# Conceptos y Paradigmas de Lenguajes de Programación

**Año**: 2020.

## **Integrantes**:

* Marcos Di Martino, Nro Alumno: 16161/2
* Federico Ezequiel Eseiza, Nro Alumno: 13209/7

## **Lenguajes Investigados**:

* Java.
* Ruby.

**Índice**

**Capítulo 1 Introducción…………………………………………………..2**

**Capítulo 2 Lenguaje Java………………………………………………...2**

**Capítulo 3 Lenguaje Ruby………………………………………………..3**

**Capítulo 4 Principales criterios de evaluación……………...….…….3**

**4.1. Simplicidad y legibilidad……………………………………3**

**4.2. Claridad en los bindings…………………………………….4**

**4.3. Confiabilidad…………….…………………………………….5**

**4.4. Soporte…………………...…………………………………….7**

**4.5. Ortogonalidad………………………………………………….7**

**Capítulo 5 Sintaxis…………………………………………………………..8**

**Capítulo 6 Semántica………………………………...……………………..9**

**Capítulo 7 Características de las variables…………………………..…10**

**Capítulo 8 Parámetros………………………………………………..…..…14**

**Capítulo 9 Sistema de tipos…...…………………………………….…..…17**

## **Capítulo 10 Manejo de excepciones………………………………………18**

## 

## 

## Capítulo 1: Introducción.

El presente trabajo tiene como objetivo analizar las características de los lenguajes de programación Java y Ruby. Al comienzo se introducirá al lector con una descripción de cada lenguaje para posteriormente analizar sus particularidades. Esto permitirá identificar las diferencias que poseen, con el fin de dar a conocer qué lenguaje es más apropiado según el contexto en el que nos encontramos. Mencionaremos los principales criterios de evaluación tales como legibilidad, confiabilidad, ortogonalidad, entre otros. Analizaremos la sintaxis de cada lenguaje y la semántica (estática y dinámica).

Todo esto va a estar explicado junto a un marco teórico introductorio y algunos ejemplos en código para facilitar la comprensión.

## Capítulo 2: Lenguaje Java.

Java es un lenguaje publicado en 1995 por Sun Microsystems (empresa adquirida en 2010 por Oracle). Es un lenguaje de programación orientado a objetos, muy usado actualmente. El pasaje desde otro lenguaje resulta sencillo para los programadores si ya conocen este paradigma. Su sintaxis es parecida a la de C o C++, ya que en su época era muy popular y Java se vio influenciado por la misma.

Cuenta con una JVM (Java Virtual Machine) que permite escribir el código una sola vez y ejecutarlo en diferentes dispositivos, dado que existen diferentes JVMs para las distintas arquitecturas de todas las plataformas, esto brinda portabilidad al lenguaje.

Posee un compilador el cual valida la sintaxis e informa si hubo algún tipo de error. En caso de estar el programa correctamente escrito, el código es compilado en un lenguaje intermedio llamado bytecode, el cual será interpretado posteriormente por una JVM.

## 

## Capítulo 3: Lenguaje Ruby.

Ruby es un lenguaje con un balance cuidado. Su creador, Yukihiro “Matz” Matsumoto, mezcló partes de sus lenguajes favoritos (Perl, Smalltalk, Eiffel, Ada y Lisp) para formar un nuevo lenguaje que incorpora tanto la programación funcional como la imperativa.

Desde su liberación pública en 1995, Ruby ha atraído devotos desarrolladores de todo el mundo.

Ruby es totalmente libre. No sólo gratis, sino también libre para usarlo, copiarlo, modificarlo y distribuirlo.

Inicialmente, Matz buscó en otros lenguajes para encontrar la sintaxis ideal. Recordando su búsqueda, dijo, “quería un lenguaje que fuera más poderoso que Perl, y más orientado a objetos que Python”, por lo tanto Ruby posee una sintaxis sencilla que es fácil de leer y escribir.

En Ruby, todo es un objeto. Se le puede asignar propiedades y acciones a toda información y código. La programación orientada a objetos llama a las propiedades variables de instancia y las acciones son conocidas como métodos.

En muchos lenguajes, los números y otros tipos primitivos no son objetos. Ruby sigue la influencia del lenguaje Smalltalk pudiendo poner métodos y variables de instancia a todos sus tipos de datos. Esto facilita el uso de Ruby, porque las reglas que se aplican a los objetos son aplicables a todo Ruby.

## Capítulo 4: Principales criterios de evaluación.

Para poder evaluar los lenguajes necesitamos establecer criterios de evaluación, los más importantes son: simplicidad y legibilidad, claridad en los binding, confiabilidad, soporte, abstracción, ortogonalidad y eficiencia.

### 4.1 Simplicidad y legibilidad

Para el colectivo de programadores que conocen la programación orientada a objetos, el cambio a Java puede ser realmente sencillo. Es un lenguaje bien estructurado, sin punteros y sin necesidad de tener que controlar la asignación de memoria a estructuras de datos u objetos.

