al Thread. El Thread hace uso de un closure en el cual se le indica lo que debe hacer.

Ejemplo Groovy

```
1  package com.ejemplos.concurrencia
2
3  class Languages2G{
4      static main(args){
5          4.times{i ->
6              ["Java","Groovy"].each { ln ->
7                  Thread.start {
8                      println "Thread $ln, n es: $i"
9                  }
10             }
11         }
12     }
13 }
```

Figura 14. Thread Groovy

La salida es la siguiente:

```
Thread Groovy, n es: 2
Thread Java, n es: 1
Thread Java, n es: 3
Thread Groovy, n es: 0
Thread Java, n es: 2
Thread Java, n es: 0
Thread Groovy, n es: 1
Thread Groovy, n es: 3
```

Como se observa es muy sencilla la manera de trabajar de Groovy debido al uso de closures.

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)

La definición de persistencia es la siguiente:

> *La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir, el objeto continúa existiendo después de que su creador deja de existir) y/o espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).* (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)

Serialización

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)

Los siguientes códigos de las imágenes muestran cómo se logra la serialización y deserialización en ambos lenguajes respectivamente:

```java
package com.test.persistencia;
import java.util.Date;

public class SerializeOJ implements Serializable{
    private static final long serialVersionUID = 1L;
    public static void main(String[] args){
        new SerializeOJ();
    }
    SerializeOJ(){
        Date date = new Date();
        try{
            FileOutputStream f = new FileOutputStream("C:/Groovy R/persist.txt");
            ObjectOutputStream out = new ObjectOutputStream(f);
            out.writeObject(date);
            out.close();
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

Figura 15. Serialización en Java

```java
package com.test.persistencia;
import java.io.*;
import java.util.Date;

public class DeserializeOJ implements Serializable{
    private static final long serialVersionUID = 1L;
    public static void main(String[] args){
        new DeserializeOJ();
    }
    DeserializeOJ(){
        Date date = null;
        try{
            FileInputStream f = new FileInputStream("C:/Groovy R/persist.txt");
            ObjectInputStream in = new ObjectInputStream(f);
            date = (Date)in.readObject();
            in.close();
        }catch(Exception e){
            e.printStackTrace();
        }
        System.out.println("Deserializando un objecto de persist.txt");
        System.out.println("Date" + date);
    }
}
```

Figura 16. Deserialización en Java

En el código de la Figura 15 se está serializando el objeto *Date*, se escribirá la fecha dentro del archivo *pesist.txt* para posteriormente deserializarse con el código de la Figura 16, obteniendo lo siguiente:

```
Deserializando un objecto de persist.txt
DateThu Dec 04 19:24:49 CST 2014
```

En Groovy se presentan los códigos respectivos:

```
2  import java.io.Serializable;
3
4  class SerializeOG implements Serializable{
5      static main(args){
6          new SerializeOG()
7      }
8
9      SerializeOG(){
10         def date = new Date();
11         def f = new FileOutputStream("C:/Groovy R/persistG.txt")
12         def out = new ObjectOutputStream(f)
13         out.writeObject(date)
14         out.close()
15     }
16 }
```

Figura 17. Serialización en Groovy

```
2  import java.io.Serializable;
3
4  class DeserializeOG implements Serializable{
5      static main(arg){
6          new DeserializeOG();
7      }
8
9      DeserializeOG(){
10         Date date = null
11         def f = new FileInputStream("C:/Groovy R/persistG.txt")
12         def oi = new ObjectInputStream(f)
13         date = (Date)oi.readObject()
14         oi.close()
15
16         println "Deserializando un objecto de persistG.txt"
17         println("Date $date")
18     }
19 }
```

Figura 18. Deserialización en Groovy

Ejecutando el código se tienen las siguientes líneas:

```
Deserializando un objecto de persistG.txt
Date Thu Dec 04 20:04:03 CST 2014
```

En los códigos de las figuras anteriores se hace la misma operación que se realizó para Java, se almacena un objeto *Date* en un archivo que posteriormente es deserializado.

CAPÍTULO II. Sintaxis

Introducción

Las páginas del presente capítulo buscan mostrar y comparar la manera de codificar para ambos lenguajes respecto a temas como clases, métodos, el manejo de excepciones, entre otros temas relevantes que son de gran ayuda a la hora de comenzar a desarrollar.

Se pretende exponer cómo Groovy hace el manejo de la sintaxis más básica respecto a lo que tiene Java para trabajar. En cada tema desarrollado se realizan pruebas de código para comparar ambos lenguajes, además se resaltan las diferencias que hacen más interesante a uno respecto del otro.

En el segundo apartado se retoma un tema que se manejó en el capítulo anterior, referido a los closures, esta característica de Groovy es una de las novedades que lo hacen interesante para trabajar más fácil, en este caso respecto a métodos.

Esos y otros temas son abordados y desarrollados en este capítulo para llevar a cabo la idea principal de este trabajo, la cual es, comparar ambos lenguajes respecto a la sintaxis básica y proporcionar una idea más precisa sobre qué lenguaje usar al momento de desarrollar tomando en cuenta las necesidades de los usuarios.

Clases

Las clases son las que almacenan las propiedades y métodos que contendrá un *objeto*. Un objeto cambiará sus propiedades o las propiedades de otros *objetos* por medio de los *métodos*. (Pérez, 2008)

El tradicional ejemplo

Para comenzar a demostrar qué tan ligero es el código Groovy respecto a Java, se muestra el clásico "*Hola Mundo*" en ambos lenguajes.

Figura 19. Muestra lo que es una clase Java con el clásico Hola Mundo

```
3  public class HolaMundoJ {
4      public static void main(String[] args){
5          System.out.println("¡Hola Mundo en Java!");
6      }
7  }
```

Figura 19.Hola Mundo en Java

Por otro lado se tiene el código Groovy, este puede manejar para realizar sus códigos, lo que son Clases y/o Scripts.

Script

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 estos deben ser convertidos a un archivo binario ejecutable (por ej: ejecutable .exe, entre otros) para poder correrlos. (Alegsa)

```
3  println "¡Hola Mundo en Groovy!"
```

Figura 20. Script Groovy

Mientras que en Java necesita 5 líneas, Groovy sólo requiere 1 línea de código, cabe resaltar que Groovy hace omisión de la mayoría de características que son necesarias para el código Java.

El programa realizado en Groovy es un script (Kousen, 2014, pág. 19) debido a que es un código de pocas líneas puede ser denominado como tal, 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.

A diferencia de Java, no es necesario poner todo el código Groovy en una clase.

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.

En Java, se necesita un método *main* dentro de una clase, y en el interior del método *main* se llama a *System.out.println* para escribir en la consola. En Groovy, todo el programa es una sola línea hablando de Scripts como en el ejemplo anterior, únicamente se manejó el método *println* para enviar a consola el mensaje. En cuanto al manejo de clases se explicará más adelante qué pasa con el método principal dentro de ellas.

Nomenclatura de clases

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 este 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. Trate de mantener sus nombres de clases sencillos y descriptivos. Usar palabras enteras - evitar siglas y abreviaturas (a menos que la abreviatura sea mucho más extendida que el de la forma larga como dirección URL o HTML). (Oracle)

Este mecanismo de nombre es llamado CamelCase[4] (Por ejemplo: NombreClase, CuentaUsuario, Factura).

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:

- 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 debe coincidir con el nombre de archivo. El compilador groovyc

[4] **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*.

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 ejecutada, así que no declarar una clase teniendo usted el mismo nombre que el nombre del archivo fuente. (König & Glover, 2007, págs. 188-189)

Como ya se ha mencionado en un punto anterior, 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 de Java de hacer las cosas de modo que los nombres de archivo de origen corresponden a los nombres de las clases que se implementan como parte de su solicitud. (Dearle, 2010, págs. 34-35)

En el siguiente ejemplo se ve un caso en el que se tiene un Script y una clase dentro del mismo archivo, y se muestra cómo se trata esta cuestión para evitar problemas.

Ver ejemplo siguiente:

```
1  package com.test.c1
2
3  class ScriptAndClass {
4      static main(args){
5          println "Hola desde Clase"
6      }
7  }
8
9  sc = new ScriptAndClass()
10 sc.main()
11 println "Hola desde Script"
```

Figura 21. Nomenclatura de script

Como se observa en el código de la Figura 21, marca el siguiente error de compilación, "*Multiple markers at this line*".

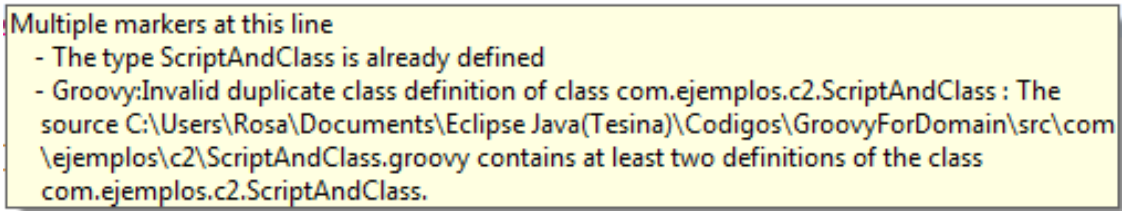

Figura 22. Error marcado en línea 3 del código

Este error quiere decir lo siguiente:

> "Duplicado inválido de definición de clases de la clase "*ScriptAndClass*": La fuente ScriptAndClass.groovy contiene al menos dos definiciones de la clase *ScriptAndClass*.

Una de las clases es generada explícitamente mediante la instrucción de clase, la otra es una clase generada desde el cuerpo script basado en el nombre del archivo. Las soluciones son cambiar el nombre del archivo o cambiar el nombre de la clase.

Al cambiar en este caso el nombre de la clase, permite una compilación y ejecución exitosa. Ver siguiente figura.

ScriptAndClass.groovy

```
1  package com.test.c1
2
3  class scriptAndClass {
4      static main(args){
5          println "Hola desde Clase"
6      }
7  }
8
9  sc = new scriptAndClass()
10 sc.main()
11 println "Hola desde Script"
```

Figura 23. Script

Al ejecutarlo se tiene lo siguiente:

```
Hola desde Clase
Hola desde Script
```

Como se observa, lo único que se hizo fue cambiar a minúscula la primer letra del nombre de la clase y en la instancia que se hace de la misma en el script para poder llamar al método.

El cambio en el nombre de la clase no es necesario que se haga pasando la primera letra a minúscula como se hizo en la Figura 23, puede ser simplemente darle otro nombre. Lo único que se requiere es que sea diferente al nombre del archivo.

Realizado lo anterior, se resalta que se ha cambiado a minúscula la primer letra del nombre de la clase y no provocó ninguna irregularidad como lo haría Java respecto a la nomenclatura CamelCase, este lenguaje puede omitir tal requerimiento, permitiendo iniciar el nombre de clase con minúscula. Un punto importante es que el al igual que Java, el nombre del archivo es obligatorio comenzar con una letra mayúscula.

Manipulación de texto

Strings

En Java una cadena no es un tipo de datos básico, es un objeto básico, con propiedades y métodos, pero el lenguaje Java permite definir un nuevo objeto con el delimitador (“), por lo que es posible concatenar (unir) texto utilizando el operador (+) con los nombres de los objetos de tipo String y los trozos de texto delimitados con (“). (Pérez, 2008, pág. 10)

```
3  public class StringJ {
4      public static void main(String[] args){
5          String lenguaje= "Groovy";
6          String vs = "Java";
7
8          System.out.println("Comparando " + lenguaje + " y " + vs +"!!");
9      }
10 }
```

Figura 24. String Java

En Groovy, una cadena se puede definir de tres maneras diferentes: utilizando comillas dobles, comillas simples o barras (llamadas "cadenas Slashy"). Figura 25 ilustra las tres formas diferentes de definir una cadena.

En la siguiente figura se puede observar que en las líneas 2, 6 y 10 se están generando datos de tipo String, esto se puede ver con la llamada al nombre de clase al que pertenece la variable definida respectivamente. Todos son java.lang.String.

```
01 // Quote
02 def helloChris = "Hello, Chris"
03 println helloChris.class.name  // java.lang.String
04
05 // Single quote
06 def helloJoseph = 'Hello, Joseph'
07 println helloJoseph.class.name // java.lang.String
08
09 // Slashy string
10 def helloJim = /Hello, Jim/
11 println helloJim.class.name // java.lang.String
```

Figura 25. Definición de Strings Groovy

Groovy también es compatible con una cadena más avanzada llamada GString. Un GString es como una cadena normal, excepto que evalúa las expresiones que están incrustados dentro de la cadena, en la forma *${...}*. Cuando Groovy ve una cadena definida con comillas o barras dobles y una expresión incrustada, Groovy construye un org.codehaus.groovy.runtime.GStringImpl en lugar de un java.lang.String. Cuando se accede al GString, se evalúa la expresión. Figura 26 muestra el uso de un GString y expresiones incrustadas. (Judd & Faisal Nusairat, 2008, págs. 18-19)

```
01 def name = "Jim"
02 def helloName = "Hello, ${name}"
03 println helloName // Hello, Jim
04 println helloName.class.name // org.codehaus.groovy.runtime.GStringImpl
05
06 def helloNoName = 'Hello, ${name}'
07 println helloNoName // Hello, ${name}
08 println helloNoName.class.name  // java.lang.String
09
10 def helloSlashyName = /Hello, ${name}/
11 println helloSlashyName // Hello, Jim
12 println helloSlashyName.class.name  // org.codehaus.groovy.runtime.GStringImpl
```

Figura 26. GString y expresiones embebidas

Concatenación e Interpolación

Concatenación de texto se le denomina a la unión de trozos o fragmentos de textos y comúnmente se utiliza la conversión de números o de objetos a su representación en texto, Java convierte automáticamente los tipos de datos básicos a texto antes de unirlos y llama por cada objeto al método especial toString() para realizar la conversión de objetos nuestros o de la biblioteca de la JVM. (Pérez, 2008, pág. 17)

Ver el siguiente ejemplo:

(Como ya se había mencionado antes la concatenación se hace con el símbolo (+) y con comillas dobles (“)).

