Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3661 – Laboratorio de Lenguajes de Programación
Trimestre: Septiembre - Diciembre 2023
Profesor: Ricardo Monascal
Estudiante: Junior Miguel Lara Torres, Carnet: 17-10303

Tarea 2: Programación Orientada a Objetos y Ruby (20 pts)

IMPLEMENTACION

- Parte 1: (4 puntos) Propiedades y herencia.

  - Considere una clase Circulo, con un único campo radio.

```ruby
"Parte 1.1.a"
class Circulo
    @radio
end

"Parte 1.1.b"
class Circulo
    def radio
        @radio
    end
    def radio=(valor)
        if valor < 0
            raise "Radio Invalido"
        else
            @radio = valor
        end
    end
end
```

```ruby
"Parte 1.1.c"
class Circulo
    def radio
        @radio
    end
    def radio=(valor)
        if valor < 0
            raise "Radio Invalido"
        else
            @radio = valor
        end
    end
    def initialize(valor)
        if valor < 0
            raise "Radio Invalido"
        else
            @radio = valor
        end
    end
end
```

```ruby
"Parte 1.1.d"
class Circulo
    def radio
        @radio
    end
    def radio=(valor)
        if valor < 0
            raise "Radio invalido"
        else
            @radio = valor
        end
    end
    def initialize(valor)
        if valor < 0
            raise "Radio Invalido"
        else
            @radio = valor
        end
    end
    def area
        3.14159*@radio**2
    end
end
```

- Considere una subclase Cilindro de Circulo, que agrega un único campo altura.

```ruby
"Parte 1.2.a"
class Cilindro < Circulo
    @altura
end

"Parte 1.2.b"
class Cilindro < Circulo
    def altura
        @altura
    end
    def altura=(valor)
        if valor < 0
            raise "Altura invalida"
        else
            @altura = valor
        end
    end
end
```

```ruby
"Parte 1.2.c"
class Cilindro < Circulo
    def altura
        @altura
    end
    def altura=(valor)
        if valor < 0
            raise "Altura invalida"
        else
            @altura = valor
        end
    end
    def initialize(radio, altura)
        if radio < 0
            raise "Radio invalido"
        elsif altura < 0
            raise "Altura invalida"
        else
            @altura = altura
            @radio = radio
        end
    end
end
```

```ruby
"Parte 1.2.d"
class Cilindro < Circulo
    def altura
        @altura
    end
    def altura=(valor)
        if valor < 0
            raise "Altura invalida"
        else
            @altura = valor
        end
    end
    def initialize(radio, altura)
        if radio < 0
            raise "Radio invalido"
        elsif altura < 0
            raise "Altura invalida"
        else
            @altura = altura
            @radio = radio
        end
    end
    def volumen
        area*@altura
    end
end
```

- Parte 2: (4 puntos) Defina una clase Moneda con subclases Dólar, Yen, Euro, Bolívar y Bitcoin.

  Donde las tasas de conversión son las siguientes:
  1 dólar -> 146.81 yenes
  1 dólar -> 0.91 euros
  1 dólar -> 35.43 bolívares
  1 dólar -> 0.000026 bitcoins
  1 yen -> 0.0062 euros
  1 yen -> 0.24 bolívares
  1 yen -> 0.00000018 bitcoins
  1 euro -> 38.93 bolívares
  1 euro -> 0.000029 bitcoins
  1 bolívar -> 0.00000074 bitcoins

- a)

```ruby
class Moneda
    attr_accessor :dolar, :yen, :euro, :bolivar, :bitcoin
end
class Dolar < Moneda
    def initialize(valor)
        valor < 0 ? (raise "Moneda invalida") : (@dolar = valor)
    end
end
class Yen < Moneda
    def initialize(valor)
        valor < 0 ? (raise "Moneda invalida") : (@yen = valor)
    end
end
class Euro < Moneda
    def initialize(valor)
        valor < 0 ? (raise "Moneda invalida") : (@euro = valor)
    end
end
class Bolivar < Moneda
    def initialize(valor)
        valor < 0 ? (raise "Moneda invalida") : (@bolivar = valor)
    end
end
class Bitcoin < Moneda
    def initialize(valor)
        valor < 0 ? (raise "Moneda invalida") : (@bitcoin = valor)
    end
end

class Float
    def dolares
        Dolar.new(self)
    end
    def yens
        Yen.new(self)
    end
    def euros
        Euro.new(self)
    end
    def bolivares
        Bolivar.new(self)
    end
    def bitcoins
        Bitcoin.new(self)
    end
end
```

