

## **Resumen**

El presente documento contiene todo lo referente al desarrollo del lenguaje Vulcan para la asignatura de Lenguajes y compiladores de la carrera de Ingeniería de Sistemas de Información de la Universidad Nacional Mayor de San Marcos.

El lenguaje Vulcan es un lenguaje de programación multiparadigma ya que es capaz de soportar la programación estructurada y orientada objetos. Se caracteriza por ser estático y de tipado fuerte; y por tener una sintaxis amigable a otros lenguajes de programación.

En este lenguaje se recopilan muchos elementos de diversos lenguajes y se unifican para concibir al lenguaje Vulcan. Los lenguajes bases son: Pascal, Python, C++, Java y Latex.

# Introducción

Durante los últimos 70 años se han desarrollado diferentes lenguajes de programación, exponentes como C, C#, C++, Java, Python, Pascal, entre muchísimos otros, nacen para ayudar al programador con facilidades en gramática, o para especializarse en ciertos campos como inteligencia artificial, programación orientada a objetos o seguridad informática. Además, los lenguajes a través del tiempo se van actualizando o generando actualizaciones que mejoran funcionalidades o paquetes del mismo lenguaje, de tal manera que permitan funcionar al lenguaje con un mejor rendimiento o simplemente para reparar ciertos problemas del mismo. Esto, a su vez, promueve el nacimiento de nuevos lenguajes cuando se distancia mucho del original, marcando cierta evolución de un lenguaje en específico. Otros, también, nacen inspirados en otros lenguajes, pero orientados a otros paradigmas, como lo es Java con C++. ¿A qué conlleva todo esto? Es importante entender como se influencia, se crea y evoluciona un lenguaje de programación para entender la razón del funcionamiento general de los lenguajes de programación, el paradigma que aborda cada lenguaje. En las siguientes páginas se presentará una simulación o pequeño proyecto de la estructuración y creación de un lenguaje de programación, si bien es primitivo, este lenguaje nos permitirá explorar los límites y excepciones que puede tener un lenguaje de programación.

# 1. Bases del Lenguaje Vulcan

El lenguaje Vulcan fue creado bajo una postura ecléctica, de esta manera se extrajo lo "mejor" de cada lenguaje de programación para dar a conocer un lenguaje con una curva de aprendizaje no tan áspera como en otros lenguajes, pero sin dejar los elementos importantes que debería tener todo lenguaje de programación.

Se dice que el lenguaje Vulcan es estático porque su definición es constante, una particularidad que comparte con la mayoría de lenguajes de programación. También es un lenguaje de tipado fuerte, esta característica hace referencia al hecho de tener que realizar un proceso de "casting" para una variable ya definida si es que quieres ser utilizada de como otro Tipo de Dato.

Este lenguaje soporta el Paradigma Orientado a Objetos, no es obligatoria su implementación a diferencia de otros lenguajes como Java, no obstante, algunos Tipos de Datos Primitivos son Objetos complejos, como es el caso del Tipo de Dato "String", por consiguiente, es fundamental dominar este paradigma para sacar el máximo partido de este lenguaje de programación. El siguiente documenta el Lenguaje Vulcan de la siguiente forma: Tipos de Datos Primitivos, Operadores, Operaciones con cadenas, .

## 1.1. Tipos de Datos Primitivos


Los Tipos de Datos Primitivos del lenguaje Vulcan son los siguientes:

Tipo de Dato	Abstracción	Descripción
integer	Número Entero	Representa un entero con signo
flot	Número Flotante	Representa un número de coma flotante
String	Cadena de caracteres	Representa una palabra o caracteres
boolean	Valor booleano	Representa el tipo de dato lógico para representar valores de lógica binaria

Para definir un tipo de dato se escribe el tipo de seguido de un **identificador** o **nombre de variable**, la sintaxis es similar a la de c++ por lo cual, si el lector domina este lenguaje, aprender estas declaraciones no debería ser problema, apreciemos el siguiente ejemplo:

La **línea 1** hace referencia a una declaración simple de variable, en este caso, el tipo de dato entero. Si queremos definir varias variables, podemos hacerlo con una simple

---



```
1 integer variable_entera;
2 flot a, b, c;
3 String nueva_cadena = "Hola mundo";
4 boolean p = FALSE, q = TRUE;
5 boolean r = p and q;
```

sentencia separando por comas como se observa en la **línea 2**. Debemos aclarar que todo elemento delimitado por comillas dobles “Esto es una cadena”, con lo cual nos vemos en la necesidad de almacenarlo en una variable de tipo String. El tipo de dato String inicia con mayúscula porque es una clase, más adelante se profundizará sobre esto. Por último definimos los tipos booleanos e hicimos algunas operaciones entre ellos como se observa en la **línea 4 y 5**

## 1.2. Operadores

Los operadores es una parte fundamental de cualquier lenguaje de programación, en Vulcan definimos los siguientes operadores: operadores aritméticos, lógicos, de comparación y otros.