```
3  public class SumaSimpleJ {
4      public static void main(String[] args){
5          int a = 1;
6          System.out.println("El valor de a=" +a);
7          a = a +10;
8          System.out.println("ahora sumándole 10 es a=" +a);
9      }
10 }
```

Figura 27. Concatenación en Java

```
3  class SumaSimpleG {
4      static main(args){
5          def a=1
6          println "El valor de a es: " + a
7          a = a + 10;
8          println "ahora sumándole 10 es: " + a + "."
9      }
10 }
```

Figura 28. Concatenación en Groovy

Groovy, además de manejar la concatenación, maneja otro concepto llamado interpolación.

La Interpolación de strings es la capacidad de sustituir una expresión o variable dentro de una cadena Strings, definidos con comillas dobles y barras, evaluarán expresiones incrustadas dentro de la cadena (Ver líneas 2 y 10 de la Figura 26), para incrustar los valores estos deben ir entre la expresión ${} (símbolo de dólar y corchetes), o sólo $ (símbolo de dólar). Strings definidos con comillas simples no evalúan expresiones incrustadas (ver línea 6 de la Figura 26).

Java no soporta la interpolación de cadenas. Figura 29 es un ejemplo del tipo de código que necesita escribir en Java. (Judd & Faisal Nusairat, 2008, pág. 20)

```
String name = "Jim";
String helloName = "Hello " + name;
System.out.println(helloName);
```

Figura 29. Strings con Java

Figura 30 muestra el uso de interpolación:

```
1  package com.test.c1
2
3  def a=1
4  println "El valor de a es: ${a}"
5  a = a + 10;
6  println "ahora sumándole 10 es a = ${a}"
```

Figura 30. Interpolación

En el ejemplo anterior se incrustan las variables dentro de la cadena, al ejecutar el código se obtiene lo siguiente.

```
El valor de a es: 1
ahora sumándole 10 es a = 11
```

Strings Multilínea

Groovy soporta cadenas divididas en múltiples líneas. Una cadena de varias líneas se define mediante el uso de tres comillas dobles o tres comillas simples.

El soporte de serie de líneas múltiples es muy útil para crear plantillas o documentos incrustados (como plantillas XML, SQL, HTML, y así sucesivamente). Interpolación de cadenas con líneas múltiples funciona de la misma manera como lo hace con las cadenas regulares: cadenas creadas de varias líneas con comillas dobles evalúan expresiones, y usando solamente una comilla no. (Judd & Faisal Nusairat, 2008, pág. 21)

El siguiente ejemplo muestra el uso de Strings múltilínea.

```
1  package com.test.c1
2
3  lenguaje = 'Groovy'
4
5  println'''String "multilinea"
6
7  Comilas simples
8  '''
9
10 println """Comillas dobles, "evaluan" expresiones..
11 Esto es ${lenguaje}
12
13 """
```

Figura 31. Strings Multilínea

Para lo que se tiene la siguiente salida:

```
String "multilinea"

Comilas simples

Comillas dobles, "evaluan" expresiones..
Esto es Groovy
```

Java no permite realizarlo de una manera así de sencilla, para ello utiliza el operador de concatenación, el símbolo **+**.

Strings Slashy

Como se mencionó anteriormente, las barras pueden ser utilizadas para definir cadenas. La notación Slashy tiene un muy buen beneficio: barras invertidas adicionales no son necesarios para escapar caracteres especiales. La única excepción es escapar de una barra invertida: \/. La notación Slashy puede ser útil para crear una expresión regular que requiere una barra invertida o una ruta. (Judd & Faisal Nusairat, 2008, pág. 22)

En Java se requeriría usar una barra de escape para imprimir “\nueve” o de lo contrario haría un salto de línea primero y posteriormente escribiría “ueve”.

```
1  package com.test.c1
2
3  def s = /.*foo.*/
4  def dirname  = /^.*\//
5  def basename = /[Strings and GString^\/]+$/
6
7  println"${s}"
8  println"${dirname}"
9  println"${basename}"
```

Figura 32. String Slashy

Ejecutando el script se arroja lo siguiente:

```
.*foo.*
^.*/
[Strings and GString^/]+$
```

Una consecuencia de esto es que no se puede tener una barra invertida como el último carácter de una cadena Slashy directamente (o Groovy pensará que usted está tratando de escapar de la cadena de terminador de cierre - y por lo tanto no va a pensar que usted ha terminado su cadena). (Groovy)

Debe mantener un espacio después de la barra invertida y antes de la cadena Slashy para terminar correctamente. El siguiente código muestra lo mencionado:

```
1  package com.test.c1
2
3  def s2 = /Cadena Slashy con barra invertida -> \ /
4  println "${s2}"
```

Figura 33. Slashy

La ejecución muestra la siguiente línea:

```
Cadena Slashy con barra invertida -> \
```

Si de lo contrario el espacio fuera omitido, se perdería la idea del manejo de una cadena.

Importaciones

Java importa automáticamente el paquete java.lang para comodidad del programador. Groovy va un paso más allá e importa automáticamente algunos de los paquetes Java de uso común, siendo los siguientes:

- java.lang.*
- java.util.*
- java.net.*
- java.io.*
- java.math.BigInteger
- java.math.BigDecimal

Dos paquetes adicionales del Groovy JDK (GDK) también se importan:

- groovy.lang.*
- groovy.util.*

Como resultado, puede hacer referencia a las clases de estos paquetes sin especificar los nombres. (Dearle, 2010, págs. 36-37)

A continuación se presentan ejemplos en ambos lenguajes respecto a uno de los paquetes.

```groovy
package com.test.c1

class ReadFile {
    static main(args){
        FileReader fr = new FileReader("C://Groovy/saludoG.txt")
        BufferedReader br = new BufferedReader(fr);
        String linea;
        while((linea = br.readLine()) != null)
            System.out.println(linea);
        fr.close();
    }
}
```

Figura 34. Leer archivo desde Groovy

```java
package com.test.c1;

import java.io.*;

public class ReadFileJ {
    public static void main(String[] args){
        try{
            FileReader fr = new FileReader("C://Groovy R/saludoJ.txt");
            BufferedReader br = new BufferedReader(fr);
            String linea;
              while((linea = br.readLine()) != null)
                System.out.println(linea);

                fr.close();
        }catch(Exception e) {
                System.out.println(e);
        }
    }
}
```

Figura 35. Lectura de un Archivo desde Java

Como se puede observar, ambos códigos son muy diferentes, el código de la Figura 34 es muy breve comparado con el código Java de la Figura 35. Además de la importación implícita que hace groovy de la librería *java.io.**.

Se ve asimismo en el código Groovy la omisión del manejo de excepciones, que será visto más delante, además de otras características respecto a Archivos.

Constructores

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)

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*.

Antes de seguir, se examina el operador *new*. Cuando se reserva espacio de memoria para un objeto, se hace de la siguiente forma:

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.

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. A continuación se presentan las maneras de pasar parámetros a los constructores, para lo cual ambos lenguajes lo hacen de manera distinta.

Declaración de un constructor

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 distinto. (García Beltrán & Arranz)

A continuación se tiene un programa en código Java que hace el manejo de 2 constructores, uno que define explícitamente los parámetros para inicializar y otro al cual se le va a pasar los parámetros.

```
1  package com.test.c1;
2
3  public class TestPersona{
4      public static void main(String[] args){
5          PersonaN p1 = new PersonaN();
6          PersonaN p2 = new PersonaN("Edgar", 21, "hombre");
7          System.out.println("Persona 1: " + p1.toString());
8          System.out.println("Persona 2: " + p2.toString());
9      }
10 }
11
12 class PersonaN {
13     String nombre;
14     int edad;
15     String genero;
16     PersonaN(){
17         nombre = "Ana";
18         edad = 15;
19         genero = "mujer";
20     }
21     PersonaN(String nombre, int edad, String genero){
22         this.nombre = nombre;
23         this.edad = edad;
24         this.genero = genero;
25     }
26     public String toString() {
27          return "Su nombre es "+ nombre + " de " + edad +
28                  " años de edad y es " + genero;
29     }
30 }
```

Figura 36. Constructores en Java

Así como el segundo constructor es definido en la línea 6, se pueden crear otros siempre y cuando tengan diferentes parámetros (tipo/número). Se puede ver la manera en que Java declara, inicializa y define los valores, hasta aquí lo hace de igual manera Groovy.

Ahora respecto al tema de Groovy, no debería sorprender que algunas características adicionales estén disponibles, las cuales se revisarán de inmediato.

Los parámetros posicionales

Tomar en cuenta que 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, con el tipo de restricción forzada mediante el uso de la palabra clave *as*, y con tipo de restricción implícita.

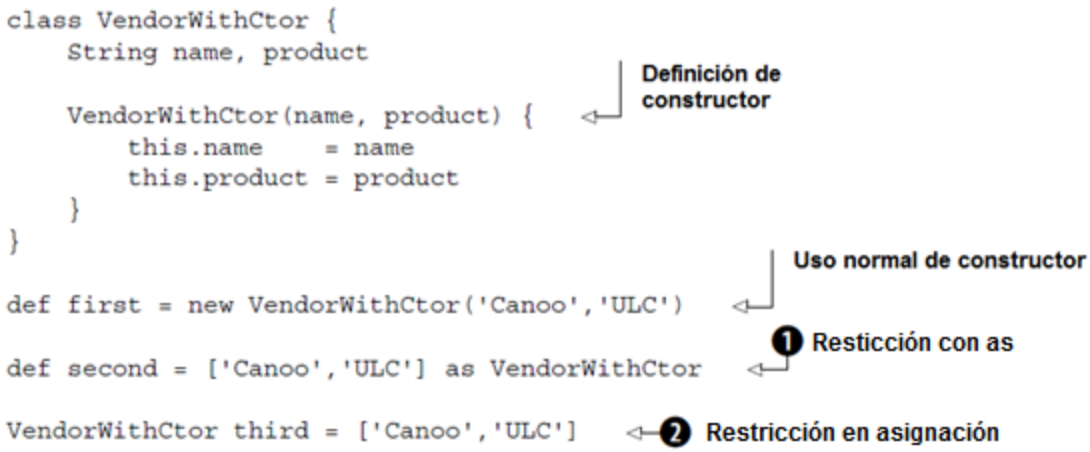

Figura 37. Llamada de constructores con parámetros posicionales

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

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. 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.

A modo de ejemplo, suponer que en la Figura 37 *VendorWithCtor* debe ser inmutable y *name* y *product* pueden ser opcionales. Se requerirían cuatro constructores: uno vacío, uno para establecer *name*, uno para establecer *product*, y uno para fijar ambos atributos. Para hacer las cosas peor, no se podría tener un constructor 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: la manera especial Groovy de apoyar parámetros con nombre viene al rescate de nuevo.

Figura 38 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 38. Parámetros nombrados

El ejemplo en Figura 38 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 39. Pasar un map a un constructor

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

Cada GroovyBean cuenta por defecto con este incorporado constructor *Map*. Este constructor trabaja iterando el objeto *map* y llama a la correspondiente propiedad *setter* 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)

A continuación se muestran códigos tanto en Java como en Groovy para una misma clase respectivamente y se demuestra la ventaja de usar parámetros nombrados:

Código Java:

```
1  package com.test.c1;
2
3  public class ParamNomJ {
4      public static void main(String[] args){
5          DatosP cn = new DatosP("Lau");
6          DatosP ca = new DatosP("Durán");
7
8          System.out.println("Nombre:" +cn.getNombre());
9          System.out.println("Apellido:" +ca.getApellido());
10     }
11 }
12
13 class DatosP{
14     String nombre, apellido;
15
16     DatosP(String nombre){
17         this.nombre= nombre;
18     }
19
20     DatosP(String apellido){
21         this.apellido= ape
22     }
23
24     public String getNombr
25         return nombre;
26     }
27     public String getApellido(){
28         return apellido;
29     }
30 }
```

Duplicate method DatosP(String) in type DatosP

1 quick fix available:

Rename method 'DatosP' (Ctrl+2, R)

Press 'F2' for focus

Figura 40. Clase Java

De inmediato el compilador marca un error debido a que se está duplicando el constructor con el mismo tipo de dato aunque la variable sea diferente.

A continuación se presenta la clase anterior codificada en Groovy.

```groovy
package com.test.c1

class ParamNomG {
    static main(args){
        Datos cn = new Datos(nombre: "Lau")
        Datos ca = [apellido: "Durán"] as Datos
        println "Nombre: ${cn.nombre}"
        println "Apellido: ${ca.apellido}"
    }
}

class Datos{
    String nombre, apellido
}
```

Figura 41. Clase Groovy usando parámetros nombrados para los constructores

Las siguientes líneas muestran la salida del código anterior:

```
Nombre: Lau
Apellido: Durán
```

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 la clase Datos.

Dentro de los constructores de los ejemplos manejados anteriormente se ha visto el uso de la palabra *this*, a continuación se explica su uso.

La palabra clave *this*

En algunas ocasiones, un método necesita referirse al objeto que lo invocó. Para permitir esta situación, Java define la palabra clave *this*, la cual puede ser utilizada dentro de cualquier método para referirse al objeto actual. *this* es siempre una referencia al objeto sobre el que ha sido llamado el método. Se puede usar *this* en cualquier lugar donde esté permitida una referencia a un objeto del mismo tipo de la clase actual. (Schildt, 2009, pág. 120)

Se tiene el siguiente ejemplo con el uso de *this*:

```
12 class PersonaN {
13     String nombre;
14     int edad;
15     String genero;
16     PersonaN(){
17         nombre = "Ana";
18         edad = 15;
19         genero = "mujer";
20     }
21     PersonaN(String nombre, int edad, String genero){
22         this.nombre = nombre;
23         this.edad = edad;
24         this.genero = genero;
25     }
26     public String toString() {
27          return "Su nombre es "+ nombre + " de " + edad +
28                  " años de edad y es " + genero;
29     }
30 }
```

Figura 42. Uso de this

Se puede observar que en las líneas 22 a la 24 dentro del constructor en el código anterior se hace uso de esta palabra, lo que hace es hacer referencia directamente al objeto y resolver cualquier colisión entre nombres.

Métodos

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

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.

Declaración de un método

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

Más general, 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)

Al igual que con las variables dinámicas, se debe utilizar la palabra clave *def* al definir un método que tiene un tipo de retorno dinámico.

Se tienen a continuación los ejemplos para declarar métodos en Groovy y Java, mostrando lo que ya se ha mencionado en párrafos anteriores:

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

Figura 43. Método Groovy

A continuación se muestra la forma clásica de Java:

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

Figura 44. Método Java

Palabra clave *return*

Un punto importante que se puede ver en el ejemplo en Groovy es que no usa la palabra *return* como lo hace Java para regresar el String que se maneja en el ejemplo. 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.

Propiedades y GroovyBeans

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 que 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).

La siguiente figura muestra cómo Groovy trabaja esta característica:

```
1  package com.test.c1
2
3  class Customer{
4      int id;
5      String name;
6
7      static main(args){
8          def customer = new Customer()
9          customer.setName("Brian Beausang")
10         println customer.getName()
11         customer.name = "Carol Coolidge"
12         println customer.name
13     }
14 }
```

Figura 45. Getters/Setters

Las siguientes líneas muestran la salida del código:

```
Brian Beausang
Carol Coolidge
```

Este fragmento de código muestra cómo se pueden utilizar opcionalmente métodos getter/setter para manipular el nombre de campo de nuestra clase *Customer*, o se puede utilizar la sintaxis de acceso a campo *customer.name*. Es importante tener en cuenta que cuando se utiliza la sintaxis de acceso a campo *customer.name*, no se está accediendo directamente al campo. El *getter* apropiado o método *setter* es llamado en su lugar. (Dearle, 2010, pág. 40)

A continuación se muestran un par de ejemplos para ilustrar la facilidad que presta Groovy respecto a este tema. Se ha tomado una clase que fue utilizada en temas anteriores.

En Java es necesario implementar los *getters* y *setters* como se muestra en el ejemplo siguiente:

```java
class Animal{
    String nombre, alimento;
    int edad;

    public String comer(String alimento){
        return "le gusta comer" + alimento;
    }
    public void setNobre(String nombre){
        this.nombre = nombre;
    }
    public String getNombre(){
        return nombre;
    }
    public void setEdad(int edad){
        this.edad = edad;
    }
    public int getEdad(){
        return edad;
    }
    public void setAlimento(String alimento){
        this.alimento = alimento;
    }
    public String getAlimento(){
        return alimento;
    }
    public String toString(){
        return "El animal es un " + nombre +", tiene " + edad +" años de edad";
    }
}
```

Figura 46. Implementación de getters y setters en clase Java

Por otro lado se tiene el código de la misma clase implementada en Groovy y se muestra que no son necesarios manejarlos explícitamente para que la clase funcione correctamente.