- b) Se sobre entiende que la siguiente imagen posee la parte (a), solo que por temas de espacio solo se muestra lo pedido en el inciso.

```ruby
class Moneda
    attr_accessor :dolar, :yen, :euro, :bolivar, :bitcoin
    def en(atomo)
        if @dolar != nil
            case atomo
            when :dolares
                @dolar
            when :yens
                @@dolar*146.81
            when :euros
                @dolar*0.91
            when :bolivares
                @dolar*35.43
            when :bitcoins
                @dolar*0.000026
            else
                raise "Cambio invalido"
            end
        elsif @yen != nil
            case atomo
            when :dolares
                @yen/146.81
            when :yens
                @yen
            when :euros
                @yen*0.0062
            when :bolivares
                @yen*0.24
            when :bitcoins
                @yen*0.00000018
            else
                raise "Cambio invalido"
            end
        elsif @euro != nil
            case atomo
            when :dolares
                @euro/0.91
            when :yens
                @euro/0.0062
            when :euros
                @euro
            when :bolivares
                @euro*38.93
            when :bitcoins
                @euro*0.000029
            else
                raise "Cambio invalido"
            end
        elsif @bolivar != nil
            case atomo
            when :dolares
                @bolivar/35.43
            when :yens
                @bolivar/0.24
            when :euros
                @bolivar/38.93
            when :bolivares
                @bolivar
            when :bitcoins
                @bolivar*0.00000074
            else
                raise "Cambio invalido"
            end
        elsif @bitcoin != nil
            case atomo
            when :dolares
                @bitcoin/0.00026
            when :yens
                @bitcoin/0.00000018
            when :euros
                @bitcoin/0.000029
            when :bolivares
                @bitcoin/0.00000074
            when :bitcoins
                @bitcoin
            else
                raise "Cambio invalido"
            end
        end
    end
end
```

- c) Se sobre entiende que la siguiente imagen posee la parte (a) y (b), solo que por temas de espacio solo se muestra lo pedido en el inciso.

```ruby
class Moneda
    attr_accessor :dolar, :yen, :euro, :bolivar, :bitcoin
    def en(atomo)
        (...)
    end
    def comparar(moneda)
        sentence = nil
        if @dolar != nil
            sentence = @dolar <=> moneda.en(:dolares)
        elsif @yen != nil
            sentence = @yen <=> moneda.en(:yens)
        elsif @euro != nil
            sentence = @euro <=> moneda.en(:euros)
        elsif @bolivar != nil
            sentence = @bolivar <=> moneda.en(:bolivares)
        elsif @bitcoin != nil
            sentence = @bitcoin <=> moneda.en(:bitcoins)
        end
        case sentence
        when -1
            :menor
        when 0
            :igual
        when 1
            :mayor
        end
    end
end
(...) # Subclases y class Float.
```

- Parte 3: (4 puntos) - Bloques e iteradores.

Esta función Map retorna elemento a elemento del mapeo, por tanto, se usa con una función lambda por ejemplo que manipule dichos retornos.

```
irb(main):009:0> Map([:a, :b, :c], [4, 5]) {|x| puts "#{x}"}
[:a, 4]
[:a, 5]
[:b, 4]
[:b, 5]
[:c, 4]
[:c, 5]
=> 0..2
```

```ruby
def Map(c1, c2)
    if c1.is_a? Array and c2.is_a? Array
        for x in (0 .. c1.length-1) do
            for y in (0 .. c2.length-1) do
                yield [c1[x], c2[y]]
            end
        end
    else
        raise "Algun argumento no es una lista."
    end
end
```

INVESTIGACION

- Parte 1: (4 puntos) Considere un lenguaje de programación puramente orientado a objetos, donde una clase **B** hereda de otra clase **A** (esto es, **B** es subclase de **A**).

  a) Considere una clase **Lista**, parametrizable en el tipo de sus elementos.