### 1.2.1. Operadores Ariméticos

Los operadores aritméticos son aquellos que nos permitiran operar con distintos valores numéricos tanto enteros como de punto flotante. En la siguiente tabla se ilustran los operadores ariméticos.

Operador	Sintaxis
Suma	+
Resta	-
Multiplicación	*
División	/
Potencia	**
Módulo	mod
Incremento	++
Decremento	--

Ejemplo de codificación con operadores en Vulcano: **Observaciones:**

```

1 print(3+2);
2 integer variable = 36/6;
3 variable++;
4 flot residuo = variable mod 7;
5 residuo--;
6 print(residuo**2);

```

- *El uso de los operadores resulta muy intuitivo.*
- *El único operador que debe ser delimitado por espacios es el operador Módulo.*
- *En el ejemplo anterior se utilizó la función `print()` para mostrar datos en consola como se observa en la línea 1 y 2.*
- *El uso de parentesis es de fundamental ayuda al momento de hacer operaciones*
- *El operador incremento y decremento son los que tienen máxima prioridad*
- *El operador raíz cuadrada no está definido pero se puede obtener el mismo resultado utilizando el operador potencia con exponentes reales*

### 1.2.2. Operadores Lógicos y de Comparación

Los operadores lógicos y de comparación, si bien son dos tipos de operadores diferentes, están íntimamente ligados. El primer tipo nos permitiría operar Tipos de Datos booleanos, mientras que los de comparación nos permitirían comparar variables, pero sobre todo, formar expresiones booleanas. Los operadores son los siguientes:

Operador	Sintaxis
Mayor	>
Menor	<
Mayor o igual	>=
Menor o igual	<=
Igual	==
Negación lógica	!
Y lógico	and
O lógico	or
incremento lógico	++

Ejemplo de codificación:

```
1 integer valor = -1;
2 float valor_2 = 10;
3 boolean p = FALSE;
4 boolean q = p++;
5 boolean r = valor > valor_2 and p or q
```

Como se puede observar el uso de estos operadores es muy intuitivo. Hay que hacer una aclaración en la **Línea 4**, lo que sucede en esa línea, primero es un cambio de la variable **p** a su negación; y después ocurre una asignación a **q**, por lo que el valor de **q** es TRUE. En esta línea se realizan dos operaciones a la vez, tanto la modificación de la variable **p** tanto como la asignación a la variable **q**. Si no se quisiera modificar el valor de la variable **p**, se tendría que usar el operador negación !

**Observaciones:**

- El operador diferencia no existe. Pero se puede obtener el mismo resultado negando una igualdad.
- El operador aritmético incremento aplicado a un booleano lo niega. Si un booleano es *TRUE* al aplicar *++* será *FALSE*.

### 1.2.3. Otros Operadores

Esta sección, es catalogada de esta forma porque no existe un título general que englobe a los siguientes operadores, a pesar de esto, estos operadores son de igual importancia que los estudiados anteriormente. Son los siguientes:

Operador	Sintaxis
Asignación básica	=
Llamada a función	()
Índice de Array	[]
Miembro	.
Coma	,
Casting	to
Asignación de memoria	size

Ejemplo de codificación: **Observaciones:**

```

1 integer valor = -1;
2 float valor_2 = 10;
3 boolean p = FALSE;
4 boolean q = p++;
5 boolean r = valor > valor_2 and p or q;
6 p++;

```

- No debemos confundir la sintaxis para definir un vector con la sintaxis para acceder a un elemento del vector. Como observamos en la **línea 4** la definición `vector[]` es una convención, mientras que en la **línea 5** estamos haciendo uso del operador `[]` para acceder último elemento del vector.