En cuanto a Ruby que también es un lenguaje orientados a objetos, este ofrece una sintaxis sin tantos caracteres, haciendo que la lectura, codificación y depuración sea más amigable para el desarrollador. Además, en Ruby, los puntos y comas (;), corchetes ({}) prácticamente no existen. Como Ruby proviene de Python, Ada, Perl, Smalltalk, etc. Muchos de sus métodos heredados tienen la misma funcionalidad, provocando una mayor familiarización y que el lenguaje sea legible para cualquier lector.

### 4.2 Claridad en los binding

Java maneja dos tipos de bindings, el estático y el dinámico, en cuanto a estos es muy claro en el momento que se realiza cada uno.

#### 4.2.1 Enlace estático (static binding):

Cuando el tipo de objeto se determina en el momento de la compilación (por el compilador), se conoce como enlace estático. En Java si hay algún método privado, final o estático en una clase, hay un enlace estático.

Código de ejemplo:

*class Dog{*

*private void eat(){*

*System.out.println("dog is eating...");}*

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

*Dog d1=new Dog();*

*d1.eat(); }*

*}*

#### 4.2.2 Enlace dinámico:

Cuando el tipo de objeto se determina en tiempo de ejecución, se conoce como enlace dinámico. Ejemplo en código:

*class Animal{*

*void eat(){ System.out.println("animal is eating..."); }*

*}*

*class Dog extends Animal {*

*void eat(){ System.out.println("dog is eating..."); }*

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

*Animal a=new Dog();*

*a.eat(); }*

*}*

En el ejemplo anterior, el compilador no puede determinar el tipo de objeto, porque la instancia de Dog también es una instancia de Animal, por lo que el compilador no conoce su tipo, solo su tipo base, aquí es donde el compilador se vuelve un poco inteligente y se da cuenta que Perro tiene una sobrecarga de la misma función presente en animal así que llama la que corresponda con el tipo de dato creado (en este caso la clase Perro).

En Rubí al ser un lenguaje interpretado el tipo de objeto siempre se determina en tiempo de ejecución por lo tanto solo posee binding dinámico.

Referencia: Javatpoint (s.f.). Recuperado el día 20/04/2020 de: <https://www.javatpoint.com/static-binding-and-dynamic-binding>

### 4.3 Confiabilidad

#### 4.3.1 Chequeo de tipos

Java combina flexibilidad, robustez y legibilidad gracias a una mezcla de chequeo de tipos durante la compilación y durante la ejecución. En Java se pueden tener punteros a objetos de un tipo específico y también se pueden tener punteros a objetos de cualquier tipo. Estos punteros se pueden convertir a punteros de un tipo específico aplicando un *cast*, en cuyo caso se chequea en tiempo de ejecución de que el objeto sea de un tipo compatible.

El programador usa entonces punteros de tipo específico en la mayoría de los casos con el fin de ganar legibilidad y en unos pocos casos usa punteros a tipos desconocidos cuando necesita tener flexibilidad.

En Ruby el chequeo de tipos se hace solamente en tiempo de ejecución. Esto supone un problema ya que si por ejemplo se usa una función que no está declarada anteriormente, el error ocurre al ejecutar esa sentencia, y no en tiempo de compilación como en Java. Para paliar este tipo de inconvenientes se han desarrollado diversas herramientas que analizan el código en busca de problemas, permitiendo que los errores se detecten con anterioridad.

### Referencias:

Luis Mateu B. (1996) Recuperado el día 20/04/2020 de: <https://users.dcc.uchile.cl/~lmateu/Java/Apuntes/java.htm>

[Scott, Michael.](https://www.amazon.com/-/es/s/ref=dp_byline_sr_book_1?ie=UTF8&field-author=Michael+L.+Scott&text=Michael+L.+Scott&sort=relevancerank&search-alias=books) (s.f.) *Programming Languages Pragmatics: Capítulo 7.*

Recuperado el día 21/04/2020 de:

<http://www.dccia.ua.es/dccia/inf/asignaturas/LPP/2010-2011/clases-domingo/sesion24.html>

4.3.2 Tratamiento de excepciones

En Java, de forma similar a C++ se pueden tratar las excepciones previstas por el programador utilizando unos mecanismos, los **manejadores de excepciones,** estos se estructuran en tres bloques, el bloque **try,** el bloque **catch** y el bloque **finally**. Un manejador de excepciones es una porción de código que se va a encargar de tratar las posibles excepciones que se puedan generar. Ejemplo en código:

*class ExcepcionTratada {*

*public static void main(String argumentos[]) {*

*int i=5, j=0;*

*try {*

*int k=i/j;*

*System.out.println("Esto no se va a ejecutar.");*

*} catch (ArithmeticException ex) {*

*System.out.println("Ha intentado dividir por cero");*

*}*

*System.out.println("Fin del programa");}*

*}*

Lo que ha ocurrido es que la máquina virtual Java ha detectado una condición de error y ha creado un objeto de la clase **java.lang.ArithmeticException.** Como el método donde se ha producido la excepción no es capaz de tratarla, se trata por la máquina virtual Java, que muestra el mensaje de error anterior y finaliza la ejecución del programa.