```groovy
class AnimalG{
    String nombre, alimento
    int edad

    AnimalG(String nombre, int edad){
        this.nombre = nombre
        this.edad = edad
    }
    def comer(alimento){
        "le gusta comer ${alimento}"
    }
    String toString(){
        "El animal es un ${nombre}, tiene ${edad} años de edad"
    }
}
```

Figura 47. Clase Groovy sin getters y setters explícitamente

Se puede observar que el código Groovy es bastante fácil de leer comparado con Java, ya que no se tienen tantas líneas de código.

Cuando se utiliza el *set* y *get* de manera explícita al tratar el valor de una variable y además la variable correspondiente se marca explícitamente con un modificador de acceso, es necesario implementar el getter y setter apropiado, o de lo contrario lanzará una excepción. La siguiente figura muestra el caso para implementar getters y setters explícitamente:

```
1  package com.test.c1
2
3  class Persona {
4      public nombre, apellido
5      def documento
6
7      static main(args) {
8          def p = new Persona();
9          p.setNombre("Ana")
10         p.setApellido("Gil")
11         p.documento = "Pasaporte"
12
13         println "La persona es ${p.nombre} ${p.apellido} y trae ${p.documento}"
14     }
15
16     def setNombre(nombre){
17         this.nombre = nombre
18     }
19     def getNombre(){
20         nombre
21     }
22     public void setApellido(apellido){
23         this.apellido = apellido
24     }
25     String getApellido(){
26         apellido
27     }
28 }
```

Figura 48. public explícitamente

La ejecución es la siguiente:

```
La persona es Ana Gil y trae Pasaporte
```

Método principal *main()*

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()*.

La palabra clave *public* es un *especificador de acceso* que permite al programador controlar la visibilidad de los miembros de una clase. Cuando un miembro de una clase va precedido por el especificador *public*, entonces es posible acceder a ese miembro desde cualquier código fuera de la clase en que se ha declarado (lo opuesto al especificador *public* es *private*, que impide el acceso a un miembro declarado como tal desde un código fuera de su clase). En este caso, *main( )* debe declararse como *public*, ya que debe ser llamado por un código que está fuera de su clase cuando el programa comienza. La palabra clave *static* (estático) permite que se llame a *main()* sin tener que referirse a ninguna instancia particular de esa clase. Esto es necesario, ya que 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)

Cualquier información que sea necesaria pasar a un método se almacena en las variables especificadas dentro de los paréntesis que siguen al nombre del método. A estas variables se las denomina *parámetros*. Aunque un determinado método no necesite parámetros, es necesario poner los paréntesis vacíos. 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 es el valor 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 los siguientes ejemplos, se puede observar que Groovy no necesita declarar el tipo de dato junto con la variable que va a llegar, mientras que en Java estrictamente se deben definir.

Código Groovy:

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

Figura 49. Parámetro sin tipo de variable

A continuación se muestra la forma Java:

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

Figura 50. 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 por ejemplo un String sin ningún problema.

Parámetros Opcionales

A continuación se muestran un ejemplo en el cual al método se le puede pasar una cantidad dinámica de parámetros:

```
3  class POpcionales {
4      static main(args){
5          pased(1, 2, 3, 4, 5, "Hello!", 6, 7, 8, 9, 10)
6      }
7
8      def static pased(a, Object[] optionals){
9          print 'Parámetros pasados: ' + a
10         for(o in optionals)
11             print ','  + o
12     }
13 }
```

Figura 51. Parámetros Opcionales Groovy

El número de parámetros recibidos en el método en la línea 8 es indefinido gracias a la expresión Object[][5], la variable *'a'* únicamente está definiendo el primer valor del método en la línea 5 y la expresión que le sigue se encarga de recibir los demás parámetros sin problema alguno, notar que se pueden pasar argumentos de diferente tipo.

Lo anterior se podría lograr con un ArrrayList en Java de la siguiente manera:

[5] Es una expresión booleana que será verdadera mientras su longitud sea mayor a 0 (length > 0.)

```java
import java.util.*;

public class POpcionalesJ {
    public static void main(String[] args){
        ArrayList l2 = new ArrayList();
        l2.add(3);
        l2.add("a");
        l2.add(6);
        l2.add("8");
        l2.add("R");

        pased2(l2);
    }

    public static void pased2(ArrayList l2){
        for(int i=0; i<l2.size(); i++){
            System.out.print(l2.get(i) + ", ");
        }
    }
}
```

Figura 52. Parámetros Opcionales Java

El código mostrado anteriormente demuestra que Java necesita mucho código para trabajar y lograr lo que Groovy puede realizar con facilidad. Java necesitó de la implementación de un ArrayList para pasar parámetros de diverso tipo. Una vez más se ve lo fácil que es trabajar con Groovy brindando un código limpio.

Java permite pasar argumentos variables con la siguiente sintaxis marcada en negrita:

nombreMetodo(String a, int b, ***int…x****){}*

Pero cabe resaltar que los argumentos variables deben de ser del mismo tipo. Si existen argumentos normales, estos deben ir al principio y los argumentos variables deberán ir al final como lo muestra la sintaxis anterior.

Paso de Parámetros

En programación hay dos formas de paso de parámetros a un método.

En Java los objetos pasados a los parámetros de un método son siempre referencias a dichos objetos, lo cual significa que cualquier modificación que se haga a esos objetos dentro del método afecta al objeto original. En cambio, las variables de un tipo primitivo pasan por valor, lo cual significa que se pasa una

copia, por lo que cualquier modificación que se haga a esas variables dentro del método no afecta a la variable original.

En Java todos los argumentos que son objetos son pasados por referencia. (Ceballos, 2010)

Se muestra un ejemplo para manejar este tema:

```
1  package com.test.metodos;
2
3  public class PassTest {
4      public static void changeInt(int value) {
5          value = 55;
6      }
7      public static void changeObjectRef(MyDate ref) {
8          ref = new MyDate(1, 1, 2000);
9      }
10     public static void changeObjectAttr(MyDate ref) {
11         ref.setDay(4);
12     }
13     public static void main(String args[]) {
14         int val = 11;
15         changeInt(val);
16         System.out.println("Variable value es: " + val);
17         MyDate date = new MyDate(22, 7, 1964);
18         changeObjectRef(date);
19         System.out.println("MyDate: " + date);
20         changeObjectAttr(date);
21         System.out.println("MyDate: " + date);
22     }
23 }
24 class MyDate{
25     private int day = 1;
26     private int month = 1;
27     private int year = 2000;
28     public MyDate(int day, int month, int year) {
29         this.day = day;
30         this.month = month;
31         this.year = year;
32     }
33     public void setDay(int day) {
34         this.day = day;
35     }
36     public String toString() {
37         return (day + "-" + month + "-" + year);
38     }
39 }
```

Figura 53. Paso de Parámetros (Oracle and/or its affiliates, 2010)

La salida es la siguiente:

```
Variable value es: 11
MyDate: 22-7-1964
MyDate: 4-7-1964
```

La variable *value* sigue siendo 11, en el método *changeInt* sólo se pasó una copia y no afecta la original, el objeto *MyDate* no se cambia en el método *changeObjectRef* debido a que dentro se está creando un nuevo objeto, sin embargo, el método *changeObjectAttr* cambia el atributo *day* del objeto *MyDate*.

Para Groovy no se encontró información al respecto.

Closures

Los closures son fragmentos de código anónimos que se pueden asignar a una variable.

Los closures 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}*.

En la siguiente figura se muestran diferentes maneras de declararlos, más adelante se mostrarán ejemplos para explicar con más detalle la manera de usar esta característica.

```
1  package com.test.closures
2
3  // Sin parámetros
4  Closure simpleCloj1 = {
5      println 'Hello, World!'
6  }
7
8  //Con un parámetro indefinido
9  def simpleCloj2 = { obj ->
10     println "Hello, $obj!"
11 }
12
13 // Con un parámetro definido de tipo String
14 def simpleCloj3 = { String obj ->
15     println "Hello, $obj!"
16 }
17
18 //Si sólo estamos pasando un parámetro, el argumento puede
19 //ser omitido y podemos usar la palabra clave it para el acceso
20 def simpleCloj4 = {
21     println "Hello, $it!"
22 }
23
24 // Tomar múltiples parámetros
25 def twoParamsCloj = { obj1, obj2 ->
26     println "$obj1, $obj2!"
27 }
28
29 simpleCloj1()
30 simpleCloj2.call("Groovy")
31 simpleCloj3.doCall(':-)')
32 simpleCloj4(5)
33 simpleCloj4 'Str'
34 twoParamsCloj('Java', 'Groovy')
```

Figura 54. Closures

Como se observa, tiene gran similitud con los métodos de Java, el manejo de parámetros lo puede realizar con tipos dinámicos.

En las líneas 20 a la 22 se trata un ejemplo en el cual se maneja un closure que recibe un único parámetro, permitiendo hacer referencia a este parámetro como *it*, de esta manera se es libre de tener que definir explícitamente el parámetro. En la línea 32 se manda a llamar con un valor entero, podría tener cualquier otro tipo de dato sin ningún problema como lo muestra la línea 33 donde se pasa un String.

En las líneas 29 a la 34 se observan las formas para llamar a un closure, se hace uso de los métodos *call* y *doCall,* en la línea 32 y 33 se llama a *simpleCloj4*, pero en la primera se colocan paréntesis y en la siguiente línea son omitidos. En estos

casos, para llamar algún closure puede o no llevar paréntesis para pasarle los parámetros.

El símbolo “->”, separa las declaraciones de parámetros del cuerpo del cierre.

Esta característica de Groovy, como ya se ha mencionado, es una de las más relevantes del lenguaje, la cual marca una diferencia en cuanto a funcionalidad respecto de Java, otra manera de trabajarlos es pasarlos como parámetro a otro closure, opción que Java permite al trabajar con sus respectivos métodos. Los siguientes códigos muestran lo ya mencionado:

```
3  class ParamCG2 {
4      static main(args){
5          sum(5, mult(3))
6      }
7      def static mult = { n ->
8          n * 5
9      }
10     def static sum = {s, n ->
11         s = s + n
12         println "La suma es: $s"
13     }
14 }
```

Figura 55. Closure como parámetro

La ejecución da como resultado lo siguiente:

```
La suma es: 20
```

Ahora se muestra cómo Java trabaja la clase anterior pero haciendo uso de métodos.

```
3  public class ParamCJ2 {
4      public static void main(String[] args){
5          sum(5, mult(3));
6      }
7      public static int mult(int n){
8          return n * 5;
9      }
10     public static void sum(int s,int n){
11         s = s + n;
12         System.out.println("La suma es: "+ s);
13     }
14 }
```

Figura 56. Método como parámetro

La ejecución arroja lo que sigue:

```
La suma es: 20
```

Al igual que los métodos en Java, es posible pasar closures a otros closures como parámetro. En la línea 10 de la Figura 55 se está pasando un closure como parámetro con la variable *n*.

Esta característica de groovy da la posibilidad de un código sin tanta verbosidad.

La Figura posterior muestra un ejemplo más de cómo se trabaja con esta característica:

```
3  def pickEven(n, block) {
4      for(int i = 2; i <= n; i += 2) {
5          block(i)
6      }
7  }
8
9  pickEven(10, { print" $it"} ) // Cierre pasado al último
10
11 println ''
12
13 pickEven(10){
14     print " $it"
15 }
```

Figura 57. Closure, último parámetro

En la declaración del método de la Figura 57, en la línea 3 se pasan como parámetros 2 variables, *n* y *block*, *n* va a representar un dato recibido, mientras que *block* contiene una referencia a un closure, el uso se muestra en las líneas 9 y 13 pero de maneras distintas, en la primer línea es pasado entre paréntesis, mientras que en la segunda llamada se pasa entre paréntesis sólo el valor *n* y posteriormente entre llaves es mandado a llamar el closure.

Únicamente el closure llama un *print* para imprimir los valores del bucle for.

Cabe resaltar que cuando es pasado un closure como parámetro entre los paréntesis, este debe ir declarado como último argumento, como se puede observar en la línea 9.

Closure dinámico.

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

El código siguiente muestra lo anterior y posteriormente es explicado.

```
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 58. Closure dinámico

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, lo anterior pudiera causar un poco de confusión al principio pero poco a poco va resultando familiar la sintaxis.