## 1.3. Operaciones con Cadenas

Las cadenas son un Tipo de Dato especial, pues estas tiene operaciones particulares en las cuales utilizan los operadores ya definidos anteriormente pero con diferentes resultados. Cabe aclarar que el Tipo de Dato String es una **Clase**, con lo cual posee una serie de métodos. A continuación definiremos las operaciones de las cadenas.

### 1.3.1. Concatenación

La concatenación se lleva a cabo con operador de suma (+), esta operación es útil para enlazar cadenas consecutivamente, operación útil para la lectura de archivos. Otra característica a tener en cuenta es que si intentamos concatenar una cadena con otro tipo de dato, se realiza un casting automáticamente; y si intentamos realizar la concatenación con un Objeto, se invoca al método toString.

```
1 integer var = 0;
2 String cadena = "El señor Juan";
3 true(var > 0){
4     cadena = cadena + " está feliz";
5 }false{
6     cadena = cadena + " está despedido";
7 }
8 print(cadena);
```

### 1.3.2. Potencia

Para realizar la potencia de cadenas se utiliza el operador de la multiplicación (\*) seguido de un número entero. Esta operación es útil para encadenar una cadena consigo misma.

```
1 String otra_cadena = "123";
2 print(otra_cadena*3);
```



### 1.3.3. Reflexión

La reflexión es la operación inversa a la potencia, no obstante, solo está definida para valor entero de -1, para otros números negativos, dará un error de compilación. Esta operación es útil para hacer funciones que necesiten del inverso de una palabra, como la función capicua o la función palíndromo;

```
1 String _cadena = "Hola como estás";  
2 print(_cadena*-1);
```

### 1.3.4. Longitud

Si bien esta no es una operación naturalmente, sino un método de la Clase String. Este método es muy útil al operar con cadenas.

```
1 String variable = "";  
2 variable = "Esto es un código de prueba";  
3 print(variable.length());
```

## 1.4. Estructuras de Selección

En vulcanos la estructuras condicionales en Vulcano están compuestos por tres bloques fundamentales: true, also, false. No se debe confundir estas palabras reservadas con los valores booleanos **TRUE** y **FALSE**.

Es necesario especificar que el bloque true funciona independientemente del bloque also y false; sin embargo, estos últimos bloques necesitan de su antecesor para poder funcionar correctamente, es decir, el bloque also necesitará siempre de un bloque true; y el bloque false necesitará siempre de un bloque also o true. Veamos un ejemplo.

```

1 integer variable = 4;
2 true(variable > 3){
3     print("Se ejecutó la sentencia 1");
4 }also(variable > 1){
5     print("Se ejecutó la sentencia 2");
6 }false{
7     true(variable-5 == -5){
8         print("Esto es un bloque anidado");
9     }false{
10         print("Esto es la segunda parte del bloque anidado");
11     }
12 }

```

## 1.5. Estructuras de Selección Múltiple

Para declarar un bloque de selección multiple necesitaremos invocar a la palabra reservada **switch**. Este bloque es fundamental para contraer líneas de código que puedes ser realizadas por los bloques **true**, **also** y **false**. Muchas veces necesitaremos hacer multiples condicionales y el bloque switch es una buena opción. A diferencia de otros Lenguajes este bloque puede recibir dos parámetros, **la variable a comparar** y **el operador de comparación**. En lenguajes como C o C++, las comparaciones se hacen con el operador de igualdad implícitamente y no hay modo de cambiarlo; con este segundo parámetro podremos hacer diversas comparaciones para variables de tipo **integer** y para las cadenas unitarias (Tipo **char** en otros lenguajes).

```

1 integer variable = 3;
2
3 switch(variable, <){
4     case 1: print("Comentario 1"); break;
5     case 2: print("Comentario 2"); break;
6     case 3: print("Comentario 3"); break;
7     case 4: print("Comentario 4"); break;
8     case 5: print("Comentario 5"); break;
9 }

```

En este caso la operación booleana que se ejecuta por cada **case** es variable ¿case (1, 2, 3, 4, 5). La palabra reservada **break** sirve para romper el flujo de ejecución, si en el

código anterior no hubieramos puesto la palabra **break** en cada sentencia, se hubieran el case 3, 4 y 5, pero gracias a la palabra reservada **break**, apenas se ejecuta una instrucción, se rompe el flujo de ejecución.

