Universidad Nacional Experimental Simón Bolívar
Lenguajes de Programación I
Prof. Ricardo Monascal

Caracas, 11 de Abril 2021

Alumno y Carné:
Victoria Torres 12-11468

Examen 3: Pregunta 1

1. Escoja algún lenguaje de programación de alto nivel, de propósito general y orientado a objetos que no comience con J, C o P.

(a) De una breve descripción del lenguaje escogido.

Ruby es un lenguaje de programación interpretado, dinámico, reflexivo y orientado a objetos, considerado de alto nivel y de propósito general que ha sido utilizado por plataformas conocidas como Twitter, Hulu, Github, entre otras, en su desarrollo.

Este lenguaje fue creado por el japonés Yukihiro Matsumoto, quién comenzó a trabajar en él en 1993 y lo presentó públicamente en 1995. Éste combinó una sintaxis inspirada en Python, Perl y algunas características de POO de Smalltalk, pero también se inspiró en algunos de sus lenguajes favoritos como lo son ADA, LISP, LUA, Eiffel, entre otros.

La motivación principal de Matsumoto para crear Ruby fue "enfatizar las necesidades humanas más que las de la máquina" ya que decía que siempre nos ocupamos más de que "la máquina entienda" y luego el programador que se topa con el lenguaje por primera vez sufre mucho tratando de comprender cómo usarlo, por lo cual con Ruby se buscaba fomentar la productividad y diversión del programador tratando de evitar que el lenguaje fuese confuso.

i. Explique la manera de crear y manipular objetos que tiene el lenguaje, incluyendo: constructores, métodos, campos, etc.

En Ruby ocurre algo muy particular y es que TODO es un objeto y se puede manipular como tal. Por ejemplo, los tipos de datos básicos como "String", "Integer", "Symbol", "Array" y "Booleanos" (TrueClass y FalseClass), son todos objetos de sus respectivas clases. Entonces podemos aplicar por ejemplo, métodos sobre ellos. Entonces si por ejemplo en ruby utilizamos el método ".class" para determinar la clase de los objetos que son estos tipos básicos, podemos obtener la clase a la que pertenecen, por ejemplo:

```
42.class        #42 es un numero entero y a su vez es un objeto, si le aplicamos el método ".class"
                #obtenemos la clase a la que pertenece que es "Integer"

"Holis".class   #Obtenemos la clase a la que pertenece que es "String"
true.class      #Obtenemos la clase a la que pertenece que es "TrueClass"
false.class     #Obtenemos la clase a la que pertenece que es "FalseClass"
[2,4,6].class   #Obtenemos la clase a la que pertenece que es "Array"
:foo.class      #Obtenemos la clase a la que pertenece que es "Symbol"
```

Esto hace también que podamos jugar con la forma en la que escribimos operaciones básicas para diferentes tipos de datos, por ejemplo:

```
40 + 2     #Esta es la forma tradicional de escribir la suma, sabemos que nos resultara 42
40.+(2)    #Podemos escribir la suma con la sintaxis de aplicar el "método" suma, de la clase
           #Integer, al objeto 40 y pasándole al objeto 2, esto nos resultara en 42 también.
```

```
"Pokemon" + " Ruby <3"    #Esta es una forma común de concatenar strings que nos resultaría
                          #en un string "Pokemon Ruby <3"
"Pokemon".+(" Ruby <3")   #Aplicando la concatenación como metodo obtendríamos el mismo
                          resultado de "Pokemon Ruby <3"
```

