Comparativa sintáctica entre los lenguajes de programación java y groovy

Syntactic comparative between Java and Groovy programming languages

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

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

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

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

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

Resumen

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

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

Abstract

One of the languages that takes several years of life, and remains as one of the most important due to their different characteristics that allow for the creation of software applications, is the Java language. Java is a language that allows development for mobile devices, desktop applications, corporate as well as for the web environment. Development of programming languages on the other hand, maintains a great dynamism. In 2003 the Groovy programming language, this language retains a syntax familiar to Java but with particular characteristics. Both Groovy and Java are object-oriented languages and running on a Virtual machine. The intention of this written is to perform a syntactic comparison of Java and Groovy languages to observe the particularities of each one, and in this way, facilitate the developers projects implementation.

Keywords: object-oriented programming, modularity, methods, polymorphism, encapsulation, hierarchy, classification, concurrency, persistence.

Fecha recepción: Enero 2015 **Fecha aceptación:** Julio 2015

Introduction

Programming languages have been around for some decades. Each new language provides new tools for everyday life. One of the languages that has years of life and has managed to remain as one of the most important is the language Java, Java was developed by Sun Mycrosystems in 1991, with the main idea of creating a language to unify electronic equipment of domestic consumption. Initially it was named Oak, but in 1995 the name was changed to Java and from 2009 is owned by Oracle Corporation.

On the other hand Groovy, created in 2003 by James Strachan and Bob McWhirter. Groovy is based on languages Smaltalk, Python and Ruby, but also said that it is based on Perl. Groovy

preserves a familiar syntax to Java, making in this way that the programmers working Java easier to become familiar with Groovy.

Since these two languages are considered Orientados to objects, to make the comparison was selected as a base of Grady Booch definition about the concept of object-oriented programming:

> *Object-oriented programming is a deployment model in which programs are organized as cooperatives collections of objects, each of which represents an instance of any class, and whose classes are, all of them members of a hierarchy of classes linked by relations of inheritance*

The object-oriented languages must meet four key elements of this model:

• Abstraction

• Encapsulation

• Modularity

• Hierarchy

To say fundamental, to say that a language without any of these elements is not object-oriented.

There are three children of the object-oriented model:

• Types (typing)

• Concurrency

• Persistence

By side it means that each is a useful part in the object-oriented language, but not essential (Booch, 1991).

By the extent of the Java language, only these points were chosen to compare, so throughout this paper will attempt to make a comparison of these languages under the scheme referred to above, including, of course, the analysis of operators control and use of the virtual machine

COMPARISON

A. MODULARITY

A class comes to represent the definition of a program module, which in turn define common methods and attributes.

An object is an instance of the class, this instance contains attributes and methods with specific values of your data. (& Prieto Ramos Rodríguez Echeverría, 2004).

A method is written inside a class and determines how it should act in order when you receive the message associated with that method. In turn, a method can send messages to other objects requesting action or information. The attributes defined in the class allow to store information for the object. (Ceballos, 2010).

When thinking is modeled objects, you must take the characteristics and properties of a real body, and bring it to an object. The term refers to the emphasis on "what does?" rather than the "how?" (Feature black box) (Di Serio, 2011).

Modularity in Java and Groovy logically organized into classes and packages and fitness through files.

Classes:

- Encapsulate the attributes and methods of an object type in one compartment.
- Conceal, using access specifiers, internal elements not intended for publication abroad.

Packages:

• Logical drives are grouping classes.

  - or public classes are part of the package interface and are visible outside.
  - or classes that are not public are only visible within the package itself.

Files:

- Among the files may reside several classes with certain restrictions that will be seen later. (TutorialesNET, 2014)

The management module allows a better program structure reduces the complexity thereof, creates a series of well defined boundaries which increases their understanding.

Java is desirable that each class is placed in a file. The filename is the name of the class, if you have more than one class in a file, the file name will be the class that has the main method.

There can be only one public class per file, the name must match the public class. There may be more than one default class in the same file.

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. (Oracle, 2014).

This mechanism is called CamelCase name (for example: ClassName, CuentaUsuario, Invoice).

Groovy, compared to Java, it allows scripts. Scripts are programs, usually small or simple, usually to perform very specific tasks. They are usually a set of instructions stored in a text file that must be interpreted line by line in real time for execution; It distinguishes this program (compiled), as these must be converted to an executable binary file (eg executable .exe, etc.) to run them. (Alegsa)

What is the same, no need to put all the Groovy code in a class. So in Groovy, if the code to implement is a few lines, you can run a simple script, creating the class come according to the size of the program, at the time they are needed most variables, instances, etc. ., that is when conducting a class (Kousen, 2014, p. 19) is required.

Two additional differences noted in the groovy handled on Java syntax:

- *The semicolons are optional.*
- *Parentheses are often optional. It is wrong not to include if desired*.