#### Observaciones:

- Si no definimos el segundo parámetro, el operador por defecto es `==`.

## 1.6. Estructuras Repetitivas

En el Lenguaje Vulcano solo se han definido solamente dos tipos de bucles. El bucle **while** y el bucle **loop**. El bucle **while** se caracteriza por tener un número de ejecuciones indefinido, mientras que el bucle **loop** se caracteriza por recorrer completamente Tipos de datos **secuenciales**. Observemos el siguiente ejemplo.

```
1 integer contador = 0, vector[] = {56, 1, 14, 5, 89};
2 while(contador < 5){
3     print("Hola mundo");
4     contador++;
5     loop(integer i, vector){
6         print(i);
7     }
8 }
```

#### Observaciones:

- El bucle **do while** se puede reemplazar con un **while(true)** y un bloque **true** que contenga un **break** para romper el flujo infinito.
- Siempre que implementemos un bucle **while** es necesario asegurarnos que exista una condición dentro del bucle para poder salir de este.
- Si necesitamos recorrer un arreglo, o realizar cualquier operación como: buscar, desplegar, etc; el bucle **loop** es nuestra mejor opción.

## 1.7. Tipos de datos secuenciales

Es necesario definir los Tipos de Datos Secuenciales, ya que son estos tipos de datos los que el bucle **loop** recorrerá. Los Tipos de Datos Secuenciales definidos en este lenguaje

son: Los **Vectores** y las **Cadenas**.

### 1.7.1. Vectores

Los vectores son un Tipo de Dato secuencial común a todos los lenguajes de programación. Estos se caracterizan por ser una porción de memoria reservada **contigua**. Esta Tipo de Dato es estático por lo que una vez definido ya no cambia, es decir, si se define un vector de tamaño 5, su tamaño no variará a lo largo de la ejecución del programa.

---

```
integer vector[5];
float vector_2[] = {1, 2, 3, 4, 5.3};
integer vector[5][5];
```

---

#### Observaciones:

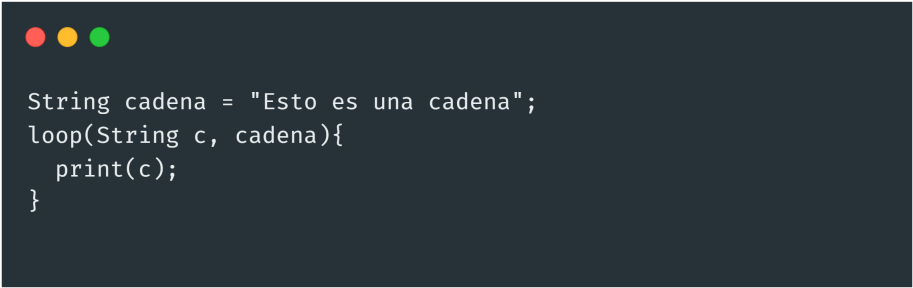
- *Si no se define el tamaño del vector de manera explícita, estaremos obligados a asignar directamente los elementos que contendrá el vector.*
- *La definición de la **línea 3** hace referencia a un vector de vectores. Esta particular manera de declarar un vector, nos sirve para simular la representación de **matrices**.*

### 1.7.2. Cadenas

El Tipo de Dato String ya lo hemos tocado en una sección anterior; sin embargo cabe aclarar que, el bucle **loop** es capaz de utilizar una cadena de la misma forma en la que usa un vector.

## 1.8. Flujos Estándar

En computación existe el término conocido como **Standar Streams** para hacer referencia a los flujos de comunicación entre el hardware y el software. Se definen tres y son: **Standar Input**, **Standar Output** y **Standar Error**.



```
String cadena = "Esto es una cadena";  
loop(String c, cadena){  
    print(c);  
}
```

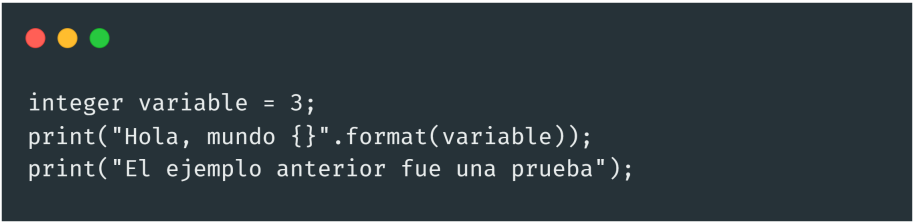
Estos flujos de comunicación nos sirve para enviar y recibir información del programa. Normalmente el **Standar Input** está asociado al teclado, mientras que el **Standar Output** y el **Standar Error** están asociados a la pantalla.