En Ruby para crear objetos utilizamos *clases* que instanciaremos en objetos. Luego cada objeto tiene sus *miembros* de *estado* (atributos, campos, etc) y *comportamientos* (métodos o mensajes). La sintaxis de las clases en Ruby es por ejemplo:

```
#Definimos la clase Lobo
class Lobo
        #Este es el método para inicializar la clase
        def initialize(tipo, nombre, random)
                #Siguen los atributos
                @tipo = tipo
                @nombre = nombre
                @random = self.random    #Este atributo es diferente a propósito para explicar
                                         #una cosita luego
        end

        #Este es el método aullar
        def aullar
                puts
        end

        #Este es el método saludar
        def saludar
                puts "Hola vale, soy el lobo que buscabas, raza #{@tipo}~"
                puts "El nombre es #{@nombre}, para servirle~ *wink wink*"
        end

        #Este es un método random que servirá para explicar una cosita más abajo
        def random
                #Digamos que genera algo random
        end
end
```

Como en Ruby todo es un objeto y podemos crear nuestros propios objetos mediante clases, la forma en la que los manipularemos será a través de los métodos que creemos.

En este ejemplo creamos una clase Lobo que posee un método de inicialización y a su vez métodos que refieren *acciones* que podrán realizar los objetos lobo que creemos. En Ruby normalmente se escriben los métodos como se observa en el ejemplo. Y a diferencia de la sintaxis de lenguajes como por ejemplo Python, vemos que al definirlos no necesitamos utilizar la palabra ***self*** necesariamente.

Por ejemplo, en Python en el inicializador de una clase nos podemos conseguir con:

```
class carro:
        def __init__(self, marca, modelo):
                self.marca = marca
                self.modelo = modelo

        def bocina(self):
                print("piiii piiii")
```

Aquí en Python el *self* ayuda a que los métodos identifiquen sobre cual instancia particular operan utilizándolos como una referencia arbitraria que se pasa como argumento. En el caso de Ruby, como vemos en, por ejemplo, el método *saludar* de la clase lobo, no necesitamos pasarle *self* como argumento puesto que él sabe que operara sobre los objetos de la clase y resuelve, en parte, el problema de saber sobre qué elementos (atributos, etc, de la instancia) podrá trabajar/estar operando utilizando la siguiente sintaxis:

```
var       #Es una variable local
$var      #Es una variable global
@var      #Es una variable de la instancia
@@var #Es una variable de la clase
```

Con esto el lenguaje establece el alcance de las variables, indicándolo desde el nombre de las mismas usando estos operadores y esto es aprovechado por los métodos también. Por ejemplo, en el método *saludar* al hacer #{@tipo} y #{@nombre} al método se le esta infdicando que *nombre* y *tipo* son variables de la instancia y este entonces cuando sea aplicado procederá a utilizarlas.

Sin embargo, es posible conseguirnos en Ruby con *self* pero en el lenguaje suele usarse para referirnos de forma más explicita a un método del objeto, por ejemplo, en nuestra clase Lobo del ejemplo, tenemos el atributo @random = self.random y en este caso lo que estamos haciendo es almacenar el resultado del método *random* en la variable *random*, entonces como tienen el mismo nombre se usa *self* para indicarle a Ruby que estamos refiriéndonos al método y no a la variable.

De hecho, puede ser común ver *self* cuando hay colisiones de nombres entre un metodo y una variable para poder realizar la distinción entre ellos.

Por otra parte, si queremos elaborar nuevas instancias de una clase en ruby tenemos al metodo constructor *new*, entonces si queremos crear objetos de nuestra clase Lobo hacemos lo siguiente:

```
#Creamos objetos Lobo con new
l = Lobo.new('Gris', 'Canis Lupus')
```

Una vez creado el objeto de la clase podemos utilizar los métodos de la misma para hacer que el lobo aulle y salude escribiendo:

```
#Hacemos que el lobo aulle y salude
l.aullar    #Se hará un put de "Auuuuuu"

l.saludar #Se hará un Puts de:
```

```
#“Hola vale, soy el lobo que buscabas, raza Gris~”
#“El nombre es Canis Lupus, para servirle~ *wink wink*”
```

Pero Ruby también posee *constructores literales,* entonces podemos usar una notación especial en vez de usar *new* para crear un nuevo objeto de la clase, para eso presentamos la siguiente tablita donde podemos verlos resumidos:

Clase	Constructor Literal	Ejemplo
String	' ó "	`"nuevo string"` o `'nuevo string'`
Símbolo	:	`:símbolo` ó `:"símbolo con espacios"`
Array	[ ]	[1, 2, 3, 4, 5]
Hash	{ }	`{"Nueva Yor" => "NY", "Oregon" => "OR"}`
Rango	.. ó ...	`0…10` ó `0..9`
Expresiones regulares	/	`/([a-z]+)/`

Y una vez creado el objeto de la clase, podemos aplicarle otros métodos, que no son necesariamente de la clase, para manipularlos, entonces podemos hacer cosas como por ejemplo:

```
if l.respond_to?(“saltar”)
        l.saltar
else
        puts “Lo siento, el lobo no sabe que es ‘correr’.”
end
```

Aquí usamos el metodo *respond_to?*, que es un método de Ruby que nos permite conocer de antemano si un objeto posee cierto método, entonces el *if* chequea si el lobo tiene el método *saltar*, si lo tiene entonces el lobo salta pero sino entonces se imprime el mensaje. En este caso por como está definida la clase lobo, nuestro objeto lobo no sabe saltar y por lo tanto obtendremos la impresión “Lo siento, el lobo no sabe que es ‘correr’.”

ii. Describa el funcionamiento del manejo de memoria, ya sea explícito (new/delete) o implícito (recolector de basura).

En Ruby el manejo de memoria es implícito y se utiliza un recolector de basura.

En el lenguaje, las variables locales viven en el stack (se reclama el espacio cuando el método retorna) y los objetos viven en el heap, estos se crean al momento con el uso del método “new” previamente mencionado.

Por otro lado, los objetos como tal nunca se liberan explícitamente, sino que se gestiona la memoria de forma automática con un recolector de basura que la recupera.

El recolector de basura de Ruby es del tipo *marcar y borrar* (mark-and-sweep). En la fase *mark* el programa recicla memoria, verifica si el objeto esta en uso. Si un objeto es apuntado por una variable, podría ser usado, y por tanto ese objeto se marca para ser conservado. Si no hay variable que apunte al objeto, entonces el objetos no se marca y entonces en la fase *sweep* se borran los objetos en desuso para liberar memoria de modo que pueda volver a ser utilizada por el interprete de Ruby.

Además, el mark-and-sweep de Ruby es *conservador* de modo que no hay garantía de que un objeto sea eliminado por el colector antes de que termine el programa. Si almacenas algo en un array, y se mantiene el array, todo dentro del array es marcado. Si almacenas algo en una constante o variable global , entonces esto queda marcado para siempre.

Si para el mismo ejemplo de nuestra clase lobo (en donde habíamos creado al objeto *l* de la clase), hiciéramos lo siguiente:

```
l1 = l       #Con la variable l1 apuntamos al mismo objeto l que creamos anteriormente
l1.aullar  #De esta manera podemos entonces aplicarle el método aullar resultaría “Auuuuuu”

l = nil      #Luego, al hacer esto hacemos que el objeto lobo l apunte a nil, lo que significa que no
             #se refiere a nada

l1 = nil     #Y luego, al hacer esto entonces el objeto Lobo l que habíamos creado dejara de estar
             #apuntado por las variables y será objetivo del recolector de basura.
```

Con el ejemplo anterior podemos observar que el objeto será objetivo del recolector de basura ya que no estará marcado, puesto que no hay variables que apunten a él (esto porque como se explicaba antes del ejemplo, el recolector de basura de Ruby es del tipo *mark-and-sweep).*

iii. Diga si el lenguaje usa asociación estática o dinámica de métodos y si hay forma de alterar la elección por defecto del lenguaje.

Ruby solo tiene asociación dinámica de métodos, esto quiere decir que la decisión del tipo dependerá del tipo dinámico y se establece a tiempo de ejecución. Como tal no podemos alterar la elección por defecto del lenguaje.