Al ejecutarse el código se presenta lo siguiente:

```
Use specialized implementation
Use specialized implementation
Using default implementation
```

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)

Fundamentos de la gestión de excepciones

Una excepción, en Java, es un objeto que describe una condición excepcional, es decir, un error que ha ocurrido en una parte de un código. Cuando surge una condición excepcional, se crea un objeto que representa esa excepción y se *envía* al método que ha originado el error. Ese método puede decidir entre gestionar él mismo la excepción o pasarla. En cualquiera de los dos casos, en algún punto la excepción es *capturada* y procesada.

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)

La forma general de un bloque de gestión de excepciones es la siguiente:

```
try {
   // bloque de código a monitorear por errores
}
catch (TipoExcepcionl exOb) {
   // gestor de excepciones para ExcepciónTipol
}
catch (TipoExcepcion2 exOb) {
   // gestor de excepciones para ExcepciónTipo2
}
// ...
finally {
   // bloque de código que se debe ejecutar después de que el bloque try termine
}
```

Figura 59. Gestión de excepciones

Donde *TipoExcepcion* es el tipo de la excepción que se ha producido. (Schildt, 2009, págs. 205-206)

Tomar en cuenta lo siguiente:

- El *finally* de los *try catch* siempre se va a ejecutar, por más que dentro del try{} exista un *return*. Sólo no se ejecutará si dentro del *try{}* hay un *System.exit(0)*;
- Puede no haber ningún un *catch(…){}* , pero si es así, necesariamente debe haber un *finally* (el *try{}* no puede estar solo, siempre debe haber de uno a muchos catch o un finally o ambas).
- Puede haber más de un *catch(…){}*.
- Solo puede haber un *finally(){}*. (Horna, 2010, pág. 48)

Tipos de Excepciones

- *Excepciones comprobadas*. Estas son condiciones excepcionales que una aplicación bien escrita debe anticipar y recuperar.

  Están sujetas a la captura o especificar los requerimientos. Todas las excepciones son excepciones comprobadas, a excepción de las indicadas por Error, RuntimeException, y sus subclases.

- Error. Estas son condiciones excepcionales que son externas a la aplicación, y que la aplicación por lo general no puede anticipar o recuperar. Los errores no están sujetos a la captura o especificar los Requerimientos. Los errores son las excepciones indicadas por Error y sus subclases.
- Excepción en tiempo de ejecución. Estas son condiciones excepcionales que son internas a la aplicación, y que la aplicación por lo general no puede anticipar o recuperar. Estos por lo general indican errores de programación, tales como errores lógicos o el uso indebido de una API.

  Las Excepciones de tiempo de ejecución no están sujetas a la captura o especificar los Requerimientos. Excepciones en tiempo de ejecución son los indicados por RuntimeException y sus subclases.

  Errores y excepciones de tiempo de ejecución se conocen colectivamente como E*xcepciones no comprobadas*. (Oracle)

En general las excepciones se clasifican en comprobadas y no comprobadas.

Throw

El propio programa puede lanzar explícitamente una excepción mediante la sentencia **throw**. La forma general de esta sentencia es la siguiente:

throw objetoThrowable;

Donde *objetoThrowable* debe ser un objeto del tipo *Throwable* o una subclase de *Throwable*.

La ejecución del programa se para inmediatamente después de una sentencia *throw*; y cualquiera de las sentencias que siguen no se ejecutarán. A continuación se inspecciona el bloque *try* más próximo que la encierra, para ver si contiene una sentencia *catch* que coincida con el tipo de excepción. Si es así, el control se transfiere a esa sentencia. Si no, se inspecciona el siguiente bloque *try* que la engloba, y así sucesivamente. Si no se encuentra una sentencia *catch* cuyo tipo coincida con el de la excepción, entonces el gestor de excepciones por omisión interrumpe el programa e imprime el trazado de la pila.

Enseguida se presenta un programa de ejemplo que crea y lanza una excepción. El gestor que captura la excepción la relanza al gestor más externo.

```
// Ejemplo de la sentencia throw.
class ThrowDemo (
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("!Capturada dentro de demoproc.");
      throw e; // se relanza la excepción
    }
  }

  public static void main(String args[]) {
    try {
      demoproc () ;
    } catch(NullPointerException e) {
      System.out.println("Recapturada: " + e);
    }
  }
}
```

Figura 60. Excepción Java

Este programa tiene dos oportunidades para tratar el mismo error. En la primera, el método *main()* establece un contexto de excepción, y a continuación llama a *demoproc()*. El método *demoproc()* entonces establece otro contexto de gestión de excepciones e inmediatamente lanza una nueva instancia de *NullPointerException*, que se captura en la siguiente línea. Entonces la excepción se relanza. La salida resultante es la siguiente:

```
Capturada dentro de demoproc.
Recapturada: java.lang.NullPointerException: demo
```

Groovy no presenta problemas con lo anterior y lo maneja de la misma forma, imprimiendo el resultado esperado como lo demuestra el posterior código:

```groovy
package com.test.c1

class ThrowDemoG {
    static demoproc() {
        try {
            throw new NullPointerException("demo")
        } catch (NullPointerException e) {
            println "!Capturada dentro de demoproc."
            throw e; // se relanza la excepción
        }
    }

    static main(args) {
        try {
            demoproc()
        } catch (NullPointerException e) {
            println "Recapturada: ${e}"
        }
    }
}
```

Figura 61. Excepción Groovy

Se muestra la salida en las siguientes líneas:

```
!Capturada dentro de demoproc.
Recapturada: java.lang.NullPointerException: demo
```

Throws

Si un método puede dar lugar a una excepción que no es capaz de gestionar él mismo, se debe especificar este comportamiento de forma que los métodos que llamen al primero puedan protegerse contra esa excepción. Para ello se incluye una cláusula *throws* en la declaración del método. Una cláusula *throws* da un listado de los tipos de excepciones que el método podría lanzar. Esto es necesario para todas las excepciones, excepto las del tipo *Error* o *RuntimeException*, o cualquiera de sus subclases.

Todas las demás excepciones que un método puede lanzar se deben declarar en la cláusula *throws*. Si esto no se hace así, el resultado es un error de compilación.

La forma general de la declaración de un método que incluye una sentencia *throws* es la siguiente:

```
tipo nombre_ método ( lista_de_parámetros ) throws lista_de_excepciones
{
// cuerpo del método
}
```

Figura 62. Declaración de throws

Donde, *lista_de_excepciones* es una lista de las excepciones que el método puede lanzar, separadas por comas. (Schildt, 2009, págs. 213-215)

```
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Dentro de throwOne.");
    throw new IllegalAccessException("demo"):
  }
  public static void main(String args[]) {
    try {
      throwOne ();
    } catch (IllegalAccessException e) {
      System.out.println("Capturada" + e);
    }
  }
}
```

Figura 63. Uso de throws

La salida generada por la ejecución de este programa es la siguiente:

```
Dentro de throwOne
Capturada java.lang.IllegalAccessException: demo
```

Con Throws se advierte de la excepción.

He aquí el ejemplo anterior al estilo Groovy para demostrar que no se tiene problema alguno en la sintaxis de excepciones. Y de igual manera se obtiene el resultado deseado.

```groovy
package com.test.c1

class ThrowsDemoG {
    static throwOne() throws IllegalAccessException {
        println "Dentro de throwOne."
        throw new IllegalAccessException("demo")
        }

    static main(args) {
        try {
            throwOne()
        } catch (IllegalAccessException e) {
            println "Capturada ${e}"
        }
    }
}
```

Figura 64. ThrowsDemo Grovy

Lo anterior es indispensable manejar en Java para una correcta ejecución de programa cuando se tratan excepciones. Groovy puede manejarlo de la misma manera sin problema alguno como se pudo ver, utilizando la misma lógica. Pero como es Groovy, ya no es sorprendente que haga algo para evitar escribir código de más.

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 del 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. A continuación se muestra el comportamiento habitual. (König & Glover, 2007, pág. 171)

```groovy
package com.test.c1

class ThrowsDemoG {
    static throwOne(){// throws IllegalAccessException {
        println "Dentro de throwOne."
        throw new IllegalAccessException("demo")
        }

    static main(args) {
        try {
            throwOne()
        } catch (IllegalAccessException e) {
            println "Capturada ${e}"
        }
    }
}
```

Figura 65. Excepción manejada como no comprobada

Es lanzada la siguiente excepción:

```
Dentro de throwOne.
Capturada java.lang.IllegalAccessException: demo
```

Comparado con el código Java, Groovy no obliga a advertir la excepción que se debería en la línea 4. Java de inmediato habría marcado un error de compilación.

Groovy con menos ceremonia que Java. Es cristalino en el manejo de excepciones. Java obliga manejar excepciones comprobadas.

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. He aquí otro ejemplo de la respuesta de Groovy para el manejo de excepciones: (Subramaniam, 2013, pág. 17)

```groovy
package com.test.c1

class ReadFileGE {
    static main(args){
        FileReader fr = new FileReader("C://Groovy/saludoG.txt")
        BufferedReader br = new BufferedReader(fr);
        String linea;
        while((linea = br.readLine()) != null)
            System.out.println(linea);
        fr.close();
    }
}
```

Figura 66. Excepción lanzada sin ser manejada

La excepción que se debería manejar no es tratada, pero sin embargo no marca error de compilación y como el archivo no se encontró es lanzada la excepción correspondiente. Lanza un FileNotFoundException.

```
Caught: java.io.FileNotFoundException: C:\Groovy\saludoG.txt (El sistema no puede encontrar la ruta especificada)
java.io.FileNotFoundException: C:\Groovy\saludoG.txt (El sistema no puede encontrar la ruta especificada)
        at com.test.c1.ReadFileGE.main(ReadFileGE.groovy:5)
```

Se presenta el código Java con el manejo de la Excepción:

```java
package com.test.c1;

import java.io.*;

public class ReadFileJE {
    public static void main(String[] args){
        try{
            FileReader fr = new FileReader("C:/Groovy/saludoG.txt");
            BufferedReader br = new BufferedReader(fr);
            String linea;
            while((linea = br.readLine()) != null)
                System.out.println(linea);
            fr.close();
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

Figura 67. Manejo de excepciones

Al ejecutar lo anterior se tiene lo siguiente:

```
java.io.FileNotFoundException: C:\Groovy\saludoG.txt (El sistema no puede encontrar la ruta especificada)
```

Ahora se tiene el código Groovy de la Figura 66 con la ruta de archivo correcta:

```groovy
package com.test.c1

class ReadFileGE {
    static main(args){
        FileReader fr = new FileReader("C://Groovy R/saludoG.txt")
        BufferedReader br = new BufferedReader(fr);
        String linea;
        while((linea = br.readLine()) != null)
            System.out.println(linea);
        fr.close();
    }
}
```

Figura 68. Excepción manejada como no comprobada 2

Se lee el archivo correctamente e imprime las líneas que contiene.

```
Groovy:

Manejo de Excepciones...
```

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

Archivos

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.

Lectura de Archivos

El siguiente código representa la manera de leer un archivo desde Java.

```
1  package com.test.io;
2  import java.io.*;
3
4  public class ReadFileJ {
5      public static void main(String[] args) {
6          try {
7               BufferedReader read = new BufferedReader(
8               new FileReader("C:/Groovy R/saludoG.txt"));
9               String line = null;
10              while((line = read.readLine()) != null) {
11                 System.out.println(line);
12              }
13         } catch(IOException ex) {
14              ex.printStackTrace();
15         }
16     }
17 }
```

Figura 69. Leer archivo desde Java

Java hace todo un esfuerzo para sólo leer el archivo. Es más definido para mostrar el flujo de datos. Al ser más definido, se crea un código más pesado.

La biblioteca estándar de Java define en su paquete java.io, una colección de clases que soportan estos algoritmos para leer y escribir.

Las clases *BufferedReader*, *FileReader* utilizadas en el ejemplo anterior entre otras más, son proporcionadas por el JDK de Java, necesarias para el manejo de archivos. (Ceballos, 2010)

El lenguaje Groovy hace el manejo de archivos más fácil, se tiene a continuación una clase con el código necesario para leer el contenido de un archivo:

```
1  package com.test.io
2
3  class ReadFileG {
4      static main(args){
5          def file = new File("C:/Groovy R/saludoG.txt")
6          file.eachLine{String line->
7              println line
8          }
9      }
10 }
```

Figura 70. Leer archivo desde 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*.

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

En el ejemplo sólo se utilizó la clase *File*, la cual es únicamente la representación de un archivo y rutas de directorio.

De igual forma se puede observar en las líneas 6 a 8 el uso de un *closure*, lo que pareciera ser desde Java un método, este hace empleo del método *eachLine*, el

cual itera a través del archivo línea por línea, cada línea se pasa al closure para posteriormente imprimirla.

Escribir en un archivo

El siguiente código muestra la manera de Java para escribir dentro de un archivo:

```
1  package com.test.io;
2  import java.io.*;
3
4  public class WriteFileJ {
5      public static void main(String[] args){
6          try{
7              File file = new File("C:/Groovy R/IO/writef.txt");
8              BufferedWriter bw = new BufferedWriter(new FileWriter(file));
9              bw.write("Java & Groovy!!");
10             bw.flush();
11             bw.close();
12         }catch(IOException e){
13             System.out.println(e);
14         }
15     }
16 }
```

Figura 71. Escribir en un archivo desde Java

Como se puede ver, son necesarias la implementación de la clase *BufferedWriter*, además del método *flush()*. El método *flush()* es utilizado para garantizar que el último de los datos va a estar realmente en el archivo, si no se coloca ese método no se escribirá nada en el archivo. (Ceballos, 2010)

La clase *BufferedWriter* es usada para hacer la clase *FileWriter* más eficiente, y esta última es usada para escribir archivos de caracteres, su método *write()* permite escribir caracteres o cadenas (Strings) a un archivo.

Por otro lado y para comparar, se muestra la manera en la que trabaja Groovy para hacer el mismo trabajo que Java en el código anterior.