Para hacer uso de estos tres flujos de datos tenemos **tres funciones**: print, input, error.

### 1.8.1. Función print

La función **print** se caracteriza por imprimir cadenas con formato. La sintaxis es la siguiente.

---



```
integer variable = 3;  
print("Hola, mundo {}".format(variable));  
print("El ejemplo anterior fue una prueba");
```

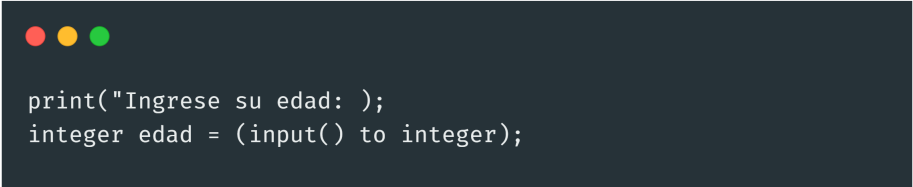
---

#### Observaciones:

- *La función print siempre añade el caracter de escape Salto de línea o Retorno de carro al final de la cadena que imprime.*

### 1.8.2. Función input

La función **input** se caracteriza por pedir una cadena. Una vez invocada a la función, la ejecución del programa se detiene para pedir una función por teclado. como la función **input** retorna una cadena, es necesario aplicar un **casting** para guardar valore en otros tipos de variables. Ejemplo:

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of code:


```
print("Ingrese su edad: ");  
integer edad = (input() to integer);
```

```
print("Ingrese su edad: ");  
integer edad = (input() to integer);
```

### 1.8.3. Función error

La función error, nos servirá para mandar información a la pantalla, cuando sucedan errores en nuestro programa. La sintaxis es la siguiente:

---

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains one line of code:

```
error("Se ha producido un error en nuestro programa");
```

```
error("Se ha producido un error en nuestro programa");
```

## 2. Programación Estructurada

### 2.1. Macros

Las macros son indicaciones que les daremos al compilador. Podemos utilizarlo para reutilizar código fuente a través de la sentencia **#select** o también para emular el uso de constantes a través de la sentencia **#set**. La definición de macros viene a ser la cabeza del código, o sea, la primera parte que se debe escribir. Ejemplo de codificación.

---

```
#select C:\Users\Alejandro\Desktop\programa1.vl

#set 3.1415 PI
```

---

La macros **select** va seguida de la ruta absoluta de un archivo. Todos los archivos que incluyamos con **select** formarán automáticamente parte de nuestro código. El uso de la macros **set** recibe dos parámetros, primero, la palabra que queremos nombres, y segundo, el alias. En el ejemplo visto, lo que realiza la sentencia, es que se reemplazan todas las ocurrencias donde se encuentre PI por 3.1415, esta notación es útil para simular el uso de constantes, cuyo valor debe permanecer inmutable.

El uso de **set** no debe limitarse solo a esto, la idea principal de esta sentencia es crear alias para ciertas sentencias o palabras. Por ejemplo.

```
#set size String("Nueva cadena") nueva_cadena

main(){
    String cad = nueva_cadena;
}
```

En este ejemplo **nueva\_cadena** es solo un alias para referirnos a **size String("Nueva cadena")**.

## 2.2. Funciones

El uso de funciones nos ayudará a tener un código mejor ordenado. La definición de funciones se da en el preámbulo del código fuente, después de las macros y antes de la función principal. Se definen de la siguiente manera, palabra reservada **func** seguido de un

identificador y de sus parámetros encerrados en paréntesis.