Un ejemplo (super original y para nada sacado de la clase 16 de lenguajes de programación I) en Ruby de esto es por ejemplo que tengamos:

```
class Base
    def p
        puts “Base”
    end
end

class A < Base
    def p
        puts “A”
    end
end

class B < Base
    def p
        puts “B”
    end
end
```

```
a = A.new
b = B.new
c = Base.new

a.p   #Imprime A
b.p   #Imprime B
c.p   #Imprime Base
```

Aquí, vemos que como es de esperarse, como cada objeto corresponde a las clases A, B y Base respectivamente, el método *p* para cada objeto nos imprime A, B y Base respectivamente. Sin embargo, si nosotros agregamos lo siguiente:

```
a = A.new
b = B.new
c = Base.new
d = A.new

c = d

a.p   #Imprime A
b.p   #Imprime B
c.p   #Imprime A
```

Para este caso vamos a obtener que *c.p* nos imprime *A* y esto es precisamente por la asociación dinámica de métodos en Ruby, ya que como le estamos asignando al objeto *c*, el objeto *d* (que es un objeto de la clase A) entonces en tiempo de ejecución el tipo de *c* es tomado como un objeto *A* y por eso vemos que se imprime este resultado.

Con este tipo de asociación el lenguaje aprovecha mas el polimorfismo de subtipos.

iv. Describa la jerarquía de tipos, incluyendo mecanismos de herencia múltiple (de haberlos), polimorfismo paramétrico (de tenerlo) y manejo de varianzas.

Debemos recordar que en Ruby TODO es un objeto. Al inicio se comentaba que esto hacia que entonces en el lenguaje los tipos (por ejemplo Integer, String, Array, etc) fuesen clases y que elementos de un tipo concreto entonces a su vez fuesen instancias u objetos de esas clases en cuestión.

Por ejemplo, el numero entero 42 (que sabemos que es del tipo Integer) es un objeto de la clase Integer respectivamente.

De esta forma, la jerarquía de tipos en Ruby es comúnmente representada como una jerarquía de clases donde:

-El padre de todos los padres de todas las clases se llama ***BasicObject*** que es básicamente una clase vacía de la que se extienden el resto de clases en Ruby. Esta tiene un conjunto de solo 13 métodos que son : new; ! ; != ; == ; __id__ ; __send__ ; equal? ; instance_eval ; instance_exec ; method_missing ; singleton_method_added ; singleton_method_removed y singleton_method_undefined.

De esta manera nosotros entonces no podemos crear instancias de BasicObject en, por ejemplo, el interprete de Ruby IRB ya que esta no tiene un método *inspect*. Entonces, como tal el objetivo de esta clase es ofrecer una especie de lienzo en blanco, es decir, una manera de que podamos crear objetos sin tener que contender con comportamientos predeterminados y a su vez pone a disposición del resto de clases por debajo en la jerarquía, sus métodos. Así que por ejemplo, cuando usamos el método *.new* para crear un nuevo objeto de una clase (ya sea una de Ruby o una creada por nosotros), estamos en realidad llamando al método *new* de BasicObject.

- En la jerarquia normalmente se suele representar que viene inmediatamente la clase ***Object,*** sin embargo si en el IRB vemos los ancestros de Object nos encontraremos con lo siguiente:

  irb(main):006:0> Object.ancestors

  => [Object, Kernel, BasicObject]

  Entonces, antes de pasar a Object, hablemos de ***Kernel,*** quien no es como tal no es una clase sino un modulo que incluido dentro de la clase Object y de hecho gran parte de las funcionalidades de esta clase vienen de dicho modulo. Kernel contiene 66 métodos definidos de los cuales podemos mencionar puts, gets, chomp y exit (todos métodos que entonces podremos usar en Object y todo lo que venga debajo en la jerarquía). De hecho, estos métodos son tan comunes que se piensa que son palabras claves del lenguaje, pero no, son métodos del modulo Kernel.