  - ¿Qué relación de herencia, de haberla, tienen **Lista<A>** y **Lista<B>**?

  La herencia de clases permite que una clase hija herede los métodos de la clase padre, incluyendo aquellos que la clase padre haya heredado de otras clases.

  Sin embargo, en el caso de las listas parametrizables, como **List<A>** y **List<B>,** no existe una relación de herencia entre ellas. Cada parametrización de la clase Lista se considera como una clase independiente, por lo que **List<B>** no hereda de **List<A>.** A lo máximo que se puede llegar es que los elementos de **List<B>** poseen herencias de los elementos de **List<A>** ya que **B** hereda a **A** por default digamos.

  - ¿Qué decisión toma el lenguaje Java? Explique dicha decisión.

  En Java, los tipos genéricos son invariantes por default, lo que significa que un tipo genérico **B** no puede ser asignado a un objeto de tipo genérico **A**, a menos que **B** sea una subclase de **A** pero definiéndolo de manera específica, esto quiere decir que en Java se debe escribir de manera explícita "`B ? extends A`" indicando así que entre **A** y **B** se admite la covarianza por lo que entre **List<A>** y **List<B>** existe la relación de herencia por covarianza.

  - ¿Qué decisión toma el lenguaje Scala? Explique dicha decisión.

  "La covarianza quiere decir que los únicos elementos aceptados son aquellos que son de tipo **A** o cualquier subtipo de **A**. Por ejemplo, las listas en Scala son covariantes.

```
abstract class Vehicle

case class Car() extends Vehicle

case class Motorcycle() extends Vehicle

val vehicles: List<Vehicle> = List(Car(), Motorcycle())
```

  "

  (Mellado, 2018)

  Con este citado, tenemos que si en particular **B** es subclase de **A** entonces podemos decir existe covarianza entre **List<B>** a **List<A>.**

Sin embargo, tanto en la documentación oficial de Scala como en este artículo escrito por *Mellado*, podemos leer que Scala permite elegir entre la contravarianza, covarianza e invarianza de tipos usando **-A**, **+A** o **A** respectivamente siendo **A** un tipo genérico, con esto indicamos que categoría toma el tipo.

Así mismo, la documentación oficial de Scala afirma que

"*You will encounter covariant types a lot when dealing with immutable containers, like those that can be found in the standard library (such as List, Seq, Vector, etc.).*"

(Variance, 2023)

Esto dice que las listas en Scala están implementadas por defecto como `class List[+A]…` Lo que da sentido al ejemplo mostrado anteriormente.

- Suponiendo que existe una clase **Bottom**, la cual es subclase de todas las demás clases: ¿Cuál es el tipo inferido de la lista vacía? Justique su respuesta.

Si existe una clase **Bottom** que es subclase de todas las demás clases, entonces el tipo inferido de la lista vacía debe ser **Lista[Bottom].** Esto se debe a que la lista vacía no tiene elementos, por lo que su tipo debe ser compatible con cualquier clase padre, en particular **A** y **B**, pues si decimos que lista vacía infiere a **Lista[A]** por ejemplo, siendo **A** la superclase de **B** y **Bottom**, el tipo no sería compatible con las subclases por el mismo argumento que venimos trabajando de que **A** es algo general, **B** es algo especifico de **A** y aún Bottom específica mas que A y B, entonces el tipo de **A** no cuadra con **Bottom** o **B**. En conclusión, como Bottom es una subclase de todas las demás clases, **Lista[Bottom]** es compatible con cualquier tipo de elemento, en particular la lista vacía.