For Groovy nomenclature it depends on some features based on management classes, scripts or both in one file. Then the foundation is to conduct this part:

File for Groovy class relationship

The relationship between files and class declarations is not as fixed as in Java. Groovy files may contain any number of statements of public classes according to the following rules:

- □ If a Groovy file contains the class declaration, it handles like a script; ie that wraps transparently in a class of type Script. This automatically generated class has the same name as the name of the script source file (without the extension). The file content is wrapped in a run method, and an additional main method is easily constructed from the script.

- If a file contains exactly one Groovy class declaration with the same name as the file (without extension), then it is the same ratio of one-to-one in Java.
- A Groovy file can contain multiple class statements of any visibility, and there is no rule imposed that any of them should match the file name. The compiler happily groovyc creates .class files for all classes declared in that file.
- A file can mix Groovy class declarations and script code. In this case, the script code will become the main class to be executed (König & Glover, 2007, pp. 188-189).
- If the file contains a class with a script, and class corresponds exactly to the filename, the compiler Groovy politely suggests rename either script or the name of the class, because it can not generate all files required classes.

As a rule, it only suggests use this mixed mode coding when you are writing separate scripts. Groovy to write code that is integrated into a larger application, it is best to stick with the Java way of doing things, so that the source file names corresponding to the names of the classes that are implemented as part of your application. (Dearle, 2010, pp. 34-35). An important point is that like Java, the file name is required to start with a capital letter.

A. METHODS

Classes consist of instance variables and methods. The concept is very broad method because Java gives them great power and flexibility.

In Java the only required elements of a method declaration are the method return type, name, a pair of parentheses "()", and a body between braces "{}".

More generally, method declarations have the following components:

- Modifiers - public, private, and others.
- The return type - data type of the value returned by the method, or void if the method does not return a value.
- The method name.
- Parameter List brackets a comma-delimited list of input parameters, preceded by their data types, enclosed in parentheses (). If there are no parameters, you must use empty parentheses.
- The method body, enclosed in braces-method code, including the declaration of local variables.
- Modifiers, return types and parameters. (Oracle)

Regarding the usual Groovy Java modifiers can be used; declare a return type is optional; and if no return type modifiers or supplied, the keyword def fills the hole. When def keyword is used, the return type is considered untyped. In this case, under the sheets, the return type is java.lang.Object. The default visibility is public methods. (König & Glover, 2007, p. 180)

You should use the keyword def to define a method that has a return type of dynamic.

In return Groovy word is not used as it does Java to return a value or object reference. Groovy by default always returns the last line of the body, so it is not necessary to explicitly specify it as in Java. You can explicitly handle without any problems, but Groovy gives comfort omit it.

In Java, the main method is as follows:

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

In this line starts executing the program. All Java programs start running with the call to main () method.

The interpreter or Java virtual machine calls main () before any object is created. The keyword void simply tells the compiler that main () returns no value. (Schildt, 2009, p. 23)

In the main () method there is only one parameter, although complicated. String args [] declares a parameter named args, which is an array of instances of the String class (arrays are collections of similar objects). String objects stored strings. In this case, args receives the arguments that are present on the command line when the program is run.

The last character of the line is {. This character signals the start of the main () method body. All included in a method code must be between the opening brace of the method and the corresponding stopcock. The main () method is simply a starting place for the program. A complex program can have a lot of classes, but it is only necessary that one has the main () method to start the program. (Schildt, 2009, p. 24)

Groovy handles the main method in a simpler way. Its form is:

static main (args){ }

The main method has some interesting touches. First, the public modifier can be omitted, since the value is predetermined. Second, args usually has to be of type String [] in order to make the main method to start the execution of the class. Thanks to the delivery of Groovy, work

anyway, but args is now implicitly static type java.lang.Object. Third, because the return types not used for shipping, you can omit the void declaration. (König & Glover, 2007, p. 180)

Settings

Considering Figure 1, we can see that Groovy need not declare the data type with the variable to get, whereas in Java must be defined strictly.

Groovy Code:

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

Figure 1. Parameter no variable

Figure 2 shows how the Java:

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

Figure 2. Java Code parameter type required

In the Groovy code it is only necessary to declare the variable to get. Not forgetting to emphasize the dynamic, in the example can run both an integer as a string without any problems.

A. Polymorphism

Polymorphism is the way to invoke an action to have different behaviors depending on the context in which it is used. More simple, "polymorphism is the possibility that a method (function) may have different behavior from the parameters that are sent (overload), or from the way that is invoked (on writing)". (Hdeleon, 2014)

Any object that passes the IS-A test can be polymorphic.

Method overloading

It is a feature that makes programs more readable. It is to re-declare a method already declared with a different number and / or type of parameters. An overloaded method can not only differ

in the type of result, but should also differ in the type and / or number of formal parameters. (Ceballos, 2010)

Builders

To create an object the keyword "new" is required, for example:

variable = new *nombre_de_clase();*

It is more evident the need for parentheses after the class name. What really happens is that it is calling the constructor of the class.

A constructor is used to create an object that is an instance of a class. Normally it carried out the necessary to initialize the object before the methods are invoked or accessed operations fields. The builders never inherited. (Oracle)

The constructor has the same name as the class in which it resides and syntactically similar to a method. Once defined, the constructor is called automatically after the object is created and before the end of the new operator. The builders are a little different to conventional methods because they do not return any type, not even void.