  Sin embargo, algo interesante a notar es que mucho de los metodos de Kernel sono privados entonces no pueden llamarse con un *receptor*. Por ejemplo, si intentamos llamar el metodo *puts* en una instancia de *Array* obtendremos lo siguiente:

  array = ["ejemplito","Ruby"]

  array.puts #NoMethodError (private method `puts' called for ["ejemplito", "Ruby"]:Array)

Esto nos permite ver que entonces, en el caso de puts, aunque esta disponible para cualquier objeto en Ruby, solo puede ser llamado sobre el *self* que se tenga en el contexto en el que se encuentre.

- Ahora si, podemos mencionar entonces la siguiente clase en la jerarquía, que es ***Object***. Esta es el padre por default de todas las clases en Ruby. Posee 49 métodos (propios y adicionales a los que incluimos del módulo Kernel) y entre estos podemos mencionar class; is_a ; methods ; send ; nil? ; inspect y object_id. Algo curioso que vale la pena mencionar es que cuando definimos un metodo de alto nivel (es decir, que no esta dentro de una clase o modulo) lo que estamos haciendo es crear la instancia de un metodo privado de la clase Object y como todas las clases en Ruby heredan de Object, entonces por esto estos metodos que definimos pueden ser llamados en cualquier parte de nuestro programa siempre y cuando los estemos llamando sobre un *self* sin colocarles un *receptor* explicito (tipo el

ejemplo de antes donde hacíamos array.puts, donde vemos que no podemos usarlo así sino que simplemente debemos colocar puts) .

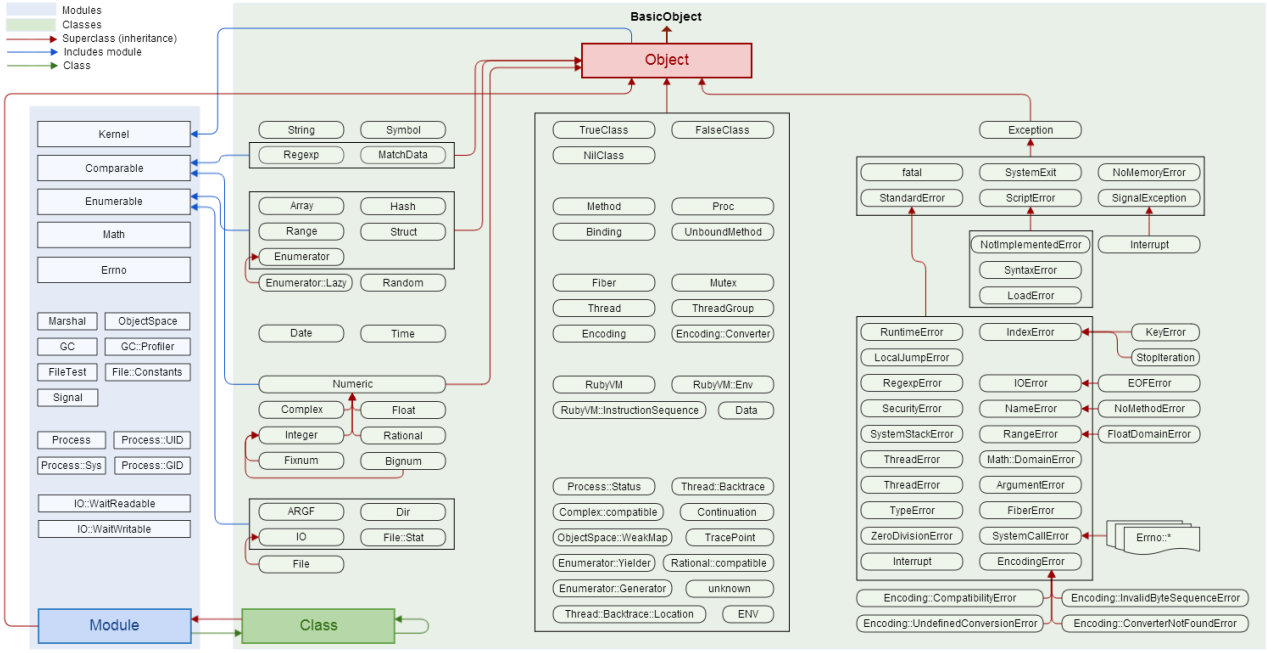
Estas tres bases son las que definen principalmente la jerarquia del tipos en Ruby, sin embargo vale la pena mencionar la existencia de la clase ***Class***, quien esta por debajo de Object en la jerarquia, y la magia oscura que ocurre en Ruby si nos adentramos en las profundidades de sus sistema jerárquico.