```groovy
package com.test.io

def file = new File('C:/Groovy R/IO/salidaFile.txt')
def out = file.newPrintWriter()
10.times {
    out.println ("Imprimiendo línea $it")
}
out.close()
```

Figura 72. Escribir en un archivo desde Groovy

Como se puede observar, sólo se hizo uso de la clase File. El GDK se hace presente al usar el método *newPrintWriter*(), este crea un nuevo *PrintWriter* para el archivo pasado a *file*, utilizando especificado conjunto de caracteres. (Groovy) De la misma manera, se utiliza el método *times* que permite la escritura de un bucle *'for'* de una manera más sencilla, el tema de bucles se atenderá más adelante.

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

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 siguiente tabla muestra las palabras reservadas de Java:

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

Figura 73. Palabras reservadas Java (Schildt, 2009)

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)

El siguiente código muestra el uso de algunas palabras reservadas del lenguaje groovy, la que más se ha manejado en ejemplos anteriores ha sido *def* para referencia de variables sin tipo.

```
1  package com.test.kws
2
3  class KeyWordsG {
4      static main(args){
5          def key = "Key Words"
6          println "$key"
7          for(i in 0..3){
8              print "$i "
9          }
10         println '\n\nUso de "it":'
11         (5..7).each{
12             print it
13         }
14     }
15 }
```

Figura 74. Uso de nuevas palabras reservadas Groovy

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 en el código de la Figura 37.

De la misma forma se pueden observar palabras reservadas que vienen del lenguaje Java, como *static, for, try,* y todas las demás manejadas en la Figura 73.

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 que a continuación se verán.

Estructuras de control condicionales

Todas ellas 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.

Evaluación de pruebas booleanas

En Java la evaluación de expresiones booleanas consiste en dos números o variables que son comparadas entre sí a través de algún operador. Los operadores utilizados para comparación están constituidos por uno o dos símbolos. Los operadores son los siguientes:

==	Comprueba si los valores de dos operandos son iguales o no, si sí, entonces condición sea verdadera.
!=	Comprueba si los valores de dos operandos son iguales o no, si los valores no son iguales, entonces la condición se convierte en realidad.
>	Comprueba si el valor del operando de la izquierda es mayor que el valor del operando derecho, si sí, entonces condición sea verdadera.
<	Comprueba si el valor del operando de la izquierda es menor que el valor del operando derecho, si es así, entonces la condición sea verdadera.
>=	Comprueba si el valor del operando de la izquierda es mayor o igual al valor del operando derecho, si sí, entonces condición sea verdadera.
<=	Comprueba si el valor del operando de la izquierda es menor o igual al valor del operando derecho, si es así, entonces la condición sea verdadera.

Figura 75. Operadores relacionales

Es posible combinar varias comparaciones a través de los operadores && y || (“and” y “or” respectivamente).

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 siguiente Figura (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 76. Secuencia de reglas utilizadas para evaluar un prueba booleana.

El uso de ellas se verá más adelante.

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)

A continuación se muestran los posibles manejos de esta estructura para ambos lenguajes y se hablará de las diferencias manejadas en ambos:

```groovy
package com.test.estruct

def n = 1;
int y = 2
String s = "hi", ss

if ((n = y)) {
    println "Hace una asignación -> n = $n"
}

if(y){
    println "Valor distinto de 0"
}

if(ss){
    println "cadena vacía"
}else if(s){
    println "cadena no vacía"
}

if(y == s){
    println "String"
}else if(y != s){
    println "permite comparar variables Integer con String "
}

if(0){
    println "0"
}else if(1){
    println "valor distinto de 0"
}
```

Figura 77. if-else usando reglas de la Figura 76

La ejecución arroja los siguientes resultados:

```
Hace una asignación -> n = 2
Valor distinto de 0
cadena no vacía
permite comparar variables Integer con String
valor distinto de 0
```

El código de la Figura 77 muestra que Groovy es más extenso en cuanto a las expresiones que permite evaluar, en base al ejemplo anterior se concluye lo siguiente:

- Un número distinto de 0 es siempre true.
- Permite evaluar una variable como en las líneas 11, 15 y 17 de la figura anterior, puesto que toma en cuenta el valor que se le pasa.
- Permite comparar un Integer con un String como en la línea 21.

- Una asignación se toma como true.

Java por el contrario no acepta ninguna de las sentencias anteriores, se muestra más estricto en cuanto a los valores evaluados dentro de las expresiones.

```
1  package com.test.estruct;
2
3  public class ElseIfJ {
4      public static void main(String[] args){
5          int a = 12, c=0;
6          String b = "f";
7
8          if(a == 12){
9              System.out.println("a == 12");
10         }
11         else{
12             System.out.println("null ");
13         }
14
15         if((b == "df") && (c!= a)){
16                 System.out.println("uso de &&");
17         }else if((b == "f") && (c!= a)){
18             System.out.println("uso de && y !=");
19         }else{
20             System.out.println("Ninguna de las anteriores");
21         }
22     }
23 }
```

Figura 78. if-else Java

Como se puede observar, es grande la diferencia manejada en Groovy respecto a Java, siendo éste más restrictivo respecto a los parámetros recibidos.

Operador ternario y Elvis (?:)

El operador ternario se puede decir que es un if-else abreviado, como lo maneja el lenguaje de programación C. Sólo se puede usar para obtener un resultado y asignarlo a una variable. Se pueden usar operaciones cuando la condición sea verdadera o falsa, no se pueden ejecutar métodos a menos que éstos devuelvan un resultado.

En la siguiente figura se muestra su estructura, y la manera en la que se haría dicha operación con un if-else:

```
// Ternary operator
x > 0 ? y = 1 : y = 2

// Is same as
if (x > 0)
   y = 1
else
   y = 2
```

Figura 79. Operador Ternario comparado con un if-else

Se presenta un código para ambos lenguajes con el uso del operador ternario (Figura 80 y Figura 81).

```
1  package com.test.estruct
2
3  def a = 2, b = 5, r
4  r = a > b ? a : b
5  println "El mayor es $r"
```

Figura 80. Operador Ternario Groovy

```
1  package com.test.estruct;
2
3  public class Ternario2J {
4      public static void main(String[] args){
5          int a = 2, b = 5, r;
6          r = a > b ? a : b;
7          System.out.println( "El mayor es " + r);
8      }
9  }
```

Figura 81. Operador Ternario Java

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)

El operador Elvis es la combinación de un signo de interrogación y dos puntos, deja fuera el valor en entre ellos que tendría el operador ternario. La idea es que si la variable frente a la interrogante no es null, la utiliza. (Kousen, 2014)

```
1  package com.test.estruct
2
3  def a = "", b = "Fail", r, n1 = 3, n2=1, n3=0, r2, r3
4
5  r = a ?: b
6  r2 = n1 > n3 ?: n2
7  r3 = n1 < n3 ?: n2
8
9  println "$r"
10 println "$r2"
11 println "$r3"
```

Figura 82. Operador Elvis

En el código se evalúan tres expresiones, la primera evalúa si la variable *a* tiene algo para ser *true*, si es así imprime su contenido, o de lo contrario manda a imprimir lo que tiene *b*. La segunda y tercera comparan *n1* con *n3*, si *n1* es mayor que *n3* imprime *true*, o de lo contrario manda a imprimir lo que tiene *n2,* lo cual se muestra con la línea 7. Los resultados son los siguientes:

```
Fail
true
1
```

* 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 76, entonces ese valor se devuelve, si no, se utiliza el valor alternativo como se pudo ver en el último ejemplo presentado. (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*, lo restrictivo de manejar un switch en Java se menciona a continuación:

- Sólo permite un tipo de dato enumerable a evaluar, ya sea un byte, short, int o un char.
- El valor de un *case* debe ser el mismo tipo de la variable que llega al switch.

El siguiente ejemplo muestra lo anterior:

```
1  package com.test.estruct;
2
3  public class SwitchJ {
4      public static void main(String[] args){
5          char c = 'z';
6
7          switch(c){
8              case 'a': System.out.println("Char -> " + c);
9                  break;
10             case 'b': System.out.println("Char -> " + c);
11                 break;
12             case 'f': System.out.println("Char -> " + c);
13                 break;
14             default : System.out.println("No se encuentra -> " + c);
15         }
16     }
17 }
```

Figura 83. Switch Java

En el ejemplo anterior se observa que únicamente trata elementos de tipo *char*, tal como se define en el *switch* respecto al tipo de dato que se le manda.

Groovy, por el contrario, es más amigable con los tipos de datos dentro de los *case*. La siguiente figura lo demuestra:

```
1  package com.test.estruct
2
3  def x = 'pear'
4
5  switch(x){
6      case 1 : println "Entero recibido";
7      break;
8      case "switch" : println "String recibido";
9      break;
10     case 'd' : println "Char recibido";
11     break;
12     case ["apple", "pear", 1, 2, 3]: println "Elemento de la lista";
13     break;
14     case 1..5 : println "Elemento contenido en el intervalo";
15     break;
16     default: println "No coincide con ninguna de las anteriores"
17 }
```

Figura 84. Switch Groovy

El elemento que se pasa a *x* es un String, pero se encuentra dentro de la lista en el *case* de la línea 12 e imprime lo siguiente:

```
Elemento de la lista
```

Como se puede observar, es mucho más práctico un switch groovy, 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.

Se tiene la siguiente figura con el código Java:

```java
package com.test.estruct;

public class WhileJ {
    public static void main(String[] args){
        int x = 1;

        while(x<5){
            System.out.println(x);
            x++;
        }
    }
}
```

Figura 85. While Java

Su homólogo en script groovy es:

```groovy
package com.test.estruct

def x = 1

while(x<5){
    println "$x"
    x++
}
```

Figura 86. While Groovy

Groovy utiliza las reglas de la Figura 76 para el manejo de valores booleanos, pero tener cuidado de pasar sólo una variable y no tratarla dentro del bucle, esto para evitar un ciclo infinito, ya que únicamente con pasar un valor diferente de null o de 0 lo toma como true y entraría al bucle.

Se muestra un código de ejemplo:

```
1  package com.test.estruct
2
3  def x = 2
4
5  while(x){
6      println "En caso de ejecutarse sería un ciclo infinito"
7  }
```

Figura 87. while con valor true

En el código anterior, únicamente está tomando a *x* como true, entra al bucle pero como siempre será true y no tiene otra manera de modificar esto, será un ciclo infinito.

** Cabe destacar que Groovy no acepta el bucle do{} while(), desconoce totalmente que 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,…

Se muestran a continuación ejemplos para ambos lenguajes:

```java
package com.test.estruct;
import java.util.ArrayList;

public class ForJ {
    public static void main(String[] args){
        for (int i = 0; i < 5; i++){
            System.out.println(i);
        }

        ArrayList<String> lista = new ArrayList<String>();
        lista.add("Pedro");
        lista.add("Olga");
        lista.add("Miguel");

        for(String nombre : lista){
            System.out.println(nombre);
        }
    }
}
```

Figura 88. for y for-each Java

Código anterior pasado a script groovy:

```groovy
package com.test.estruct

// Clásico for
for (def i = 0; i < 5; i++){
    println "$i"
}

//For mejorado each
def lista = new ArrayList<String>();
lista.add("Pedro");
lista.add("Olga");
lista.add("Miguel");

for(String nombre : lista){
    println "$nombre"
}
```

Figura 89. for y for-each Groovy

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)

El siguiente código muestra algunos usos de este for:

```
1  package com.test.estruct
2
3  println "Itera sobre caracteres del String pasado"
4  def hello = "Hello"
5  for ( x in hello)
6     println x
7
8  println "\nRecorre elementos de la lisa"
9  def lista = new ArrayList<String>();
10 lista.add("Pedro");
11 lista.add("Olga");
12 lista.add("Miguel");
13
14 for(l in lista)
15     println "$l"
16
17
18 println "\nItera sobre un rango"
19 for(n in 1..3)
20     println "$n"
```

Figura 90. for-in

Al ejecutar el código anterior se obtiene el siguiente resultado:

```
Itera sobre caracteres del String pasado
H
e
l
l
o

Recorre elementos de la lisa
Pedro
Olga
Miguel

Itera sobre un rango
1
2
3
```

Terminada esta sección se han visto las ventajas que Groovy presenta para trabajar de una manera más práctica, resaltando que la idea no va con programación perezosa, sino más bien, con la eficiencia para mejorar tiempos y terminar con un código fácil de leer e interpretar por el desarrollador.

CAPÍTULO III. Desarrollos

Introducción

Groovy es un lenguaje que lleva pocos años en la vida de los programadores, pero debido a características similares a Java, y por lo que ofrece, se ha posicionado como un lenguaje con gran futuro como el mismo Java.

En lo que se centra este capítulo es presentar algunos datos respecto a herramientas que facilitan el empleo de ambos lenguajes por separado para el desarrollo de aplicaciones, demostrando cómo han ido tomando terreno en la vida de los programadores.

Además se hablará un poco respecto a las versiones que tiene cada uno. Los datos aquí presentados tienen la finalidad de brindar un panorama más amplio referente a lo que ofrece cada lenguaje.

Versiones

Java

Actualmente Java lanzó la versión 8 Update 25 (8u25) que salió a la luz en 14 de Octubre del 2014. Esta versión cuenta con las siguientes características.

- Datos IANA 2014c
  JDK 8u25 contiene datos de zona horaria IANA versión 2014c.
- Correcciones de bugs.
  Esta versión incluye correcciones para vulnerabilidades de seguridad.
  Un Patch de actualización crítica (CPU) es una colección de parches para múltiples vulnerabilidades de seguridad. Oracle ha recibido informes específicos de la explotación maliciosa de vulnerabilidades para el que Oracle ha lanzado ya correcciones.

La versión Update 20 (8u20) presentó las siguientes características:

- Cambios de instalador de Java.
- Cambios en el panel de control de Java.
- Compilador Java actualizado.
- Cambio en la versión de Java mínima necesaria para el plugin Java y Java Webstart.
- Cambios en las herramientas de empaquetado de Java.

Todo lo anterior resalta los cambios que afectan a los usuarios finales para las versiones de Java 8. (Oracle, 2014)

El 2013 fue un año crítico para Firefox debido a complementos que tiene implementados con Java, el motivo fue la falta de seguridad que se presentó, pero Java 8 en su última versión como se ha referenciado aquí, está al pendiente de mejorar el aspecto de seguridad. (Nabih, 2013)

Se tiene reportes sobre complementos de Firefox que han sido bloqueados debido a la falta de seguridad, estabilidad y problemas de rendimiento presentados. (ADD-ONS)

Java se utiliza para diversos entornos respecto a las necesidades.

De acuerdo con datos recabados por una empresa llamada ZeroTurnaround realizada a 2164 profesionales Java, se muestra la siguiente información respecto a las estadísticas del uso de versiones Java en el 2014. (White, 2014)

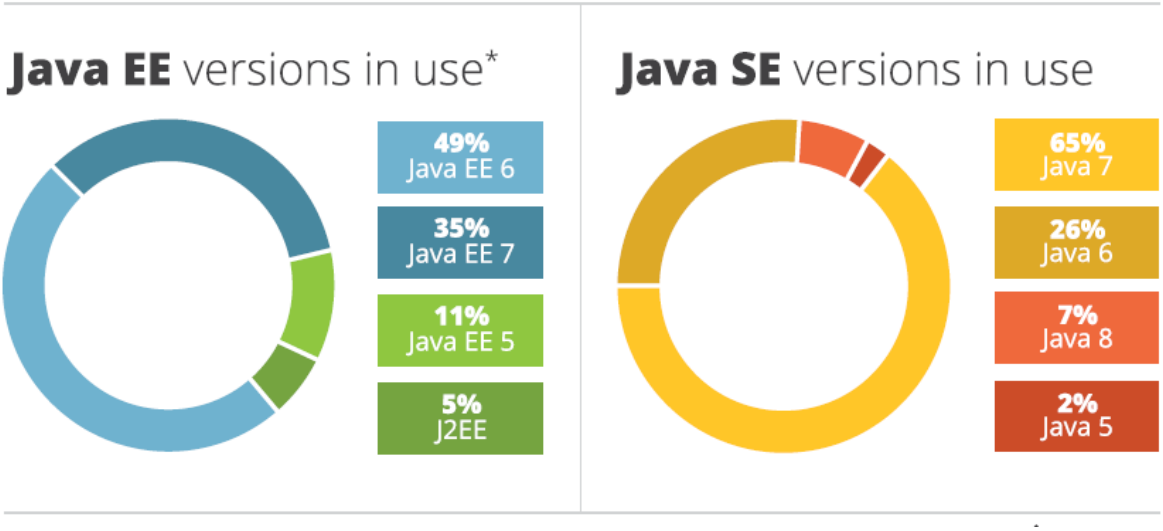

Figura 91. ¿Quién usa Java EE? (White, 2014)

Del total de esos profesionales encuestados, el 68% de los programadores usa Java EE(+SE), mientras que el 32% usa únicamente Java SE.

Java Platform, Standard Edition (Java SE) permite desarrollar y desplegar aplicaciones Java en equipos de escritorio y servidores, así como en los exigentes entornos integrados de hoy en día. Java ofrece la rica interfaz de usuario, rendimiento, versatilidad, portabilidad, y la seguridad que las aplicaciones actuales requieren. (Oracle)

Java Platform, Enterprise Edition (Java EE) es el estándar en software empresarial impulsado por la comunidad. Java EE se desarrolla utilizando la Java Community Process, con las aportaciones de expertos de la industria, las organizaciones comerciales y de código abierto, grupos de usuarios Java, y un sinnúmero de personas.

Hoy, Java EE ofrece una rica plataforma de software empresarial, y con más de 20 implementaciones compatibles de Java EE 6 para elegir, de bajo riesgo y muchas opciones más. (Oracle)

Java Platform, Micro Edition (Java ME) proporciona un entorno robusto y flexible para aplicaciones que se ejecutan en dispositivos embebidos y móviles en la Internet de las Cosas: micro-controladores, sensores, gateways, teléfonos móviles, asistentes digitales personales (PDA), impresoras y más. Java ME incluye interfaces flexibles de usuarios, seguridad robusta, una función de los protocolos de red y soporte para aplicaciones en red y fuera de línea que se pueden descargar de forma dinámica. Las aplicaciones basadas en Java ME son portátiles a través de muchos dispositivos. (Oracle)

Groovy

Actualmente se encuentra fuera la versión oficial 2.3 de Groovy, la cual presenta las siguientes características: (Groovy)

- Apoyo oficial para ejecutar Groovy en JDK 8.
- Nuevas y mejores transformaciones AST como @TailRecursive, @Builder y @Sortable.
- Nuevo módulo NIO2 con soporte Path.
- Aligeramiento JSON rápido de análisis y construcción.
- Nuevo motor de formato de plantilla.
- Groovysh y GroovyConsole facilitan el uso de mejoras.
- Nuevo test de utilidad GroovyAssert.
- Más capacidades de clase @BaseScript, y más.

Groovy tendrá soporte oficial para Android a partir de Groovy 2.4 (que ahora mismo está en beta) y necesita de ciertos plugins para hacer que Gradle[6] compile el código a código Android. (Martín, 2014)

[6] Gradle es la automatización de construcción evolucionada. Gradle puede automatizar la creación, prueba, publicación, distribución y más de paquetes de software u otro tipo de proyectos, tales como generación de sitios web estáticos, documentación generada o incluso cualquier otra cosa. http://www.gradle.org/

Herramientas y Tecnologías

Java

Se muestra la siguiente relación en porcentaje de las herramientas y tecnologías más utilizadas en 2014 por Java. (White, 2014)

*JUnit - 82.5%** - Marco superior de prueba utilizado por los desarrolladores.
Jenkins - 70%º - El mayor servidor CI utilizado en la industria.
*Git - 69%** - Tecnología de control # 1 versión ahí fuera.
Hibernate - 67.5%/º* - Framework superior ORM utilizado.
Java 7 - 65% - El líder de la industria para el desarrollo de SE.
Maven - 64% - La mayor herramienta de construcción usada en Java.
Nexus - 64%º - El repositorio principal utilizado por los desarrolladores.
MongoDB - 56%º - La tecnología NoSQL de elección.
FindBugs - 55%/º* - La mayor-utilizada herramienta de análisis de código estático en Java.
Tomcat - 50%º - El servidor de aplicaciones más popular en el mercado.
Java EE 6 - 49%º - Encontrado en la mayoría de los entornos empresariales Java.
Eclipse - 48% - El IDE más utilizado que cualquier otro.
Spring MVC - 40%/º* - El framework de desarrollo web más usado.
MySQL - 32%º - La tecnología SQL más popular.

Groovy

Por otro lado, Groovy cuenta con diversas herramientas que brindan soporte para comenzar a desarrollar, en las páginas siguientes se habla al respecto.

GVM (Administrador de entorno Groovy)

Para un comienzo rápido y sin esfuerzo en Mac OSX, Linux o Cygwin, puede utilizar GVM (el Administrador de entorno Groovy) para descargar y configurar cualquier versión Groovy de su elección.

GVM es una herramienta para la gestión de versiones paralelas de múltiples kits de desarrollo de software en la mayoría de los sistemas basados en Unix.

Proporciona una conveniente interfaz en línea de comandos para la instalación, el cambio, la eliminación y la lista de candidatos.

GVM se inspiró en las herramientas RVM[7] y rbenv, utilizadas en general por la comunidad Ruby. (Long)

Android Studio

Es un nuevo entorno de desarrollo de Android basado en IntelliJ IDEA.

Este IDE permite el desarrollo de aplicaciones Android con Groovy, actualmente está en versión beta, por lo cual puede haber fallos. No hay soporte oficial de Groovy para Android pero sólo es cuestión de tiempo para que Groovy sea acogido por Android. (Rodríguez Alemán, 2014)

De igual manera se puede utilizar para programar con Java.

Integración IDE

El lenguaje Groovy es apoyado por una gran cantidad de IDEs y editores de texto. (Groovy, 2014)

Editor	El resaltado de sintaxis	Autocompletado de texto	Refactorización
UltraEdit	Si	Basada en texto	No
Groovy Eclipse	Si	Si	Si
IntelliJ IDEA	Si	Si	Si
Netbeans	Si	Si	Si
Groovy and Grails Toolsuite	Si	Si	Si
Emacs Groovy Mode	Si	Soportes	No

[7] RVM es una herramienta de línea de comandos que permite instalar fácilmente, gestionar y trabajar con múltiples entornos de Ruby.

Editor	El resaltado de sintaxis	Autocompletado de texto	Refactorización
TextMate	Si	Snippets	No
vim	Si	No	No

Grails

Grails (desarrollado sobre el lenguaje de programación Groovy, el cual a su vez se basa en la plataforma Java) es un framework de código abierto para el desarrollo web, empaqueta las mejores prácticas tales como convención sobre configuración y las pruebas unitarias con lo mejor de los mejores entornos de aplicaciones de código abierto como Spring, Hibernate, y SiteMesh. Junto con la productividad del lenguaje de scripts Groovy, todo se ejecuta en la parte superior de las plataformas robustas Java y Java EE.

No sólo es un framework de código abierto Web para la plataforma Java, sino una plataforma de desarrollo completa también.

Grails incluye una configuración mínima y un ciclo de desarrollo más ágil. Grails elimina la mayor parte de los descriptores de configuración e implementación estándar MVC mediante el uso de convenciones de iniciativa. Además, debido a Grails se aprovecha de características del lenguaje dinámico de Groovy, por lo general es capaz de acortar el ciclo de desarrollo a sólo codificación, restauración, prueba y depuración. Esto ahorra valioso tiempo de desarrollo y hace el desarrollo mucho más ágil que con otros frameworks MVC Java o Java EE.

Características Grails

Grails realmente tiene demasiadas características para mencionarlas todas. En esta sección se van a destacar algunas de las más importantes.

- Convención sobre la configuración.
- Pruebas unitarias.

- Scaffolding.

Integrado de Código Abierto

Grails no padece del síndrome No Inventado Aquí (NIH). En vez de reinventar la rueda, él integra lo mejor de lo mejor de la industria estándar y frameworks de código abierto probadas. (Judd & Faisal Nusairat, 2008)

- Groovy
- Spring Framework
- Hibernate
- SiteMesh
- Frameworks Ajax
- Jetty
- HSQLDB
- JUnit

¿Quién usa Groovy?

Figura 92. Todos ellos usan Groovy (Groovy, 2014)

- Netflix - un tercio del tráfico de descarga en los EE.UU. proviene de Netflix, ese enorme tráfico pasa a través de varias capas de Groovy. Usa Groovy en su plataforma en la nube Asgard, además de la librería Glisten para interactuar con Amazon Simple Workflow Service.

- Google ha migrado el build de sus aplicaciones a Gradle que usa Groovy. Además, pronto se plantea la futura posibilidad de usar Groovy para crear aplicaciones Android.
- Linkedin ha desarrollado Glu, un sistema opensource para despliegue y monitorización, en Groovy, y además usa Grails para algunas de sus aplicaciones webs. (Rodríguez, 2013)

CAPÍTULO IV. La JVM

Introducción

El presente capítulo tiene como objetivo mostrar la manera de trabajar de ambos lenguajes respecto a la Máquina Virtual, uno de los puntos interesantes que tiene Groovy es que hace uso de ella al igual que 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, pero lo anterior será explicado con más detalle durante el desarrollo de este capítulo.

Se verá además la manera de ejecutar aplicaciones para cada uno de los lenguajes y todo lo que conlleva a ello.

¿Qué es la JVM?

La Especificación de la Java Virtual Machine define la JVM como:

> Una máquina imaginaria que se implementa mediante la emulación en el software de una máquina real. Código para la JVM se almacena en archivos *.class*, cada uno de los cuales contiene código como máximo una clase pública.