When not explicitly define a class constructor, Java creates a default class constructor.

For simple classes, it is sufficient to use the default constructor, but not for more sophisticated classes. Having defined the builder himself, the default constructor is no longer used. (Schildt, 2009, pp. 117-119). The above is referenced to Java and Groovy.

The declaration of a different default constructor builder, requires being assigned the same identifier as the class and is not explicitly stated a type of return value. Whether or not parameter is optional. Furthermore, the overhead allows various builders that may be designated (with the same identifier as the class), provided they have a type and / or number of various parameters. (Garcia Beltran & Arranz)

In groovy, like methods, the default constructor is public. You can call the constructor in three different ways: as normal Java (normal use builder Figure 3), the type of restriction enforced by using the keyword as, and type of implicit constraint.

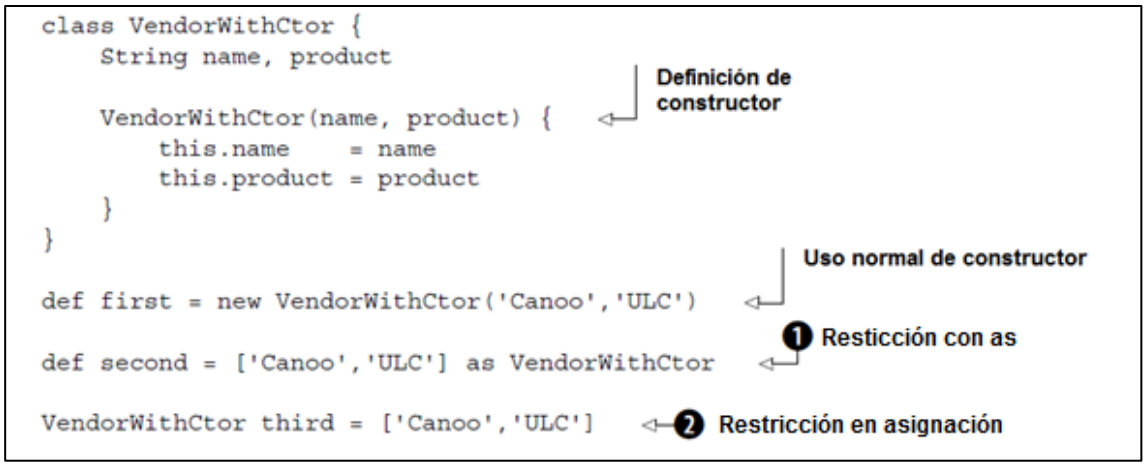

Figure 3. Call builders with positional parameters

The restriction in numbers 1 and 2 can be striking. When Groovy sees the need to restrict a list of some other type, try contacting the manufacturer of the type with all the arguments provided by the list in the order listed. This need can be enforced to restrict the keyword or can arise as assignments to statically typed references.

Ace: Change the object type.

Named parameters are useful builders. A use case that arises frequently is creating immutable classes that have some parameters that are optional. Using positional parameters quickly become cumbersome because they have to have constructors that allow all combinations of optional parameters.