Resulta que todas las clases en Ruby son subclases de la clase *Class* y esto incluye a Object. Si, ***Object*** es una instancia de la clase ***Class***… pero dijimos que *Class* es una subclase de *Object* en la jerarquía, por lo cual entonces es como si *Object* fuese esencialmente una subclase de si misma \(@A@)/. Esto de hecho podemos verlo en el IRB si hacemos lo siguiente:

```
irb(main):001:0> Object.ancestors

=> [Object, Kernel, BasicObject]

irb(main):002:0> Class.ancestors

=> [Class, Module, Object, Kernel, BasicObject]

irb(main):003:0> Object.class

=> Class

irb(main):004:0> BasicObject.class

=> Class

irb(main):005:0> Kernel.class

=> Module
```

De hecho, hasta podemos ver que tanto Object, como BasicObject son ancestros de Class pero a su vez la clase de ambas es, a su vez, Class. Y pues literalmente, la explicación más coherente que podemos encontrar de esto es… **brujería** (xd).

A continuación, tenemos una imagen que nos muestra gráficamente, de forma general, las relaciones jerárquicas de las clases en Ruby:



Ahora si hablamos sobre la herencia, en Ruby cada clase solo puede tener una superclase de la que herede comportamientos, es decir, tenemos herencia simple y su sintaxis es, como vimos anteriormente en el ejemplo de asociación de métodos dinámica, con el uso de "<", es decir:

```
class Superclase
    def el_metodo
        puts "El papix"
    end
end
```

```
class LaQueHereda < ClasePadre
    def metodo
        puts "Dame los riales"
    end
end

a = LaQueHereda.new
a.metodo                #Imprime "Dame los riales"
a.el_metodo             #Imprime "El papix"
```

Observamos en el ejemplo que entonces al crear una instancia de la clase que hereda podremos utilizar sobre esta tanto métodos de su clase como métodos de la clase heredada.

En Ruby no tenemos como tal herencia múltiple sino solo simple, sin embargo una manera de "simular" la herencia múltiple es por medio de mixin utilizando módulos incluyéndolos en la clase. Por ejemplo:

```
module A
        def un_metodo_pro
                puts "pro"
        end
        def otro_metodo_pro
                puts "otro pro"
        end
end
module B
        def que_pro_vale
                puts "que pro"
        end
        def wow_otro_beta
                puts "es otro beta"
        end
end
class Normalita
      include A
      include B
      def metodo_normalito
      end
end

print Normalita.ancestors   #Imprime sus ancestros [Normalita, B, A, Object, Kernel, BasicObject]
n = Normalita.new           #Creamos objeto de la clase normalita
n. un_metodo_pro            #Imprime "pro"
n. otro_metodo_pro          #Imprime "otro pro"
n. que_pro_vale             #Imprime "que pro"
n. wow_otro_beta            #Imprime "es otro beta"
```

En el ejemplo tenemos los módulos A y B, que tienen métodos respectivamente, y luego en la clase Normalita los incluimos, esto hace que entonces ahora todo objeto de la clase Normalita pueda acceder a los métodos de los módulos y utilizarlos. Con esta forma de realizar mixin en Ruby se suele decir que simulamos herencia múltiple porque es como si la clase Normalita, que incluye los módulos, estuviese heredando de los mismos. De hecho en el mismo ejemplo podemos ver como los módulos A y B se convirtieron en ancestros de la clase Normalita.