La Especificación de la Máquina Virtual Java proporciona las especificaciones de la plataforma de hardware para que se compile todo el código de la tecnología Java. Esta especificación permite al software Java ser independiente de la plataforma porque la compilación se realiza para una máquina genérica, conocida como la JVM. Puede emular esta máquina genérica en el software para ejecutarse en diferentes sistemas informáticos existentes o implementarse en hardware.

El compilador toma el código fuente de la aplicación Java y genera *bytecodes*. Bytecodes son instrucciones de código máquina para la JVM. Cada tecnología interprete de Java, independientemente de si se trata de una herramienta de desarrollo de la tecnología de Java o un explorador web que pueda ejecutar applets, tiene una implementación de la JVM.

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)

Compilador

Para traducir un programa escrito en un lenguaje de alto nivel (programa fuente) a lenguaje máquina se utiliza un programa llamado *compilador*. Este programa tomará como datos el programa escrito en lenguaje de alto nivel y dará como resultado el mismo programa pero escrito en lenguaje máquina, programa que ya puede ejecutar directa o indirectamente en el ordenador. (Ceballos, 2010)

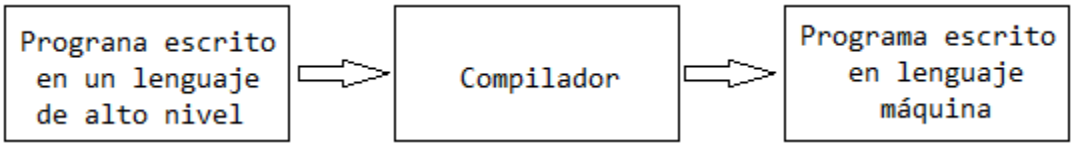

Figura 93. Compilación

Intérprete

A diferencia de un compilador, un intérprete no genera un programa escrito en lenguaje máquina a partir del programa fuente, sino que efectúa la traducción y ejecución simultáneamente para cada una de las sentencias del programa.

A diferencia de un compilador, un intérprete verifica cada línea del programa cuando se escribe, lo que facilita la puesta a punto del programa. La ejecución resulta más lenta ya que acarrea una traducción simultánea. (Ceballos, 2010)

Lenguajes Alternativos

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.

Groovy no fue la primera alternativa JVM de elección para los desarrolladores, pero el ambiente general es que Groovy también es algo a considerar - por lo menos para casi un tercio de los desarrolladores. Para muchos desarrolladores que utilizan la codificación en Java, el primer indicio de Groovy viene tratando Gradle, que es una herramienta de construcción DSL Groovy recibiendo una gran cantidad de atención en estos días.

Herramientas Groovy, y relacionados, como Grails, Gradle y Griffon, han impulsado en las mentes de casi 1 de cada 3 desarrolladores, según una encuesta reciente, y se prepara para, ya sea co-existir o desafiar al lenguaje Scala en el mundo del desarrollo de la empresa.

Figura 94. Los próximos lenguajes para la JVM (White & Maple, 2014)

El líder de proyectos Groovy, *Guillaume Geordi*, menciona lo siguiente respecto al éxito de este lenguaje:

"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)

Ejecución de aplicaciones

Todo código Groovy corre dentro de la JVM al igual que lo hace Java.

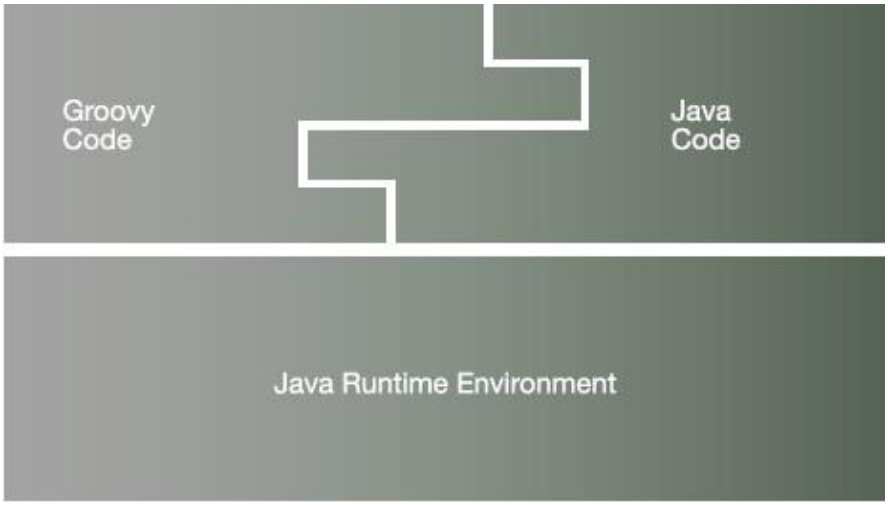

Figura 95. Java/Groovy dentro de la JVM (Egiluz, 2011)

El JRE contiene a la Máquina Virtual de Java donde se corren las aplicaciones.

Java

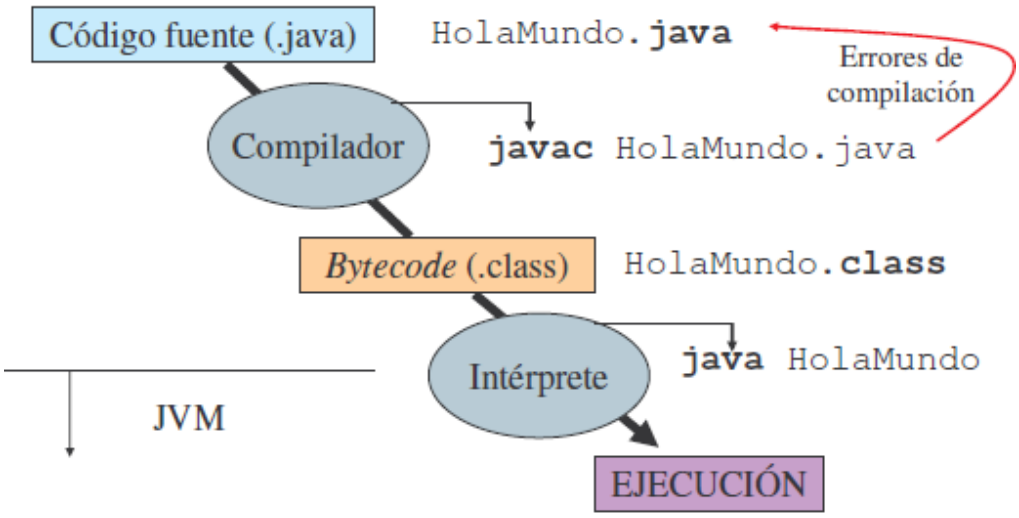

Figura 96. Ejecución de Java en la JVM (ESCET, 2009)

Java es un lenguaje compilado, antes de ejecutar las aplicaciones estas deben ser compiladas para detectar los errores que pudiese haber y posteriormente generar un archivo *.class*. Es necesario realizar el proceso de la Figura 96 con el archivo *.java* para poder ejecutar la aplicación.

Groovy

Groovy maneja 2 maneras para ejecutar sus programas, la Figura 97 muestra un diagrama en el que se ilustran ambos procesos.

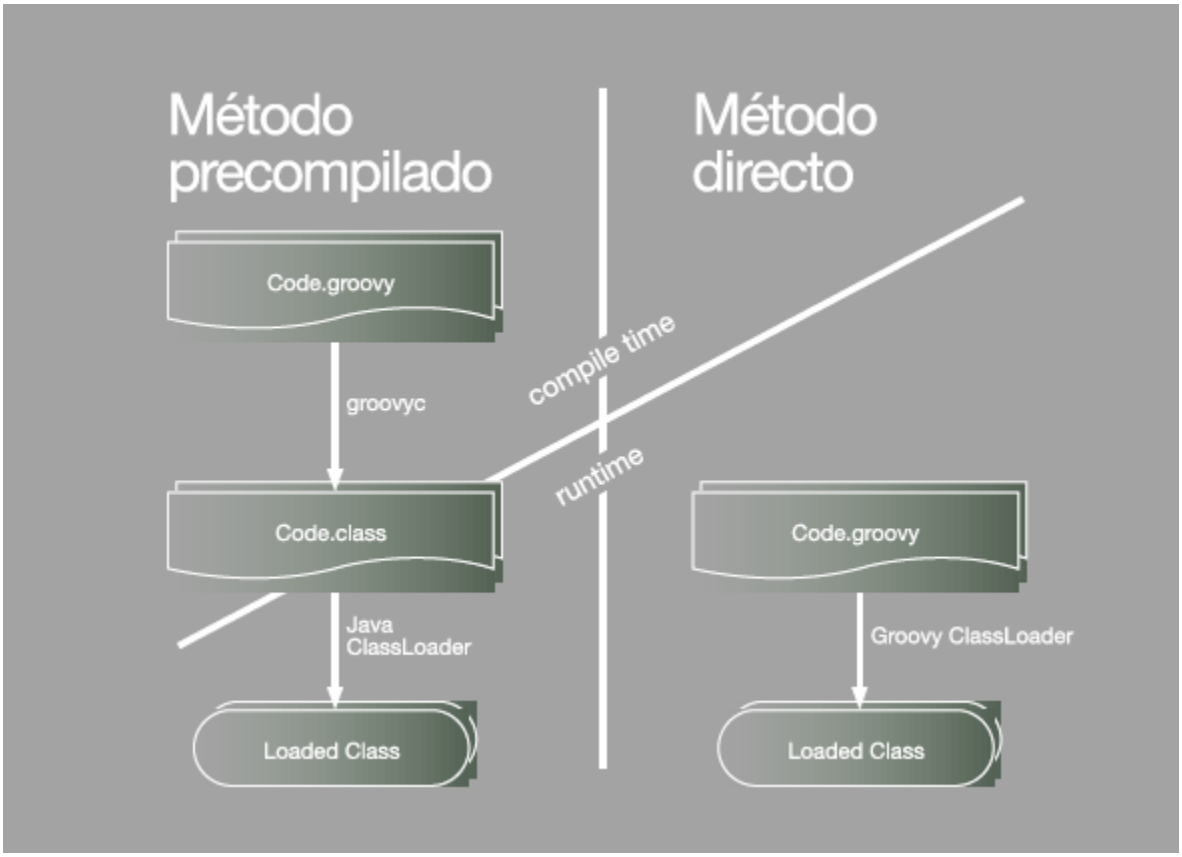

Figura 97. Formas de ejecutar código Groovy (Egiluz, 2011)

Se tienen dos opciones para ejecutar código Groovy. En primer lugar si se desea tener un enfoque más tradicional, como Java, respecto a la compilación explícita de código para crear código de bytes - el archivo *.class* - puede hacer eso usando línea de comandos y el comando *groovyc*.

En segundo lugar, es posible utilizar el comando *groovy* en el código fuente. Entonces Groovy automáticamente compila código en la memoria y lo ejecuta. No requiere realizar la compilación de manera explícita. (Subramaniam, 2013)

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)

CAPÍTULO V. Comparativa

Introducción

En este último capítulo se muestra una tabla donde se realiza una comparativa de ambos lenguajes bajo todos los criterios analizados en el desarrollo de este trabajo.

La información manejada es muy breve ya que sólo se centra en las características más sobresalientes para ambos lenguajes debido a que la explicación respectiva ya se ha realizado.

Los datos mostrados sirven para tener una idea más general sobre lo que ofrece y cómo trabaja cada lenguaje.

<table>
<tr><th>Criterio</th><th>Java</th><th>Groovy</th></tr>
<tr><td colspan="3">Programación Orientada a Objetos</td></tr>
<tr><td>Abstracción</td><td colspan="2">Concentra propiedades y comportamientos necesarios para la representación de un objeto.
Se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?" (Característica de caja negra).</td></tr>
<tr><td>Encapsulamiento</td><td>Utiliza modificadores de acceso para controlar la forma en que se utilizan los datos y métodos.
- private
- default
- protected
- public</td><td>Utiliza modificadores de acceso para controlar la forma en que se utilizan los datos y métodos.
- private
- protected
- public (modificador por default para clases, métodos y variables)</td></tr>
<tr><td>Modularidad</td><td colspan="2">La modularidad se organiza de forma lógica en clases y paquetes y de forma física mediante archivos.
Eclipse, el IDE utilizado para el desarrollo de este trabajo brinda esta propiedad.</td></tr>
<tr><td rowspan="2">Jerarquía</td><td colspan="2">La herencia proporciona la relación jerárquica.</td></tr>
<tr><td>Herencia:
- Sólo soporta herencia simple.
- Hace una simulación de herencia múltiple con el uso de interfaces.</td><td>Herencia:
- Soporta herencia simple.
- Para el manejo de herencia múltiple hace uso de la anotación @Mixin, lo que permite que se utilice una jerarquía de clases.</td></tr>
</table>

Tipos (Tipificación)	Tiene comprobación estricta de tipos (errores detectados en tiempo de compilación). Realiza ligadura dinámica, permitiendo el polimorfismo.	Tiene comprobación débil de tipos (errores detectados en tiempo de ejecución). Realiza ligadura dinámica, permitiendo el polimorfismo.
	Tipado estático - Define ocho tipos *primitivos*: *byte, short, int, long, char, float, double* y *Boolean.* - Obliga a asignar algún tipo de dato a la variable.	Tipado dinámico - Puede utilizar los tipos primitivos de Java. - Utiliza la palabra reservada *def* para definir una variable dinámica.
	Orientado a Objetos.	Orientado a Objetos Puro. - En Groovy todo es un Objeto - Aunque permite declarar primitivos de Java, Groovy hace una referencia a una instancia de la clase wrapper apropiada.
	Polimorfismo - Permite sobrecarga y sobreescritura de métodos. - Necesita de la notación @Override para sobrescribir un método.	Polimorfismo - Permite sobrecarga y sobreescritura de métodos. - Necesita la propiedad *MetaClass y el operador "=" para sobrescribir un método*. - Para sobrecargar un método se utiliza la anotación leftShift "<<". Utiliza la misma

<table>
<tr><td></td><td></td><td>anotación para agregar métodos.</td></tr>
<tr><td>Concurrencia</td><td>Permite manejo de hilos para lograr la representación de procesos.
Necesita extender e implementar Thread y Runnable respectivamente además de utilizar los métodos run() y start().

class MyThread extends Thread {
public void run() { }
}

class MyRunnabl implements Runnable {
public void run() { }
}</td><td>Permite manejo de hilos para lograr la representación de procesos.
Utiliza closures para trabajar.

Hace uso de métodos estáticos en la clase Thread:

Thread.start { /* Closure body */ }

Closure es implementado en Runnable.

t = new Thread() { /* Closure body */ }
t.start()</td></tr>
<tr><td>Persistencia</td><td colspan="2">Es la propiedad de un objeto a través del cual su existencia trasciende en el tiempo y/o espacio.
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.
Se utiliza la interfaz Serializable para marcar las clases cuyas instancias pueden ser convertidas a</td></tr>
</table>

	secuencias de bytes y así lograr la persistencia.	
Sintaxis		
Clases	Formato tipo: - Clases	Formato tipo: - Clases - Script
	- Puntos y comas obligatorios. - Paréntesis obligatorios. - Las clases son *default* o *public.*	- Puntos y comas opcionales. - Paréntesis opcionales. - Todas las clases por default son *public.*
	Manipulación de texto: - Strings “comillas dobles” - Concatenación Símbolo + - String multilínea Utiliza el operador de concatenación,	Manipulación de texto: - Strings “comillas dobles” ‘comillas simples’ /barra/ - Concatenación Símbolo + - Interpolación Strings definidos entre comillas dobles o barras. Utiliza ${} o $ para incrustar las variables - String multilínea Se define mediante el uso de tres comillas

	símbolo +.	dobles o tres comillas simples.
	Importaciones: Importa automáticamente el paquete java.lang.	Importaciones: Automáticamente se importan los siguientes paquetes: - java.lang.* - java.util.* - java.net.* - java.io.* - java.math.BigInteger - java.math.BigDecimal - groovy.lang.* - groovy.util.*
	Constructores: - Es necesario que cada constructor sobreescrito que se llame se defina en la clase. - Llamada: Class var = new Class(String v1, int v2);	Constructores: - Es posible omitir la definición de un constructor. - Llamada: *Class var = new Class(v1, v2)* *def var = new Class(v1,v2)* *def var = new [v1, v2] as Class*

		Class var = [v1, v2] - Parámetros posicionales, se manda a llamar el constructor adecuado respetando el orden de las variables en la declaración del mismo. *def var = new Class(nombre, edad)* - Parámetros nombrados, útiles para clases inmutables con parámetros opcionales. Permiten tener 2 o más constructores con un solo argumento del mismo tipo. `Datos cn = new Datos(nombre: "Lau")` `Datos ca = [apellido: "Durán"] as Datos` - Es posible pasar un map a un constructor: `map = [id: 1, name: "Barney, Rubble"]` `customer1 = new Customer( map )`
Métodos	Declaración: `public String comer(String alimento){` `  return "le gusta comer" + alimento;` `}`	Declaración: `def comer(alimento){` `  "le gusta comer ${alimento}"` `}` - Return opcional. - Nivel de acceso por default es public. - No requiere declarar tipo de retorno. - Variables aceptadas de tipo dinámico.

		Agregar un método: Es posible agregar un método desde un script con la anotación “<<”, `AutoG44.metaClass.correr << {->` `  println "El Auto corre"` `}`
	Propiedades y GroovyBeans: - Requiere implementar los *getters* y *setters* para cada variable a utilizar.	Propiedades y GroovyBeans: - Implementa implícitamente los getters y setters de las variables. - Cuando se utiliza el *set y get* de manera explícita y la variable correspondiente se marca explícitamente con un modificador de acceso, es necesario implementar el getter y setter correspondiente.
	Método principal: *public static void main(String args[]) { }*	Método principal: *static main (args){ }* - public por default - debido a que los tipos de retorno no se utilizan para el envío, es posible omitir la declaración *void.* - *args* es ahora implícitamente de tipo estático *java.lang.Object.*

<table>
<tr>
<td rowspan="2"></td>
<td>Parámetros:
- Declaración estricta de tipo de dato a recibir.

Parámetros opcionales
- Se permite lograr con un ArrayList pasar n cantidad de valores de diferente tipo.