**Referencia**: [Ken](https://glifos.umg.edu.gt/library/index.php?title=Special:GSMSearchPage&process&lang=%20%20%20%20%20%20%20%20&mode=&autor=ARNOLD,%20KEN%20) [Arnold](https://glifos.umg.edu.gt/library/index.php?title=Special:GSMSearchPage&process&lang=%20%20%20%20%20%20%20%20&mode=&autor=ARNOLD,%20KEN%20) y [Gosling](https://www.google.com.ar/search?tbo=p&tbm=bks&q=inauthor:%22James+Gosling%22&source=gbs_metadata_r&cad=4) James.(1997). El lenguaje de Programación Java™. Madrid, España: Addison-Wesley.

Ruby tiene una serie de clases, Exception y sus hijas, que ayudan a manejar los errores que pueden ocurrir. Para tratar una excepción, se pone el método que puede causar el error dentro de un bloque **begin…end**. Dentro de este bloque, se pueden poner varios rescue para cada tipo de error que pueda surgir. Ejemplo en código:

*def raise\_and\_rescue*

*begin*

*puts 'Estoy antes del raise'*

*raise 'Un error ha ocurrido' # simulamos un error*

*puts 'Estoy después del raise'*

*rescue*

*puts 'Estoy rescatado del error.'*

*end*

*puts 'Estoy después del bloque'*

*end*

*raise\_and\_rescue*

**Salida del código:**

Estoy antes del raise

Estoy rescatado del error.

Estoy después del bloque

Observar que el código interrumpido por la excepción, nunca se ejecuta. Una vez que la excepción es manejada (por el rescue), la ejecución continúa inmediatamente después del bloque begin fuente del error.

Referencia: [Dave](https://www.buscalibre.com.ar/libros/autor/thomas-davefowler-chadhunt-andy), [Thomas](https://www.buscalibre.com.ar/libros/autor/thomas-davefowler-chadhunt-andy).(2004). *Programming Ruby [ Programando Ruby ]. Segunda edición.* Raleigh, Carolina del Norte: [The Pragmatic Programmers](https://www.bookdepository.com/es/publishers/The-Pragmatic-Programmers).

### 4.4 Soporte

En cuanto a Soporte Java es muy apto ya que los programas de Java se compilan a un lenguaje intermedio denominado Bytecode. Este código es interpretado por la máquina virtual de Java (JVM) del entorno de ejecución (JRE) y así consigue la portabilidad en distintas plataformas. Un programa Java compilado en Bytecode se puede ejecutar en sistemas operativos como Windows, Linux, Android, Solaris, Mac Os, o IOs utilizando el entorno de ejecución de Java (JRE) apropiado.

El hecho de que Ruby se ejecuta directamente en un intérprete y Java mediante una máquina virtual hace que sean completamente portables. Si existe un intérprete o una máquina virtual para una arquitectura y un sistema operativo específico, entonces el programa puede correr en dicha combinación de arquitectura y SO.

### 4.5 Ortogonalidad

Se refiere a que un conjunto relativamente pequeño de constructores primitivos pueden ser combinados para construir el flujo y las estructuras de datos de todo el lenguaje. Ejemplo en código:

*public class Ejemplo{*

*public static void main(String…args){*

*//Inicializar a y b al valor 1*

*a = b;*

*//modificar b para que valga 0 } }*

Si **a** y **b** son tipos primitivos, entonces **a** valdría 1, mientras que si se modelizan como objetos, entonces valdría 0 (el cambio le afecta porque **a** es una referencia que apunta **b**).

Una situación similar aparece en el paso de parámetros de Java, para los tipos primitivos es por valor, mientras que para los objetos es por referencia. Se puede simular el paso por valor mediante clonación. Un usuario del lenguaje Java se ve obligado a tener en cuenta que la asignación funciona diferente con tipos de datos primitivos que con objetos, lo cual complica un poco más el aprendizaje y el uso del lenguaje.

En cuanto a la ortogonalidad de Ruby, este es un lenguaje que representa bien este criterio ya que el simple hecho de que toda sentencia del lenguaje sea una expresión (incluso las estructuras de control) permite realizar combinaciones y/o composiciones sin ninguna clase de limitación. Otro factor que favorece la ortogonalidad es que casi todo en este lenguaje son objetos.

## Capítulo 5: Sintaxis.

La sintaxis es un conjunto de reglas que define cómo se componen las letras, los dígitos y otros caracteres para formar programas. Los elementos más importantes que la componen son: **Conjunto de caracteres**, **Identificadores**, **Operadores**, **Palabra clave o Palabra reservada**, **Comentarios** y **Uso de blancos.**

## **Identificadores:** Un identificador es el nombre de variables, métodos, clases e interfaces. Un identificador es una secuencia ilimitada de caracteres alfabéticos o dígitos (unicode) que comienzan con un carácter alfabético.

**En Java:**  Los identificadores que comienzan con una letra mayúscula se reservan solo para nombres de clase o interfaces. Los identificadores que están todos en mayúsculas representan constantes. Los nombres de las variables y métodos comienzan con letras minúsculas.

**En Ruby**: Las variables y otros identificadores normalmente comienzan con alguna letra del alfabeto o algún modificador especial, pero siempre debe seguir estas reglas:

● Variables locales (pseudovariables como *self* o *nil*) deben comenzar con una letra minúscula o el carácter underscore (\_).

● Variables globales deben comenzar con el signo dólar ($).