Algo curioso además, es que en Ruby no tenemos ni interfaces ni clases abstractas pero podemos modelarlas de distintas formas y una de estas es precisamente con módulos. Ya que lo que podríamos hacer es crear un "modulo abstracto" con los métodos no implementados, incluirlos luego en las clases que será como su "subclase" (podemos verlo así por lo hablado anteriormente) y en dicha

subclase implementar como tal los métodos en cuestión. Veremos un ejemplo de esto en la pregunta "(b)" sección "i" que se encuentra en el archivo "sec_pila_cola.rb" en esta misma carpeta de la "Pregunta 1" :3

Ahora si hablamos de polimorfismo paramétrico recordamos que este se refiere a cuando escribimos código, por ejemplo hacemos un método, sin mencionar algún tipo específico, de manera que entonces ese código podamos utilizarlo con cualquier numero de tipos diferentes de forma transparente. En algunos lenguajes por ejemplo se utilizan los *genéricos* para crear código que responda de esta forma. La cosa en Ruby es que el lenguaje es de *tipo dinámico* y este comportamiento del polimorfismo paramétrico es más característico de los lenguajes con *tipo estático*.

Además, está el hecho de que en Ruby mencionamos antes que TODO es un objeto, entonces los "Tipos" en el lenguaje corresponden a clases que son a su vez clases de otras clases, etc, por lo cual técnicamente decimos que tenemos "tipos" pero *legalmente* estos son clases, objetos, etc, de manera que todo este concepto de polimorfismo paramétrico pierde sentido realmente y no tiene mucho sentido en el lenguaje.

Sin embargo, podemos conseguir cosas como que se pueden pasar clases a un método como si fuesen objetos haciendo, por ejemplo:

```
def crear_objeto(clase, *args)
        clase.new(*args)
end
puts crear_objeto(String) #Nos imprime el objeto String vacío: ""
puts crear_objeto(Hash)   #Nos imprime el objeto Hash, diccionario vacío: {}
print crear_objeto(Array,3,:el_simbolo)#Imprime el arreglo[:el_simbolo,:el_simbolo,:el_simbolo]
```

Aquí prácticamente se simula esto de que para un objeto de cada tipo el método se adapta al mismo y crea el objeto que corresponde a la clase que se le paso.

Sin embargo, lo que ocurre es que Ruby usa *duck typing*, de manera que entonces podemos pasar argumentos de cualquier clase a cualquier método. En Ruby no se declaran los tipos de variables o métodos por este hecho de que todo es un objeto y todos estos pueden ser modificados (ya que siempre pueden añadírsele métodos nuevos a la clase a posteriori). En este lenguaje, con el *duck typing* nos fijamos menos en el tipo o clase del objeto y mas en sus capacidades, es decir, le damos prioridad a su comportamiento: que métodos se pueden usar y que operaciones se pueden hacer con el .

Entonces en Ruby podemos tener por ejemplo cosas como:

```
puts 'holi'.respond_to? :to_str  #Imprimirá true
puts 4.respond_to? :to_str       #Imprimirá false
```

En este ejemplo, comprobamos si los objetos responden al método “to_str”, observamos que el string ‘holi’ nos retorna true y el entero 4 false. Con este ejemplo podemos ver de manera muy simple la “filosofía” del *duck typing* conocida comúnmente con la frase “si un objeto hace quack como pato, entonces trátalo como pato”, en el caso del ejemplo “si un objeto actúa como string, entonces trátalo como string”.

Por lo tanto, en Ruby tratamos a los objetos por lo que pueden hacer y no como tal por su tipo, o en este caso más específicamente por las clases de las que proceden, o los módulos que incluyen.