```
ArrayList l2 = new ArrayList();
l2.add(3);
l2.add("a");
l2.add(6);
l2.add("8");
l2.add("R");

pased2(ArrayList l2){ }
```


- Permite pasar argumentos variables de manera más sencilla pero con la desventaja de que no admite diferentes tipos:
nombreMetodo(String a, int b, int…x){}</td>
<td>Parámetros:
- No necesita declarar tipo de dato a recibir.

Parámetros opcionales
- Object[] –> se encarga de recibir n cantidad de valores pasados.
Permite pasar diferentes tipos de datos.


```
pased(1, 2, 3, 4, 5, "Hello!", 6, 7, 8, 9, 10)

pased(a, Object[] optionals){ }
```

</td>
</tr>
<tr>
<td></td>
<td>Closures
- Similar a un método.</td>
</tr>
</table>

		- Se permite pasar parámetros a ellos. - Se pueden pasar como parámetros. - Pueden devolver un valor. - Se diferencia de un método ya que no está pegado a un objeto, es anónimo.
Manejo de Excepciones	Java obliga a manejar Excepciones en caso de requerirlas. Un ejemplo es a la hora de tratar de leer el contenido de un archivo.	Manejo opcional, en caso de no manejarlas Groovy se encarga de ello. Al realizar código para leer un archivo no necesita tratar la excepción, en caso de que ocurra algo inesperado, Groovy lanza la excepción adecuada.
Archivos	- Requiere importar el paquete java.io. - Necesita manejo de excepciones. - Requiere las siguientes clases y sus métodos respectivos. o *BufferedWriter* o *BufferedReader* o *FileReader* o *FileWriter*	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. - No se necesita hacer el manejo de excepciones. - El paquete *java.io* es importado automáticamente por el GDK - No requiere el manejo de clases como *BufferedReader,FileReader, entre otras*.

		Unicamente requiere la clase File. - Hace uso de closures.
	Métodos: - *readLine* – itera sobre las líneas de un archivo - *flush* - necesario para garantizar que el último de los datos va a estar realmente en el archivo. - *write* – requerido para escribir un flujo de caracteres.	Métodos: - *eachLine* – itera sobre las líneas de un archivo. - *newPrintWriter* - crea un nuevo *PrintWriter* para el archivo pasado.
Palabras reservadas	Palabras propias del lenguaje.	Hace uso de las palabras reservadas de Java y además implementa algunas más para hacer al lenguaje más sencillo. - def- define variables, métodos - as - permite cambiar el tipo de objeto - *in* - se utiliza en los bucles para especificar el rango de un bucle - it - se refiere al parámetro de un closure
Estructuras de control	If/ if-else - Requiere de operadores entre las variables que le llegan para evaluar y	If/ if-else - Puede evaluar una sola variable, ya que sólo evalúa que sea diferente de 0 o de

	poder hacer operaciones dentro del ciclo. - Requiere de 2 variables mínimo a evaluar. *if((b == "df") && (c!= a)){}*	null. *if(y)* - Una asignación se toma como true y entra al ciclo. *if((n=y)){}*
	Operador ternario - Trabaja como un if-else. - *(a ? a : b)* - Evalúa la primer condición, si es true realiza la operación después del signo de interrogación, o dado lo contrario realiza la última sentencia (b).	Operador ternario y Elvis - Utiliza el operador ternario de la misma forma que Java. *Operador Elvis (a ?: b)* - Su equivalente ternario es *(a ? a : b)* - La idea es que si la variable frente a la interrogante no es null, la utiliza.
	Switch - Sólo permite un tipo de dato enumerable a evaluar, ya sea un char, int, etc. - El valor de un *case* debe ser el mismo tipo de la variable que llega al *switch.*	Switch - Los elementos evaluados en el *case* no es necesario que sean del mismo tipo pasado al *switch.* - Puede manejar al mismo tiempo en los diversos *case* enteros, listas, cadenas, rangos.

	While `while(x<5){ }`	While - Utiliza la misma lógica que Java. - Puede evaluar una sola variable, ya que sólo evalúa que sea diferente de 0 o de null.
	For - *for (int i = 0; i < 5; i++) { ... }* - *for-each* Utilizado para iterar sobre los elementos de colecciones.	For Utiliza el clásico for y for-each de Java, pero añade un par de ciclos para iterar - *each* Método que toma un cierre como argumento para colecciones. Es la construcción del bucle más común en Groovy. *(0..5).each { println it }* - *for-in* Se puede utilizar la expresión "*in*" para repetir cualquier tipo de colección. Itera sobre un rango. Itera sobre los caracteres de un String pasado. `for(l in lista)`

	Do-while	No reconoce el bucle *do{} while()*
Desarrollos		
Versiones	Java lanzó la versión 8 Update 25 (8u25) que salió a la luz en 14 de Octubre del 2014. Aplicaciones para equipos de escritorio y servidores. Software empresarial. Aplicaciones para dispositivos móviles.	Actualmente se encuentra fuera la versión oficial 2.3 de Groovy. Groovy tendrá soporte oficial para Android a partir de Groovy 2.4 (que ahora mismo está en beta). Software empresarial. Aplicaciones web.
Herramientas y tecnologías	- *Spring MVC* - El framework de desarrollo web más usado. - *JUnit* - Marco superior de prueba utilizado por los desarrolladores. - *Hibernate* - Framework superior ORM utilizado. - *Jenkins* - El mayor servidor CI utilizado en la industria. - *Git* - Tecnología de control # 1 versión ahí fuera. - *Java 7* - El líder de la industria para el desarrollo de SE.	*Android Studio* - Es un nuevo entorno de desarrollo de Android basado en IntelliJ IDEA. - Permite el desarrollo de aplicaciones Android con Groovy, actualmente está en versión beta. Integración IDE - *Eclipse* - *NetBeans* - UltraEdit - IntelliJ IDEA - Groovy and Grails Toolsuite

	- *Maven* - La mayor herramienta de construcción usada en Java. - *Nexus* - El repositorio principal utilizado por los desarrolladores. - *MongoDB* - La tecnología NoSQL de elección. - *FindBugs* - La mayor-utilizada herramienta de análisis de código estático en Java. - *Tomcat* - El servidor de aplicaciones más popular en el mercado. - *Java EE 6* - Encontrado en la mayoría de los entornos empresariales Java. - *Eclipse* - El IDE más utilizado que cualquier otro. - *MySQL* - La tecnología SQL más popular.	- Emacs Groovy Mode
	Grails Desarrollado sobre el lenguaje de programación Groovy, el cual a su vez se basa en la plataforma Java. Es un framework de código abierto para el desarrollo web, integra lo mejor de lo mejor de la industria estándar y frameworks de código abierto probadas. - Groovy	

	- Spring Framework - Hibernate - SiteMesh - Frameworks Ajax - Jetty - HSQLDB JUnit	
Java Virtual Machine		
Lenguaje compilado o interpretado	Java es un lenguaje compilado e interpretado. Necesita de ambos pasos para ejecutar correctamente.	Groovy es un lenguaje interpretado puesto que no es necesario compilar el código antes de ser ejecutado, pero también se le da la oportunidad de compilar primero.
Lenguajes alternativos	Java es el lenguaje principal que se ejecuta sobre la Máquina Virtual.	Groovy es uno de los lenguajes recomendables que se encuentran no muy por debajo de Java para trabajar en la JVM.
Ejecución	Cuando se ejecuta vía línea de comandos, las clases Java deben ser compiladas con el comando *javac* para detectar los errores que pudiese haber y así generar un archivo *.class*, para posteriormente ejecutar ese archivo con el comando *java.*	Groovy maneja 2 maneras de ejecutar un archivo vía línea de comandos: - Método precompilado: Compila las clases con el comando *groovyc*, se generan archivos .class como en Java.

		Posteriormente se ejecutan las clases con el comando *groovy.* - Método directo: Únicamente utiliza el comando *groovy* para ejecutar las clases. Cuando se utiliza un IDE, cada vez que guarde un script o clase Groovy, esta se compila.

CONCLUSIONES

En la actualidad el empleo de lenguajes de programación se ha hecho de suma importancia para la vida diaria de las personas, de manera inconsciente la gran mayoría hace uso de ellos, ya sea en celulares, tablets, computadoras y gran número de dispositivos que se encuentran al alcance de los usuarios emplean diversos sistemas.

En el trabajo desarrollado se ha visto la manera de trabajar de ambos lenguajes, uno de ellos con gran renombre ya entre los programadores y el otro por su parte, está incursionando poco a poco en la vida de ellos, prometiendo ser apto para ayudar a satisfacer las necesidades que se tienen.

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 con 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 éste 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, sencillamente 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 los 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 corren sobre la Máquina Virtual, esta 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 estos 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 este 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 ver 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.

ÍNDICE DE FIGURAS

BIBLIOGRAFÍA

ADD-ONS. (n.d.). *ADD-ONS*. Retrieved Octubre 30, 2014, from Blocked Add-ons: https://addons-dev.allizom.org/en-US/firefox/blocked/

Alegsa, L. (n.d.). *Definición de script*. Retrieved Septiembre 2014, 2014, from Alegsa: http://www.alegsa.com.ar/Dic/script.php

Álvarez, M. A. (2014, Octubre 5). *http://www.desarrolloweb.com.* Retrieved Octubre 22, 2014, from http://www.desarrolloweb.com: http://www.desarrolloweb.com/manuales/teoria-programacion-orientada-objetos.html

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

García Beltrán, A., & Arranz, J. M. (n.d.). *Constructores.* Retrieved Septiembre 23, 2014, from Programación orientada a objetos con Java: http://ocw.upm.es/lenguajes-y-sistemas-informaticos/programacion-en-java-i/Contenidos/LecturaObligatoria/13-constructores.pdf

Groovy. (2014). *IDE integration*. Retrieved Noviembre 4, 2014, from Groovy: http://groovy-lang.org/ides.html

Groovy. (2014). *They all use Groovy!* Retrieved Noviembre 4, 2014, from Groovy: http://groovy-lang.org/

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

Groovy. (n.d.). *Experience Groovy 2.3*. Retrieved Noviembre 24, 2014, from A dynamic language for the Java platform: http://groovy.codehaus.org/

Groovy. (n.d.). *Strings and GString*. Retrieved Septiembre 18, 2014, from Documentation: http://groovy.codehaus.org/Strings+and+GString

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.

Long, J. (n.d.). *GVM the Groovy enVironment Manager*. Retrieved Noviembre 4, 2014, from GVM: http://gvmtool.net/

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

Martín, J. (2014, Agosto 13). *La pregunta del trillón de dolares: ¿sirve para Android?* Retrieved Noviembre 23, 2014, from Groovy, o por qué Java está viejuno: http://blog.arasthel.com/groovy-o-por-que-java-esta-viejuno/

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

Nabih, S. (2013, Octubre 23). *Firefox bloquea Java: una apuesta arriesgada por la seguridad*. Retrieved Octubre 29, 2014, from LA GUÍA DE SOFTWARE MÁS COMPLETA DEL MUNDO: http://noticias.softonic.com/firefox-le-cierra-la-puerta-a-java-una-apuesta-arriesgada-por-la-seguridad

Oracle. (2014). *Características principales de la versión Java 8*. Retrieved Noviembre 13, 2014, from Java: https://www.java.com/es/download/faq/release_changes.xml

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.). *Constructors.* Retrieved Septiembre 23, 2014, from Java Documentation: http://docs.oracle.com/javase/tutorial/reflect/member/ctor.html

Oracle. (n.d.). *Defining Methods*. Retrieved Septiembre 25, 2014, from Java Documentation: http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html

Oracle. (n.d.). *Java.* Retrieved Octubre 16, 2014, from Java: https://www.java.com/es/download/faq/develop.xml

Oracle. (n.d.). *Java.* Retrieved Octubre 16, 2014, from Java: https://www.java.com/es/download/faq/whatis_java.xml

Oracle. (n.d.). *Java EE at a Glance*. Retrieved Noviembre 2014, 24, from Oracle Technology Network: http://www.oracle.com/technetwork/java/javaee/overview/index.html

Oracle. (n.d.). *Java Micro Edition (Java ME)*. Retrieved Noviembre 24, 2014, from Oracle Tecnología de red: http://www.oracle.com/technetwork/java/embedded/javame/index.html

Oracle. (n.d.). *Java SE at a Glance*. Retrieved Noviembre 24, 2014, from Oracle Technology Network: http://www.oracle.com/technetwork/java/javase/overview/index.html

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

Oracle. (n.d.). *The Catch or Specify Requirement*. Retrieved Septiembre 27, 2014, from Java Documentation: http://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

Oracle. (n.d.). *Variables*. Retrieved Septiembre 21, 2014, from Java Documentation: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

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/

Pérez, G. G. (2008). *Aprendiendo Java y Programación Orientada a Objetos.*

Rodríguez Alemán, J. (2014, Octubre 10). *Usar Groovy para desarrollar en Android*. Retrieved Noviembre 4, 2014, from El arte del programador: http://elartedelprogramador.com/usar-groovy-para-desarrollar-en-android/

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

Rodríguez, T. (2013, Octubre 16). *Crece la popularidad de Groovy, entra en el ranking TIOBE dentro del top 20*. Retrieved Octubre 21, 2014, from Genveta

Dev: http://www.genbetadev.com/java-j2ee/crece-la-popularidad-de-groovy-entra-en-la-lista-tiobe-dentro-del-top-20

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. (2014). *Java Tools & Technologies Landscape for 2014.* Retrieved Octubre 30, 2014, from ZeroTurnaround: http://pages.zeroturnaround.com/Java-Tools-Technologies.html?utm_source=Java%20Tools%20&%20Technologies%202014&utm_medium=reportDL&utm_campaign=kick-ass-tech&utm_rebellabsid=88

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