For example, if two builders are desired type string argument, they could not have constructor with a single argument, because it does not distinguish whether to put the product name or attribute (both are strings). An extra argument for the distinction would be sought, or would have to strongly type parameters. To avoid this, Groovy is based on named parameters. Figure 4 shows how to use named parameters with a simplified version of the Vendor class. It is based on the implicit default constructor. (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
```

Figure 4. Parameters appointed

The example in Figure 4 shows how the named parameters are flexible to their builders. (König & Glover, 2007, pp. 186-187)

Similarly it makes for a map to the constructor of a bean that contains the names of the properties, along with associated initialization value:

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

Figure 5. Pass a map to a builder

Passing "map" directly to the Customer constructor, it is permissible to omit "map" of parentheses, as shown in customer2 initialization.

Each GroovyBean default account with this incorporated Map builder. This builder works iterating the map object and calls the appropriate property setter for each entry in the map. Any map entry that does not correspond to an actual bean property will cause an exception to be thrown. (Dearle, 2010, p. 41).

Groovy to declare two builders with the same kind of arguments, and support through the parameters are appointed for not cause error. Additionally you can tell when your definition is possible to omit the builder.

In Java, a JavaBean is a class that implements methods getters and setters for all or some of its instance fields. Groovy automatically generates getters and setters instance fields in a class and have the default visibility public. It also generates the default constructor. Instance fields with getters and setters automatically generated are known in Groovy as property, and refers to these classes as GroovyBeans, or colloquial POGO (Plain Old Groovy Object).

The closures are anonymous code fragments that can be assigned to a variable and have features that make them look like a method to the extent that it allows to pass parameters to them, and they can return a value. However, unlike the methods, the closures are anonymous, are not glued to an object. A closure is just a snippet of code that can be assigned to a variable and run later. (Dearle, 2010)

Need not be declared within classes, they can be declared anywhere.

A closure Groovy code is wrapped as an object of groovy.lang.Closure type, defined and recognized by braces {// code here}. A closure is very similar to Java methods, parameter handling can be done by dynamically typed. This feature of Groovy is one of the most important language, which makes a difference in functionality regarding Java. A closure can be passed as a parameter to another closure, Java option that allows working with their methods, it should be noted that when a closure is passed as a parameter within the parentheses, it must be declared as the last argument.

You can determine if the closure has been provided. Otherwise, you may decide to use a default implementation to handle the case. (Subramaniam, 2013)

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

Figure 6. Dynamic Closure

The code in Figure 6 shows the above where the closure sends for 3 times, the first and second with a value, and the third without it, the call will receive if lines 5 and 6, while the run else call line 7 because he does not send any value.

On line 5 it appears that is implementing the body of a method, but it is not, line 6 passes the value between the parentheses, both lines are equivalent.

A. ENCAPSULATION

Encapsulation can control how data and methods used. You can use access modifiers to prevent external methods or class methods execute read and modify their attributes. To allow other objects to query or modify the attributes of objects, classes often have access methods. (UNAM, 2013)

The following levels of access (being from low to high) must:

• **private:** accessible only from the class itself.

• **default:** accessible from the class itself and from the same package.

• **protected:** accessible from the class itself, package and subclass.

• **public:** access from any package.

Note: In Java, there are 4 levels of access, but only three access modifiers. Classes can only be public or default. (Horna, 2010)

Within the Java code only the main class must be public when more than one class in a file.

An object provides a well-defined border around a single abstraction, encapsulation and both the modularity provided as barriers around this abstraction. (Booch, 1991)

Java always requires explicitly express the public visibility of a class. By default, unless you specify otherwise, default in Groovy all classes, properties and methods are public access regardless of which has the main method. (Judd & Faisal Nusairat, 2008, p. 23).

A. RANK

A set of abstractions often form a hierarchy, and identifying these hierarchies in the design greatly simplifies understanding the problem.

(Booch, 1991) Defines the term as follows:

> *The hierarchy is a classification or sort of abstractions.*

The inheritance hierarchy is the most important classes and is an essential element of object-oriented systems. (Booch, 1991)

If a class can only receive characteristics of another base class, inheritance is called single inheritance. But if a class receive properties from more than one base class, inheritance is called multiple inheritance. (Di Serio, 2011).

In the terminology of Java, a class that is inherited is called superclass. The class is called a subclass inherits. Therefore, a subclass is a specialized version of a superclass, it inherits all instance variables and methods defined by the superclass and adds its own elements.

To inherit a class, simply the definition of a class is incorporated into the other using the extends keyword. (Schildt, 2009, p. 157)

Keyword super has two general forms. The first calls the constructor of the superclass. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

The parameter list specifies any action the need in the superclass constructor. super () must always be the first statement is executed inside a subclass constructor. (Schildt, 2009, pp. 162-163)

Sometimes it is required to call a superclass method. That is done with the Super keyword. If this refers to the current class, super refers to the superclass respect to the current class, which is an essential method to access methods overridden by inheritance.

By default Java performs these actions:

- If the first statement in a constructor of a subclass is a sentence that is not super-nor esta, invisibly adds Java and implicit super () call to the default constructor of the superclass, then starts subclass variables and then continues with the normal execution.
- If using super (...) in the first statement, then you call the superclass constructor selected, then starts the properties of the subclass and then continues with the other builder sentences.