● Variables de instancia (las que están dentro de un objeto) deben comenzar con el signo arroba (@).

● Variables de clase (dentro de una clase) deben comenzar con 2 signos arroba (@@).

● Constantes deben comenzar con la primera letra en mayúscula.

● Cuando se quiera formar identificadores, el underscore (\_) puede ser utilizado como letra minúscula.

● Variables especiales comienzan con el signo dólar ($) como por ejemplo $1 o $/.

**Operadores:** Los operadores permiten a las expresiones combinarse en expresiones más complejas.

**En Java:** Java proporciona una colección grande de operadores que se pueden utilizar para manipular datos, incluyendo operadores aritméticos, asignación ,lógicos y de moldeado. Las reglas de asociación y de prioridad se utilizan para determinar cómo evaluar expresiones utilizando operadores. Los grupos de operadores más importantes son:

* **Operadores aritméticos:** Los operadores aritméticos se utilizan en expresiones matemáticas de igual modo que se utilizan en Álgebra.

“+” (suma), “-“ (resta), “ \*” (multiplicación), “/” (división), “%” (módulo), “++” (incrementó en 1) “—“ (decremento en 1)

* **Operadores de asignación:** El operador de asignación simple se utiliza para asignar un valor a una variable, vas = expresión. Otros operadores de asignación combinan la asignación con una operación aritmética. Ejemplo: Var=var op expresión, equivale a var op =expresión; X + y equivale a x=x+y..
* **Operadores relacionales:** Los operadores relacionales determinan la relación que un operador tiene con otro. Ejemplo : “= =” (igual a), “! =” (no igual a), “>” (mayor que), ”<” (menor que), “>=” (mayor o igual que), “<=” (menor o igual que)0.

**En Ruby:** Ruby es compatible con un amplio conjunto de operadores. La mayoría de éstos son realmente llamadas a métodos. Por ejemplo, a + b se interpreta como un. + (B), en el que la variable se invoca un método que apunta +, b como un parámetro en la llamada al método. Para cada operador (+ - \* / \*\*% y | ^ << >> && ||), tiene un operador de asignación abreviatura correspondiente (+ = - =, etc.). En lo que respecta a Ruby este comparte los mismos operadores de los tres grupos mencionados anteriormente en Java.

**Comentarios:** Los comentarios permiten añadir al código fuente notas o comentarios de texto que son ignorados por el compilador. Los comentarios vienen en tres formatos.

**En Java**: Los comentarios de una sola línea comienzan con // y continúan hasta el final de la línea. Los comentarios comienzan con /\* y terminan con \*/. Los comentarios de documentación son un tipo especial de comentarios multi línea que arranca con /\*\*. Se utiliza para empotrar en la documentación del código fuente de una clave y se puede leer con el propio código fuente con la herramienta Javadoc para genera documentación HTML para sus clases.

**En Ruby:** Los comentarios en Ruby comienzan con el signo (#) , el intérprete considera como comentario todo lo que viene después de este signo hasta el final de línea. Los comentarios no deben estar declarados entre otras sentencias de Ruby, por que no serían considerados como tal, sino como parte de la sentencia. Por ejemplo:

|  |  |
| --- | --- |
| 1  2  3 | x = y **+** 5 *# Esto es un comentario.*  *# Otro comentario.*  **print** "# Pero esto no lo es." |

**Palabra clave o palabra reservada:** Palabra clave o keywords, son palabras claves que tienen un significado dentro de un contexto. Palabra reservada, son palabras claves que además no pueden ser usadas por el programador como identificador de otra entidad.

Ventajas de su uso:

- Permiten al compilador y al programador expresarse claramente

- Hacen los programas más legibles y permiten una rápida traducción.

**En Java:** Ciertas palabras están reservadas para uso interno y no se pueden utilizar como nombres de variables. Algunas de las más utilizadas son:

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | return | static | import |
| conft | throw | tranfien | private |
| int | break | double | switch |
| public | default | if new | class |
| this | fov long | try | final |
| bolean | glont | catch | instance |
| continue | throw | implemet | of |
| float | goto | package | protecte |
| interface | native | extendy | while. |

**En Ruby:** La siguiente tabla muestra las palabras reservadas más utilizadas en Ruby. Estas palabras no pueden ser retenidos como el nombre de una constante o variable. Sin embargo, pueden ser utilizados como el nombre del método.

|  |  |  |  |
| --- | --- | --- | --- |
| BEGIN | do | next | then |
| END | else | nil | true |
| alias | elsif | not | undef |
| and | end | or | \_\_FILE\_\_ |
| begin | ensure | redo | until |
| break | false | rescue | when |
| case | for | retry | while |
| class | if | return | super |
| def | in | self | \_\_LINE\_\_ |

**Uso de blancos:** Carácter en blanco tanto en el código de Java como Ruby, tales como espacios y tabulaciones en general se ignora, excepto cuando aparecen en la cadena si no son ignorados.