---

```
func nueva_funcion(int a, int b){
    true(a > b){
        print("{} es mayor que {}".format(a, b));
    }
    return true;
}
```

### 2.2.1. Funcion main

La función **main** es la función principal que siempre será ejecutada. Es obligatorio implementar esta función, de lo contrario nuestro programa no tendrá razón de ser. Su definición es la siguiente.

```
#select "C:\Users\Juan\Desktop\nuevo_archivo.vl"

func nueva_funcion(int x, int y){
    print("{} - {} es igual a {}".format(x, y, x-y));
}
main(){
    integer entero = 9, entero2 = 17;
    nueva_funcion(entero, entero2);
}
```

### 2.2.2. Comentarios

Cuando empezamos codear es importante **documentar** nuestros avances. Para realizar un comentario, simplemente debemos encerrar nuestras oraciones entre “%”.



```
%El siguiente elemento es un comentario%

main(){
    print("Hola, mundo"); %La sentencia anterior imprime
    "Hola mundo"%
}
```

### 3. Programación Orientada a Objetos

La programación orientada a objetos es un paradigma de programación que nos permitirá realizar programas más ordenados gracias a la relación que existe entre los objetos, los objetos son materializaciones de las **Abstracción** que nosotros como programadores hagamos. Para definir una nueva **Abstracción** hacemos uso de la palabra reservada **Abstract**.

---

```
Abstract{
    %Aquí van los métodos y atributos}%
}
```

#### 3.1. Atributos y Métodos

Los atributos son las características de nuestra Abstracción, estas características pueden ser Tipos de Datos Primitivos u otras Nuevas Abstracciones. Los métodos son aquellas funciones que o comportamientos que realizarán nuestros objetos pertenecientes a dicha Abstracción. Ejemplo de definición de una Abstracción.

```

Abstract Autobus{
%Definición de atributos
integer ruedas;
integer asientos;
String nombrePropietario;
boolean acelerando;
%Definición de métodos
func acelerar(boolean aceptar);
func desacelerar(boolean aceptar);
}

```

### 3.1.1. Método Constructor

El método constructor será el encargado fundamentalmente de instanciar nuevos objetos. Este método es especial, ya que este dará el estado inicial a nuestro objeto. Continuando con la clase Autobus, vista anteriormente, ahora definiremos su método constructor.

#### Observaciones:

- La palabra reservada **self** nos ayuda a diferenciar entre los parámetros de un método (getter, setter o constructor) y los atributos de la Abstracción o Clase.

## 3.2. Encapsulación y Otros Modificadores

Cuando trabajamos con el paradigma orientado a objetos, muchas veces no necesitaremos que una Abstracción o Clase conozca toda la información de otra, o simplemente hay características de mi Clase que solo deben ser manipuladas por esta misma. Para aplicar un modificador a cualquier atributo o método, combinamos cualquiera de las palabras reservadas: **public**, **private** y **static**.

Ejemplo de codificación.

No es necesario definir todos los campos, pero es una buena practica, no agregar un campo si no se va a poner nada.

```

Abstract Autobus{

    %Definición del método constructor
    Autobus(integer ruedas, integer asientos, String
nombrePropietario){
        self.ruedas = ruedas;
        self.asientos = asientos;
        self.nombrePropietario = nombrePropietario;
        self.acelerando = FALSE;
    }

    %Definición de atributos
    integer ruedas;
    integer asientos;
    String nombrePropietario;
    boolean acelerando;

    %Definición de métodos
    func acelerar(boolean aceptar){
        print("Estamos acelerando");
    }
    func desacelerar(boolean aceptar){
        print("Estamos desacelerando");
    }
}

```

### 3.3. Instanciación

Una vez hayamos creado nuestra Abstracción y le hayamos dado la estructura correcta, pasaremos a instanciar ejemplares u objetos de nuestra Abstracción. Primero va el Tipo de Abstracción seguido de un identificador, a esta nueva variable, le asignaremos un nuevo espacio de memoria con la ayuda de nuestro constructor.

### 3.4. Herencia

La herencia es un concepto muy útil para la reutilización de código. Este lenguaje no soporta la herencia múltiple. La sintaxis es la siguiente.

En este ejemplo, hemos asumido que la clase Vehículo ya está definida.

Modificador	Descripción
Public	Otorga acceso maximo al resto de clases
Private	Otorga el mínimo acceso, solo puede ser accesado por la misma clase
Static	Este modificador convierte en atributos o métodos de Clase

### 3.5. Polimorfismo

Este concepto se define de la siguiente manera: Cuando el programa espera un objeto de la Clase, puede recibir un objeto de la Subclase. Esto significa que un objeto que es hijo de una Clase, se puede comportar como su Clase Padre.

## 4. Conclusiones

El lenguaje de programación **Vulcan** ha si do diseñado con una sintaxis amigabale, debido que a tiene elementos de diferentes lenguajes.