Finally, if the first statement is this (...), then you call the builder selected by esta, and then continues with the judgments of the constructor. The initialization of variables have completed the constructor that was called by esta. (Marin, 2012)

All features of Java heritage (including abstract classes) are available in Groovy.

Because Java does not support the concept of multiple inheritance as other languages, object-oriented, it makes a simulation, introduces the concept of interfaces as an alternative to multiple inheritance, they are quite different, although the interfaces can solve similar problems . In particular:

- Since an interface, a class inherits only constant.
- Since an interface, a class can not inherit method definitions.
- The hierarchy of interfaces is independent of the class hierarchy. Several classes may implement the same interface and does not belong to the same class hierarchy. But when we speak of multiple inheritance, all classes are in the same hierarchy. (Ceballos, 2010)

In groovy, to handle multiple inheritance makes use of a concept called "Mixins". Mixins are used to inject the behavior (methods) of one or more classes in another. Normally they used withMixin annotation. (Emalvino, 2013)

About writing methods

When using inheritance, to inherit from a class we can use their methods, and there may be occasions where the method is not the father of our utility and we should create a new one with the same name, we use the Sobreecritura. (Hdeleon, 2014)

Within Java to override a method, you may want to use theOverride annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then an error is generated. (Oracle, 2014)

Unlike Java, Groovy does not handleOverride annotation to override this using the MetaClass property and the "=" operator. (EduSanz, 2010) Using the MetaClass property you may add new methods or replace existing methods of the class. If you want to add new methods that have the same name but with different arguments, you can use a shortcut notation. Groovy uses for this leftshift () (<<). (Klein, 2009)

A. TYPES (typing)

The O.Ö. languages make (for simplicity) that there is virtually no difference between types and classes, however, formally differentiation is usually done stresses that a specified type and a semantic structure and a class is a concrete implementation of a type.

In P.O.O. Classes typically define a type, and other basic types exist. In Java, so we can treat them as classes when we need it (for example, to put them where required Object), container classes (English wrapper classes) are defined. (Castro Souto, 2001)

The advantages of being strict with the types are:

• Reliability, as to detect errors at compile time are corrected.

• Readability, providing some "documentation" to the code.

• Efficiency, optimizations can be made. (Castro Souto, 2001)

Based on the concept of types, a few concepts are more support to classify the languages are handled. These are checked and ligation types, which are defined below:

In type checking it ensures that the type of construction coincides with the provisions of its context. There are two ways to handle type checking:

- Strongly typed: Requires all type expressions are consistent at compile time. The languages that handle this type are Java, C ++, Object Pascal. (Castro Souto, 2001)
- Checking weak types: All types checks are done at runtime (given, therefore, a greater number of exceptions). That is, the flexibility is much higher, but we encounter many more problems when running. Example: Smalltalk. (Castro Souto, 2001)

There are several important benefits derived from the use of language with strict types.

- "No type checking, a program can explode mysteriously in the execution.
- In most systems, the cycle edit-compile-debug is so tedious that early detection of errors is indispensable.
- The type declaration helps document the programs.
- Most compilers can generate more efficient code if you have declared types. " (Tesler, 1981)

With the above, and based on prior knowledge of how working Java and Groovy this aspect, one can conclude that Java is a language strongly typed, because when going encoding, and if he came to assign a different value type regarding the type, brand immediately compiler error, warning that something is wrong and should be revised or otherwise the execution is not possible.

Groovy, by contrast, is a language with weak type checking, assign it to a different type value assigned respectively, sees nothing and execution time is when you send errors.

In short, while Java warns about errors at compile time types, Groovy does at runtime.

Ligation types

Ligation is the process of associating an attribute to a name in the case of functions, the term binding (binding) refers to the connection or link between a function call and the actual code executed as a result of the call. (Joyanes Aguilar, 1996)

Ligation is classified into two categories: static binding and dynamic binding. (Booch, 1991)

- Tubal Static: static allocation, also known as static types temprana- ligation or ligation refers to the time when the names are linked to their types. Static binding means that the types of all variables and expressions at compile time are set.
- Tubal Dynamics: Also called late binding, means that the types of variables and expressions are not known until runtime.

Polymorphism exists when interacting characteristics of inheritance and dynamic binding. It is perhaps the most powerful feature of object-oriented languages after its ability to support abstraction, and is what distinguishes object-oriented programming with a more traditional abstract data types programming. (Booch, 1991)

That said, both Java and Groovy are languages with late binding, both support polymorphism.

Static typed

This classification, each variable must be declared with a type associated with it.

Java is a strongly typed language. Part of their security and robustness is due to this fact.

First, each variable and each expression has a type, and each type is tightly defined. Second, in all assignments, whether express or via parameter passing in the method call, the type compatibility is checked. In Java there is no automatic conversion of incompatible types as some other languages. The Java compiler checks all the expressions and parameters to ensure that the types are supported.

Java defines eight primitive types: byte, short, int, long, char, float, double and boolean. Primitive types are also called simple types. (Schildt, 2009)

Any variable used in a Java code requires declare a data type, otherwise the compiler will do its job highlighting that mistake when coding.

Dynamic typing

This type of offense does not require that the variable is assigned a data type as could be int, String, etc., as Java would just simply assign values to these variables and its data type will be considered depending on the value you assign to the variable.

SAFETY GROOVY

Regardless of whether the type of a variable is declared explicitly or not, the system is type safe. Unlike untyped languages groovy not allow treatment of an object of a type as an instance of a different type, without a well-defined conversion available. You would never treat a java.lang.String value "1", as if it were a java.lang.Number. Such behavior would be dangerous - it's the reason that Groovy does not allow this more than makes Java. (König & Glover, 2007)