**Referencias**:

- Recuperado el día 21/04/2020 de:<http://www.w3big.com/es/ruby/ruby-syntax.html>

- Libro APÉNDICE W3 Guía de sintaxis del lenguaje Java 2

## Capítulo 6: Semántica

La semántica describe el significado de los símbolos, palabras y frases de un lenguaje ya sea lenguaje natural o lenguaje informático. Esta puede clasificarse en dos tipos semántica dinámica y semántica estática

**Semántica estática:** Se las llama así porque el análisis para el chequeo puede hacerse en compilación. Para describir la sintaxis y la semántica estática formalmente sirven las denominadas gramáticas de atributos. Los errores en un programa relacionados con la semántica estática son detectados por el compilador.

Un ejemplo en Java sería la sentencia: “for (**x** = 1; **x** < 10; **x**++)” en donde la variable **x** no está definida, esto produce un error semántico que es detectado por el compilador.

En Ruby todos los errores de semántica se detectan en ejecución ya que es un lenguaje interpretado.

**Semántica dinámica:** Es la que describe el efecto de ejecutar las diferentes construcciones en el lenguaje de programación.Su efecto se describe durante la ejecución del programa. Los programas solo se pueden ejecutar si son correctos para la sintaxis y para la semántica estática. Ejemplo en código:

**En Java:**

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

*String saludo;*

*System.out.println(saludo) ; }*

**En Ruby:**

puts saludo

En ambos casos se produce un error relacionado con la semántica dinámica ya que la variable saludo no está inicializada con ningún valor, por lo tanto imprimiría basura. Este error es detectado en tiempo de ejecución.

## Capítulo 7: Características de las variables

Las variables tienen características diferentes en los distintos lenguajes, en el código proporcionado a continuación, vamos a explicar los atributos de las variables en Java.

*1 class Animal{*

*2 private static int count;*

*3 boolean hungry = true;*

*4 void eat(){ System.out.println("animal is eating..."); }*

*5 void ate(){ count++; }*

*6 }*

*7 public class Dog extends Animal {*

*8 void eat(){ System.out.println("dog is eating..."); }*

*9 public static void main(String args[]){*

*10 Animal a; a = new HelloWorld();*

*11 if(a.hungry){*

*12 a.eat();*

*13 a.hungry = false;*

*14 a.ate();*

*15 }*

*16 System.out.println("Hungry is "+a.hungry);} }*

### ***Nombres***:

Según el diseño que le dieron a java, los nombres pueden tener cualquier longitud, iniciando siempre con un caracter del alfabeto mayuscula o minuscula seguido de caracteres y/o números, el único carácter especial aceptado para los nombres es el guión bajo (“\_”). Por convención los nombres de las clases inician con letra Mayúscula, y el resto de nombres de métodos y variables inician con minúscula.

Como podemos ver en la línea 1 del código, se declara la clase Animal iniciando con la letra (A) mayúscula, mientras tanto en la línea 4 el método eat comienza con letra minúscula. Hay que tener en cuenta igual que hay palabras claves y reservadas; Por ejemplo, if (linea 11), while, switch, import, public (línea 7), extends (línea 7), etc.

### ***Alcance***:

Se puede definir 4 tipos de variables en Java, hay variables de Instancia, variables de clase, variables locales y parámetros.

Las variables de instancia almacenan la información en campos no-estáticos y se declaran sin el modificador “static” y puede llevar escrito un modificador para su visibilidad. Por ejemplo, la variable de instancia que es pública por defecto, declarada en la línea 3.

Las variables de clase se declaran con el modificador “static” y hay una sola copia para todas las instancias de la clase. Por ejemplo, la variable de clase count, que además es privada, declarada en la línea 2.

Las variables locales se declaran parecido a las variables de instancia, no utiliza palabras clave para identificarse como local, depende donde la variable es declarada.

Por último los parámetros, que son variables declaradas en la firma de un método. Con valores iniciales según los datos proporcionados al llamar el método.

El alcance en Java está definido por la visibilidad que se le otorgue y de donde se haya declarado.

Una variable Local es visible desde donde se haya declarado hasta el fin del método que lo declaró. Por ejemplo, solo se puede referir a “a” desde su declaración en la línea 10, hasta la finalización del método main (línea 16) que fue dentro del método que fue declarado. Los parámetros, se conocen en todo el método que fue llamado, como el parámetro args[] que es conocido desde su declaración (línea 9) hasta el fin del método (línea 16). Las variables de clase y de instancia se conocen en cualquier método declarado dentro de la clase y puede extenderse su alcance a otras clases sólo si no tiene el modificador private y antes se haya instanciando la clase. Por ejemplo, la variable de instancia “hungry” declarada en la línea 3 puede ser accedida desde otras clases directamente llamando a la instancia, como lo es utilizada en la línea 13, sin embargo, la variable de clase “count” (línea 2) no es visible ni puede ser accedida por ninguna otra clase que no sea la clase “Animal”.

### ***Tipo***:

En java hay tipos de datos primitivos, tipos estructurados y tipos envoltorio o wrappers.