Finalmente si tenemos que hablar de las *varianzas de tipos* lo primero que tenemos que tener en mente es que los lenguajes de tipado dinámico , como Ruby, no piden que el programador tenga que especificar la varianza de tipos, por lo cual quienes programan con este tipo de lenguajes terminan prácticamente desarrollando un “instinto de cuáles usos son aceptables y cuales no (aja, es desarrollado pero no estamos diciendo que sea 100% efectivo o correcto xd eso depende de cada quien y de cada caso).

Un ejemplo clásico para ver como se comporta la varianza en lenguajes de programación es en el que tenemos una clase **Animal** y una subclase de esta llamada **Perro** entonces nos preguntamos *¿ una lista de perros seria subclase de una lista de animales?*

Para ese caso sabemos que, en teoría, en donde pueda usar cosas del supertipo, deberíamos poder usar algo del subtipo y todo lo que pueda hacer sobre el supertipo yo debería poderlo hacer sobre el subtipo, pero en el caso planteado a una lista de animales yo le puedo agregar, por ejemplo, un *gato* sin embargo a una lista de perros no.

Este problema en particular lo estamos considerando para una List<Animal> y una List<Perro>, sin embargo, en Ruby mencionamos antes que realmente no tratamos a los objetos por tipo sino por lo que pueden hacer de manera que generar una lista parametrizando sus elementos de esta manera no es algo que el lenguaje nos provee a priori y como al final todo es tratado como objeto entonces para modelar algo de esta manera hay que “ponerse creativo” para hacer la parametrización, por ejemplo:

Si quisiéramos simular un método parametrizado, como el que sigue de Java:

```
void un_metodo<Animal> (Animal animalito) { //Hace algo}
```

Entonces en Ruby tendríamos que hacer algo como lo siguiente:

```
def un_metodo(animalito)
    raise “Solo funciona con ‘tipo’ Animal ” unless animalito.is_a? Animal
    #El método hace alguna cosa
end
```

```
class Animal
    def _requerido_para_el_metodo()
        raise “Animal es abstracto”
    end
end
class Perro < Animal
    def _requerido_para_el_metodo()
        #Hace algo
    end
end
class Gato < Animal
    def _requerido_para_el_metodo()
        #Hace algo
    end
end
```

En este ejemplo, tuvimos que crear el método y *levantar* una verificación para que se ejecute solo si lo que le pasamos *es un* objeto de *Animal*. Pero entonces luego la “magia” para que esto simule lo que queremos vendrá de tener que definir el posible conjunto de Animales que el método podría recibir creando entonces las subclases relacionadas para que entonces cuando se verifique que el objeto que recibe un método es Animal, entonces la cosa funcione.

Sin embargo, como podemos observar, aplicando este tipo de cosas podemos simular el caso para ver la varianza de tipos en Ruby, sin embargo como estaríamos construyendo todo nosotros entonces, por ejemplo de esta forma sabemos que el lenguaje podría quejarse si de repente tuviésemos una “Lista de Perros” y le intentamos agregar un *gato*, puesto que si hiciéramos un método para agregar elementos a la lista de Perros pero condicionándolo a que verifique que el objeto sea un perro antes de agregarlo a la lista (usando algo similar al raise “Solo funciona con ‘tipo’ Perro ” unless valor.is_a? Perro), entonces sabemos que la implementación no permitirá que agreguemos al *gato* o de repente si la condición la ponemos general como raise “Solo funciona con ‘tipo’ Animal ” unless valor.is_a? Animal), entonces podremos agregar gatos a la lista y no habrá problema pero ya es porque prácticamente estamos decidiendo que se haga de esa forma.

A pesar de estas construcciones vemos que como tal Ruby no toma en cuenta a priori la varianza de tipos ya que por como es este no es algo que tenga tanto sentido dentro del lenguaje, sin embargo podemos simular esta utilizando este tipo de opciones pero no estamos viendo como Ruby como

lenguaje maneja la varianza de tipos como tal ya que el control realmente lo tiene el programador, que hace toda la construcción.

PD: El meme solo funciona si no ha visto aun el video… así que si, por casualidad, ya lo vio…