Groovy to define a primitive data type to a variable as does Java is very valid, without change, to make everything easier, Groovy easy handling, without forgetting the main theme based on the way this language works primarily as an object.

Groovy designers decided to end primitives. When Groovy need to store values that have used the primitive types of Java, Groovy already uses the wrapper classes provided by the Java platform.

Whenever you see what appears to be a primitive value (for example, the number 5 in the Groovy source code, this is a reference to an instance of the appropriate wrapper class). For the sake of brevity and familiarity, Groovy can declare variables as if they were variables of primitive types.

The conversion of a single value in an instance of a type of wrapper is called boxing in Java and other languages that support the notion. The reverse action - taking an instance of a wrapper and retrieve the primitive value - is called unboxing. Groovy performs these operations automatically if necessary. This automatic boxing and unboxing is called autoboxing.

All this is transparent - you do not have to do anything in the Groovy code to enable it.

Because of this we can say that Groovy is more object-oriented Java. (König & Glover, 2007)

A. COMPETITION

For certain types of problem, an automated system may need to handle different events simultaneously. Other problems may involve many calculations that exceed the capacity of any single processor. In both cases, it is natural to consider using a distributed set of computers to implement pursued or use processors capable of multitasking. A single thread of control--called process is the root from which independent dynamic actions occurring within the system.

Systems running on multiple CPUs allow truly concurrent threads of control while systems running on a single CPU can only get the illusion of concurrent threads of control.

There are 2 types of concurrency, concurrency heavy and light. A heavy process is handled independently by the target operating system and includes its own address space. A lightweight process usually exist within a single operating system process together with other lightweight processes that share the same address space and often involve shared data.

While object-oriented programming focuses on data abstraction, encapsulation and inheritance, concurrency abstraction focuses on process and timing. The object is a concept that unifies these two points of view: each object (drawn from an abstraction of the real world) may represent a separate thread of control (an abstraction of a process). (Booch, 1991)

Based on the above Grady Booch defines competition as follows:

> *"The competition is the property that distinguishes an active object of one who is not active."*

As can be seen, the thread and process concepts that are defined below are handled to better understand the issue.

- Hilo: also known as light or process thread. Are small independent processes or pieces of a process. You can also say that a thread is a single flow of execution within a process.
- Process: is a program running in its own address space. (Cisneros, 2010)

Java supports the concept of thread from the language itself, with some classes and interfaces defined in the java.lang package and specific methods for manipulating Threads in the Object class.

You can define and instantiate a thread (thread) two ways:

- Extending class java.lang.Thread
- Implementing the Runnable interface

Since Groovy you can use all the normal facilities that Java provides for concurrency, only Groovy facilitates the work as will be seen below. (König & Glover, 2007)

The first and main feature of Groovy is that support for multithreading is implemented Closure Runnable. This allows simple definitions of threads as:

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

This even can be simplified with new static methods in the Thread class:

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

Java has the concept of a daemon thread, and therefore so does groovy. A daemon thread can be started through:

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

A. PERSISTENCE

Persistence is the property of an object through which its existence transcends time and / or space. This means that a persistent object still exists after you have completed the program that created it and also can be moved from memory location in which it was created. (Ortiz, 2014)

It is suggested that there is a continuum of existence of the object, ranging from transient objects that arise in the evaluation of an expression to the objects in a database that survive the execution of a single program. This aspect of persistence covers:

- Temporary results in the evaluation of expressions.
- Local variables in the activation procedures.
- Own variables, global variables and elements of the heap (heap) whose duration differs from its space.
- Data between executions of a program.
- Data between multiple versions of a program.
- Data surviving the program.

The traditional programming languages usually deal only with the first three types of persistent objects; the persistence of the past three types typically belongs to the domain of technology databases. (Booch, 1991)

The persistence attribute should only be present in those objects that an application requires to maintain between runs, otherwise it would probably storing a huge amount of unnecessary objects. Persistence is achieved by storing a secondary storage device (hard drive, flash memory) the information required to restore an object later. Typically persistence has been the domain of database technology, so this property has not been until recently that has been incorporated into the basic architecture of object-oriented languages.

The Java programming language allows serialize objects into a stream of bytes. This flow can be written to a file on disk and then read and deserialized to reconstruct the original object. This is done with what is called "light persistent" (lightwigth persistence in English). (Ortiz, 2014)

Serialization of an object is to obtain a sequence of bytes representing the state of the object. This sequence can be used in several ways (can be sent across the network, saved to a file for later use, used to reconstruct the original object, etc.).

Serializable object

A serializable object is an object that can be converted into a sequence of bytes. For an object is serializable, you must implement the java.io.Serializable interface. This interface does not define any methods. Just used for 'dial' those classes whose instances can be converted to sequences of bytes (and later rebuilt). As common objects such as String, Vector or ArrayList implement Serializable, so they can be serialized and rebuilt later. (Miedes, 2014)

Serialization and deserialization can be performed both Java and Groovy.

In Java, file handling is done by using the File class java.io package, which provides support through methods for operations / O.