Los tipos de datos primitivos son los encargados de gestionar la información más básica que te pueden proporcionar, como lo son los números, almacenando un único valor. Java en para los números enteros existen los tipos: *byte, short, int y long*, que lo único que los diferencia es el rango de números que se puede almacenar en la variable. Para números en punto flotante: *float y double*. Para los caracteres se utiliza *char* y, por último, para facilitar los valores verdadero y falso, obtenido de evaluar expresiones lógicas, se utiliza *boolean.*

Los tipos de datos estructurados están destinados, en su mayor parte, a almacenar múltiples valores, también utilizados para representar “objetos”. Es común ver el tipo “string”, que almacena una cadena de caracteres. También se pueden declarar estructuras como lo son los vectores y matrices, partiendo de un tipo y siguiendo con llaves (“[]”), por ejemplo, *“int[] c”* es la declaración de un vector de enteros, llamado “c”. Se puede declarar cualquier clase que puede funcionar como un tipo para una variable y otorgarle el comportamiento que sea conveniente. Como está mostrado de la línea 1 a las 6 que se declara la clase animal que puede almacenar un dato para saber si tiene hambre (“hungry”) de tipo boolean y un contador (“count”) de tipo int para almacenar la cantidad de veces que comieron los animales. Además es posible obtener clases hechas por terceros y hay muchísimas clases de la biblioteca estándar de java (como lo es String y Object).

Los tipos datos wrapper un tipo de dato estructurado, equivalente a cada uno de los tipos primitivos, utilizados, por ejemplo, para facilitar datos por referencia. Los tipos wrapper son: *Byte, Short, Integer, Long, Float, Double, Boolean y Character*.

Java es un lenguaje de tipado estático. Lo que significa es que al momento de declarar una variable, se le indica explícitamente, en Java, el tipo de dato que va a poder almacenar desde su declaración hasta el fin de la ejecución.

### L-Value:

En Java se aloca la memoria que utilizan las variables dinámicamente, tanto automática, como explícitamente dependiendo el tipo de dato que se declare. Por ejemplo, se reserva el espacio para “count” (línea 2) una vez ejecutada esa línea de código, en cambio, la variable “a” (línea 10) no reserva la memoria hasta ejecutar la instrucción que contiene el new.

### R-Value:

Java puede tener variables tanto dinámicas como constantes, además de inicializar todas sus variables con valores por defecto para evitar errores y el uso de valores nulos.

La declaración de constantes, se declaran como variables de instancia con el modificador *“final”.*

La inicialización de las variables de tipos primitivos es: para los números: 0 o 0.0 (ejemplo, línea 2, count inicializa en 0); para char “*u0000*”; para boolean: “*false*”. Para tipos estructurados el valor inicial por defecto es: “*null”*.

## 

## Capítulo 8: Parámetros

Una rutina es un conjunto de sentencias que simbolizan una acción abstracta, representa una unidad de programa. Para poder compartir datos con ellas es necesario el uso de **parámetros**.

En Java todos los parámetros de los métodos se pasan por valor. Cuando se realiza la llamada a un método, los parámetros formales (parámetros indicados en la declaración) reservan un espacio en memoria independiente y reciben los valores de los parámetros reales (parámetros indicados en la llamada al método). Los efectos del paso por valor son:

* Cuando el argumento es de tipo primitivo, el paso por valor significa que durante la ejecución del método se reserva un nuevo espacio para el parámetro formal y no se puede modificar el parámetro real durante la ejecución del método.
* Cuando el argumento es de tipo referencia (por ejemplo, un array, un objeto,) el paso por valor significa que no puede modificarse la referencia, pero se pueden realizar llamadas a los métodos del objeto y modificar el valor asignado a las variables miembro accesibles del objeto durante la ejecución del método.

En Java los parámetros se declaran en la cabecera de la declaración de los métodos. Al declararse el parámetro, se indica el tipo de dato y el identificador correspondiente. Los parámetros o argumentos de un constructor o de un método pueden ser de cualquier tipo, ya sean tipos primitivos o referencias de objetos (en este caso debe indicarse el identificador de la clase correspondiente). Ejemplos de declaraciones de cabeceras de métodos:

*// Dos parámetros, a y b de tipo int*

*public int elMayor(int a, int b) {*

*...*

*}*

*// Un parámetro, v, array real*

*public static double sumatorio (double [] v) {*

*...*

*}*

*// Un parámetro de la clase Fecha*

*public boolean caducado (Fecha fechaLimite) {*

*...*

*}*

los parámetros de un mismo método no pueden compartir el mismo identificador (no pueden coincidir) con el de una variable local, sinó se generará un error de compilación.

Por ejemplo, en el siguiente programa el método cambiar utiliza un parámetro de tipo primitivo (a, un valor numérico entero) y un parámetro de tipo referencia (b, un *array* de valores enteros):

*public class Parametros {*

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

*int n;*

*int [] v = new int[2];*

*n=10;*

*v[0]=20;*

*v[1]=30;*

*System.out.println("Antes: " + n + " " + v[0] +" "+ v[1]);*

*cambiar(n, v);*

*System.out.println("Después: " + n +" "+ v[0] +" "+ v[1]);*

*}*

*public static void cambiar (int a, int [] b) {*

*a = 50;*

*b[0] = 60;*

*b[1] = 70;*

*System.out.println("Dentro: " + a + " " + b[0] +" "+ b[1]);*

*}*

*}*

La ejecución del código anterior origina la siguiente salida por pantalla:

$>java Parametros

Antes: 10 20 30

Dentro: 50 60 70

Después: 10 60 70

Esto es a menudo una fuente de errores: el programador escribe un método que trata de modificar el valor de uno de sus parámetros y el método no funciona como esperaba.