The focus on how Groovy makes file management, compared with Java, makes little difference. First language uses the same package with working Java, only it adds several convenience methods and as always, makes reduction necessary code to work lines.

The Java JDK, addresses this need with its java.io and java.net packages. It provides support made with the File, URL and numerous versions of streams, readers and writers classes.

Over without change, Groovy extends the Java JDK its own GDK (Groovy Development Kit). GDK has a number of methods that make the magic to work more easily.

In Groovy features that can be seen with the naked eye they are as follows:

- No need for exception handling.
- The java.io package is automatically imported by the GDK, not necessitating its explicit use.
- Management BufferedReader and FileReader is not necessary.
- Only the File class is used, which is only the representation of a file and directory paths.

The elimination of the above features reduces the code into multiple lines, allowing faster encoding and more readable code.

Groovy JDK can be found on their official website. (Groovy)

A. MANAGEMENT OF OBJECTIONS

An exception is an event that occurs during the execution of a program and interrupts the normal flow of instructions. (Oracle)

Exception handling in Java is performed by five keywords: try, catch, throw, and finally throws. The following briefly describes how it works: The program statements that want to monitor, are included in a try block. If an exception occurs within the try block, it is thrown. The code can catch this exception using catch, and manage it rationally. Exceptions generated by the system are automatically sent by the Java interpreter. To send an exception manually throw keyword is used. Must be specified by clause throws any exception that is sent from one method to the external method that called it. You must put any code that the programmer wishes to be always run after a try block is completed, the block finally statement. (Ceballos, 2010).

In Groovy, exception handling is exactly the same as in Java and follows the same logic. You can specify a complete sequence of blocks try-catch-finally, or just try-catch, or simply try-finally. Note that unlike other control structures, braces are required around the block bodies whether they contain more than one statement. The only difference between Java and Groovy in terms of exceptions is that the statements of objections in the method signature is optional, even checked exceptions. (König & Glover, 2007, p. 171)

Groovy does not require handle exceptions that do not want to manage or are inappropriate in the current code level. Any unhandled exception is automatically passed to the next level (Subramaniam, 2013, p. 17).

In general, all exceptions are unproven Groovy, or rather its management is optional

A. RESERVED WORDS

Java defines a number of words of the language for the identification of operations, methods, classes, etc., in order that the compiler can understand the processes that are being developed. These words can not be used by the developer to name methods, variables or classes, because as mentioned, each has a target within language.

If an identifier to define any of the keywords the compiler warns that error to the programmer do something about it.

Table 1 shows the Java reserved words:

Table 1. Java reserved words (Schildt, 2009)

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

The Groovy language makes the implementation of new keywords, which help to manage certain characteristics in a more simple way, making the code easier to write, read and understand.

def and in are among the new keywords Groovy. def defines the methods, properties and local variables. used in loops to specify the range of a loop, as in for (i in 1..10).

The use of these keywords as variable names or method names can lead to problems, especially when existing Java code is used as Groovy code.

Nor is it a good idea to define a variable called it. Groovy not complain though, if a field with that name is used in an enclosure, the name refers to the parameter lock and not a field within the class. (Subramaniam, 2013)

Another new word that brings it as Groovy, which lets you change the type of object, the practical example was the subject Builders.

A. CONTROL STRUCTURES

Groovy supports almost the same logical structures Java control, making exception to the accepted standards or implementations thereof. Within the conditional test Boolean evaluate and make a decision based on whether the result was true or false. None of these structures should be a completely new experience for any developer of Java, but of course Groovy adds some touches.

In Groovy expression of a Boolean test can be of any type (not void). It can be applied to any object. Groovy decides whether to consider the expression as true or false by applying the rules shown in Figure 8 (König & Glover, 2007):

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

Figure 8. Sequence of rules used to evaluate a Boolean test.

if / if-else

It works exactly the same way as it does Groovy to Java.

As in Java, Boolean test expression must be enclosed in parentheses. The conditional block usually enclosed in braces. These braces are optional if the block is comprised of a single statement.

The only difference is in how Groovy interprets these conditions. Groovy can not promote a series of Boolean to true or false conditions. For example, a number other than zero is always true. (Dearle, 2010)

Elvis ternary operator (? :)

In Java, the ternary operator can say that is an if-else abbreviated as owns the C programming language Groovy supports the Java standard way to handle it, but has a similar operator called Elvis, which highlights more practicality when programming, its structure is (a: b) being the equivalent ternary (a to? b). (Dearle, 2010). The symbols "?" Should not go with a space between them, contrary to what will mark error.

Elvis works by maintaining a hidden local variable, which stores the initial result. If that result is given according to the rules of Figure 8, then that value is returned, if not, the alternative value is used. (Dearle, 2010)

Switch statement

In Java, as in the C language, the switch statement allows certain variable being tested by a list of conditions handled in the case.

Groovy, by contrast, is more friendly to data types within the case. The case elements are not the same type that receives the switch can handle both integers, lists, strings, ranges, etc. (Dearle, 2010)

While loop

This loop, regarding the syntax and logical structure is the same in both languages.

Note that Groovy does not accept the loop do {} while (), completely unknown what that sentence means if someone tries to use.

Loop

When Groovy was created, and for some time afterwards, he did not support the Java standard for loop:

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

In version 1.6, however, it was decided it was more important that supports Java constructs that try to maintain a free language that Java syntax bit uncomfortable that he inherited from his predecessors (As the C language).
Similarly supports Java for-each. (Kousen, 2014)
The for-each it is used to iterate over the elements of collections that are arrays, ArrayList, HashMap, ...