Referencia: Arkaitzgarro (s.f.). Recuperado el día 21/05/2020 de:

<https://www.arkaitzgarro.com/java/capitulo-15.html>

En Ruby los parámetros se pasan por valor. Todo es un objeto, por lo tanto, esto significa que lo que se pasa es una referencia al objeto (que no se puede modificar) pero se pueden realizar llamadas a los métodos del objeto y modificar el valor asignado a las variables miembro accesibles del objeto durante la ejecución del método.

A diferencia de Java, Ruby deja especificar los valores por defecto de los argumentos, que son usados si no se especifica un valor explícitamente. Se hace esto mediante el operador de asignación (=).

Ejemplo:

*def titulo\_subrayado(titulo, caracter="\*")*

*puts titulo*

*puts caracter \* titulo.length*

*end*

*# bloque principal*

*titulo\_subrayado "Sistema de Administracion"*

*titulo\_subrayado "Ventas", "-"*

Por defecto, todos los métodos retornan el resultado de la última línea que se ejecute en el método. Aunque si se quiere se puede especificar con la palabra reservada “return”.

*def hello(name)*

*return "Hola #{name}"*

*end*

*# bloque principal*

*puts hello("Germán")*

*puts hello "David"*

Aunque en general los programadores de Ruby suelen omitir el “return” a menos que sea completamente necesario.

## Capítulo 9: Sistema de tipos

El Sistema de tipos es un conjunto de reglas usadas por un lenguaje para estructurar y organizar sus tipos.El objetivo de un sistema de tipos es escribir programas seguro. Conocer el sistema de tipos de un lenguaje nos permite conocer de una mejor forma los aspectos semánticos del lenguaje. El sistema de tipos se puede dividir en dos grupos el tipado fuerte y tipado débil.

Los lenguajes con un tipado fuerte son aquellos que no permiten comparar u operar con tipos de datos distintos sin realizar una conversión previa. Un ejemplo de esto es **Java**.Las ventajas de este tipo de tipado es que permite un código más expresivo, podremos saber de qué tipo espera un argumento una función. Además, permite que ocurran menos errores ya que una vez declarada una variable sabremos de qué tipo es y por lo tanto podemos evitar por ejemplo intentar realizar operaciones entre variables de distintos tipos. Una desventaja de este podría ser escribir más código ya que tenemos que declarar el tipo de variable al declararla.

Por otro lado, el tipado débil es en donde no indicamos el tipo de variable al declararla. La verdadera diferencia es que podemos asignar, por ejemplo, un valor entero a una variable que anteriormente tenía una cadena. También podemos operar aritméticamente con variables de distintos tipos. Por ejemplo, sumar “x” + 5. **Ruby** es un ejemplo de esto.

**Ventajas:**

* Podemos cambiar el tipo de la variable sobre la marcha. Por ejemplo, asignar un string a un int
* Escribimos menos código
* Nos olvidamos de declarar el tipo

**Desventajas:**

* Al hacer operaciones, a veces éstas salen mal. Por ejemplo, puede que intentemos sumar 500 + “400.00” + 10, cosa que será errónea
* Hay que castear muchas veces.En ocasiones, tendremos que castear forzosamente las variables para que se comporten como queremos y no generan errores como los mencionados arriba.
* Código menos expresivo. Al declarar los argumentos de una función no sabemos si ésta espera un flotante, un entero, un string, etcétera. Tenemos que ir a la función, ver lo que hace e inferir el tipo de variable que espera

## Capítulo 10: Manejo de excepciones

Una excepción es una situación anómala que se da en la ejecución de un programa y que se supone que ocurre con poca frecuencia. Para que un lenguaje trate excepciones debe haber un modo de definirlas y de invocarlas.

En Ruby el método raise procede del módulo Kernel. Por defecto, **raise** crea una excepción de la clase **RuntimeError**. Para lanzar una excepción de una clase específica, se puede poner el nombre de la clase como argumento de raise.

*def inverse(x)*

*raise ArgumentError, 'El argumento no es numerico' unless x.is\_a? Numeric*

*1.0 / x*

*end*

*puts inverse(2)*

*puts inverse('patata') # da un error que es manejado por raise*

Notar que a los métodos que actúan como preguntas, se les pone un **?** al final: **is\_a?** pregunta al objeto cuál es su tipo. Y **unless** cuando se pone al final de una instrucción, significa que **NO** se ejecuta cuando la expresión a continuación es verdadera.

Al escribir **rescue** sin parámetros, el parámetro **StandardError** se toma por defecto. En cada rescue se pueden poner varias excepciones a tratar. En el caso de poner múltiples rescues:

*begin*

*rescue UnTipoDeExcepcion*

*rescue OtroTipoDeExcepcion*

*else*

*# Otras excepciones*

*end*

Ruby compara la excepción que produce el error, con cada rescue hasta que sea del mismo tipo; o sea una superclase de la excepción. Si la excepción no concuerda con ningún rescue, usar **else** se encarga de manejarla.