However, none of these loops is the most common way to loop in Groovy. Instead of writing an explicit loop, as in the examples above, Groovy prefer a more direct application of the iterator design pattern. Groovy adds the each method, which takes a closure as an argument for collections.

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

The each method is the construction of the most common loop in Groovy.

Groovy another loop is run by for-in, you can use the term "in" to repeat any collection. (Dearle, 2010)

A. Java Virtual Machine (JVM)

One of the interesting points is that you have Groovy uses the Java virtual machine to run the code. As is well known, Java is a compiled and interpreted language, while Groovy is played, although it is also possible to use the compiler for it.

The JVM specification provides specific definitions for the implementation of the following: a set of instructions (equivalent to a central processing unit [CPU]), a set of records, the class file format, an execution stack, stack garbage-collector, a memory area, mechanism fatal error reports, and support high-precision timing.

The format of the JVM code is compact and efficient bytecode. Represented by JVM bytecode programs must maintain discipline right kind. Most type checking is done at compile time.

Any interpreter Java compatible technology must be able to run any program with class files that conform to the class file format specified in Specification of the Java Virtual Machine. (Oracle and / or its affiliates, 2010)

The philosophy of the virtual machine is as follows: the source code is compiled, detecting the syntactic errors, and a kind of executable is generated with a code machine aimed at an imaginary machine, with an imaginary CPU. In this kind of machine code it is called intermediate code, or sometimes also intermediate language, p-code, or byte-code.

As imaginary machine that does not exist, you can run the executable, an interpreter is built. This interpreter is able to read each of the imaginary machine code instructions and execute them in real platform. This interpreter is called the interpreter of the virtual machine. (Vic, 2013)

Currently, as is well known, the JVM is no longer just for Java, there are other languages that can be compiled into the virtual machine and are available for use. These languages compile to bytecode in class files, which can be executed by the JVM.

Groovy is a language of object-oriented programming that runs on the JVM. It also retains full interoperability with Java.

the following concerning the execution of Groovy classes or scripts are mentioned and concluded the author of that letter that this is a compiled language:

"In Java compiled with javac and run the resulting Java bytecode. Groovy can be compiled and run with groovyc groovy, but really do not have to compile first. The groovy command can function with an argument of source code and compile it first and then executed. Groovy is a compiled language, but you do not have to separate the steps, although most people do. When an IDE, for example, each time you save a script or Groovy class is compiled "is used. (Kousen, 2014)

Conclusions

It was possible to confirm that the two are languages that satisfactorily meet the necessary elements to be considered Object Oriented, remembering that they are only four primary elements to consider, but in the same way meet the side, these elements are part of Oriented Model Object Graddy proposed by Booch.

Both languages have great syntactic resemblance, Groovy scripts makes use of small programs, whereas for a larger system is recommended to use classes like Java does, just to have a better development and this allows it to be modified or improved in a simple manner.

One of the most noticeable features is simplicity code that handles the new language, it is easier to program in Groovy, coming to work with Java coding is more convenient, remembering that 99% of Java code recognizes Groovy, but also the latteronly it uses what it takes to work. What could work with Java 5 lines in the simple "hello world" Groovy is done in only one using scripts, or with the same 5 lines using Java but omitting part of the forced syntax handling point that has given rise to development of this work.

Simply you can highlight the following points regarding handles Groovy syntax compared to Java:

- The classes, variables and methods are public by default.
- Groovy respecting handles multiple inheritance class hierarchy.
- Semicolons at the end of a sentence omitted.
- The parentheses are optional.
- Groovy no need to define a data type variables, since one of its strengths is that it is a dynamic language.
- Some of the shortened sentences.
- Imports of libraries ago and does not require explicitly expressed.
- The return keyword is optional.
- To implement Strings offers new ways to do besides working with the traditional way of Java.
- Reduce code to create getters and setters variables implicitly to public by default.
- Regarding control structures that Java handles the same except for the do-while it does not recognize, but all the other works with the same logic, and even makes some implementations of the same.
- Work a feature called Closure, which offers the same functionality as a method, but the difference is that the Closures are anonymous, ie they are not linked to an object.
- The handling try-catch block and all that implies exception handling makes optional.
- Groovy is pure object-oriented, apparently working with primitive variables like Java, but under the covers uses wrappers to convert to objects.

Both Java and Groovy processes running on a virtual machine, it works with a compiler and an interpreter, the first language is compiled and interpreted, while the second is played, but it also allows you to compile.

Admittedly, Java continues to excel and has managed to be one of the best languages because it offers more tools to meet the needs of programmers, allowing them to meet the requirements of users. A Groovy lacks address some points that allow use of more satisfactory manner, at least to work up to his counterpart, but not forget that gradually he himself has improved and also have been creating tools that help you cover with demand. Java, in the same way it is improving aspects, showing that it is not an easy language to replace or match, has seen him have flaws but this has helped him work on it and improve.

It would be interesting later revisit the issue hereof and see how they have progressed both languages, keep in mind that development is a topical issue with great future, there will be new

languages, others will disappear perhaps and many more will improve their characteristics such as It has been seen.

> *"Groovy's success is mainly due to their familiarity with Java. Developers want to do things, want to master the language quickly, they want power characteristics. Groovy is a more productive than Java, but still with a Java-like syntax that is already well known language. On top of that, Groovy simplifies everyday tasks that used to be complex to develop."* (White & Maple, 2014)

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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