Para saber acerca del tipo de excepción, hay que mapear el objeto **Exception** a una variable usando rescue:

*begin*

*raise 'Test de excepcion'*

*rescue Exception => e*

*puts e.message # Test de excepción*

*puts e.backtrace.inspect # ["nombre de fichero:linea de la excepción"]*

*end*

Si además de manejar la excepción, se necesita que se ejecute un código, se debe usar la instrucción **ensure**: lo que haya en ese bloque, siempre se ejecutará cuando el bloque begin…end termine.

Aquí hay algunas excepciones de las más comunes, con la causa que las origina y un ejemplo:

* RuntimeError : la excepción que se lanza por defecto. Ejemplo:

raise

* NoMethodError : el objeto no puede manejar el mensaje/método que se le envía. Ejemplo:

string = ‘patata’

string.multiplicarse

* NameError : el intérprete encuentra un identificador que no puede resolver ni como método, ni como variable. Ejemplo:

a = variable\_sin\_definir

* IOError : lectura de un stream cerrado, escribir a un sistema de sólo lectura y operaciones parecidas. Ejemplo:

STDIN.puts(¡”No escribas a STDIN!”)

* TypeError : un método recibe un argumento que no puede manejar. Ejemplo:

a = 3 + "no puedo sumar un string a un número!"

* Errno::error : errores relaccionado con el fichero IO. Ejemplo:

File.open(-12)

* ArgumentError : causado por un número incorrecto de argumentos. Ejemplo:

def m(x)

end

m(1,2,3,4,5)

Referencia: Ruby Tutorial - Wikidot(s.f.). Recuperado el día 24/05/2020 de:

<http://rubytutorial.wikidot.com/excepciones>

Cuando un programa **Java** viola las restricciones semánticas del lenguaje (se produce un error), la máquina virtual Java comunica este hecho al programa mediante una excepción.

Muchas clases de errores pueden provocar una excepción, desde un desbordamiento de memoria o un disco duro estropeado hasta un disquete protegido contra escritura, un intento de dividir por cero o intentar acceder a un vector fuera de sus límites. Cuando esto ocurre, la máquina virtual Java crea un objeto de la clase exception o error y se notifica el hecho al sistema de ejecución. Se dice que se ha lanzado una excepción (“Throwing Exception”).

Un método se dice que es capaz de tratar una excepción (“Catch Exception”) si ha previsto el error que se ha producido y prevé también las operaciones a realizar para “recuperar” el programa de ese estado de error.

En el momento en que es lanzada una excepción, la máquina virtual Java recorre la pila de llamadas de métodos en busca de alguno que sea capaz de tratar la clase de excepción lanzada. Para ello, comienza examinando el método donde se ha producido la excepción; si este método no es capaz de tratarla, examina el método desde el que se realizó la llamada al método donde se produjo la excepción y así sucesivamente hasta llegar al último de ellos. En caso de que ninguno de los métodos de la pila sea capaz de tratar la excepción, la máquina virtual Java muestra un mensaje de error y el programa termina.

Los programas escritos en Java también pueden lanzar excepciones explícitamente mediante la instrucción **throw**, lo que facilita la devolución de un “código de error” al método que invocó el método que causó el error.

Los manejadores de excepciones se estructuran en tres bloques:

* **El bloque try**: las instrucciones susceptibles de generar una excepción se escriben en este bloque. En caso de haber una excepción se abandonará este bloque y se continuará en el bloque o bloques **catch**
* **El bloque catch**: analiza la excepción ocurrida en el bloque **try**. Se debe declarar un objeto cuya clase es la clase de la excepción que se desea tratar o una de sus superclases.
* **El bloque finally**: se utiliza para ejecutar un bloque de instrucciones sea cual sea la excepción que se produzca. Este bloque se ejecutará en cualquier caso, incluso si no se produce ninguna excepción. Sirve para no tener que repetir código en el bloque try y en los bloques catch.

Las excepciones son objetos pertenecientes a la clase **Throwable** o alguna de sus subclases.

Dependiendo del lugar donde se producen existen dos tipos de excepciones:

* Las excepciones **síncronas** no son lanzadas en un punto arbitrario del programa sino que, en cierta forma, son previsibles en determinados puntos del programa como resultado de evaluar ciertas expresiones o la invocación de determinadas instrucciones o métodos.
* Las excepciones **asíncronas** pueden producirse en cualquier parte del programa y no son tan “previsibles”. Pueden producirse excepciones asíncronas debido a dos razones:

**Referencia**: [Ken](https://glifos.umg.edu.gt/library/index.php?title=Special:GSMSearchPage&process&lang=%20%20%20%20%20%20%20%20&mode=&autor=ARNOLD,%20KEN%20) [Arnold](https://glifos.umg.edu.gt/library/index.php?title=Special:GSMSearchPage&process&lang=%20%20%20%20%20%20%20%20&mode=&autor=ARNOLD,%20KEN%20) y [Gosling](https://www.google.com.ar/search?tbo=p&tbm=bks&q=inauthor:%22James+Gosling%22&source=gbs_metadata_r&cad=4) James.(1997). El lenguaje de Programación Java™. Madrid, España: Addison-Wesley.