PD: Triki Question: Si **A** es papa de **B**, y Bottom es hijo de **A** y **B** ¿quiere decir que Bottom es hermano e hijo de **B**?...

b) Considere ahora que el lenguaje de programación en el que se está trabajando es funcional. Como es puramente orientado a objetos, las funciones también deben ser objetos. Suponga otra clase cualquiera C.

- ¿Qué relación de herencia, de haberla, tienen las funciones con firmas A -> C y B -> C? Justique su respuesta.

Según la teoría que se explica en la Ciencia de la Computación para covarianzas y contravarianzas, si tiene una función que recibe un entero, digamos F_int, y una función que recibe un float, F_float, donde Integer es subclase de Float, y si podemos usar la función F_float en lugar de F_int entonces se dice que las funciones son contravariantes en sus argumentos.

En nuestro caso particular, como se tiene que B es subclase de **A**, lo que intuitivamente puede verse como que el conjunto **A** contiene a **B**, pero no necesariamente B contiene a **A**, entonces, tendríamos en este sentido una función **A->C** digamos "Grande" que tiene como dominio elementos en **A**, pero si **B** es subclase de **A**, entonces dicha función "Grande" admite valores de **B** en particular, y claro el caso inverso no puede pasar ya que no existe seguridad que la función **B->C** admita todos los elementos de **A**, esto define como una relación de ***CONTRAVARIANZA*** en los argumentos de las funciones.

- ¿Qué relación de herencia, de haberla, tienen las funciones con firmas **C->A** y **C->B**? Justique su respuesta.

Se tiene una función que retorna un entero, digamos F_int, y una función que retorna un float, F_float, donde Integer es subclase de Float, entonces según la teoría que se explica en la Ciencia de la Computación para covarianzas y contravarianzas, se dice que F_int es subclase de F_float, lo que afirma que son funciones con covarianza en los valores de retorno.

En nuestro caso particular, como se tiene la función **C->A** quiere decir que esta función retorna tipo **A**, lo mismo para **C->B** retorna tipo **B**, y como inicialmente B es subclase de **A**, quiere decir entonces que las funciones tienen una relación ***COVARIANZA*** en los valores de retorno.

- Parte 2: (4 puntos) En Ruby existen dos tipos de jerarquía: una basada en herencia y otra basada en instanciación.

  - a) ¿Cuáles son las raíces para las jerarquías de herencia e instanciación?

En Ruby, podemos empezar a probar aplicando .**superclass** y .**class** a todo lo que conozcamos y veremos que eventualmente terminamos en **Class**, que a su vez vamos terminando en **Object**, esto es porque **Object** es la clase raíz principal en el orden de jerarquías de herencias. Sin embargo, **Object** tiene como .**superclass** a **BasicObject**, pero esto es particular, siempre tendremos como nodo común a **Object**.

En la parte de instanciación tenemos que toda instancia viene de una **Class** pues todo en Ruby es un objeto digamos.

- b) Si A y B son las raíces encontradas para las jerarquías, respectivamente:

  ~ ¿Cuál es el resultado de aplicar A.class?

  En este caso A = Object, por tanto Object.class = Class.

  ~ ¿Cuál es el resultado de aplicar A.superclass?

  En este caso A = Object, por tanto Object.superclass = BasicObject.

  ~ ¿Cuál es el resultado de aplicar B.class?

  En este caso B = Class, por tanto Class.class = Class

  ~ ¿Cuál es el resultado de aplicar B.superclass?

  En este caso B = Class, por tanto Class.superclass = Module

- c) ¿Cual deberá ser el resultado de aplicar R.class? Sera cierto que $R \in R$?

Debido a las características de Ruby, R es un Object, y sabemos que Object.class es Class, es decir R.class = Class, por lo que, la definición afirma que $R \in R$. Ahora bien, si se quiere instanciar R dado que es una clase, digamos x es instancia de R, este x debe cumplir por definición que debe ser un elemento compuesto tal que no se instancia a sí mismo, y el único elemento que cumple dicha especificación es el mismo R, entonces *R es una instancia de R*, lo cual contradice la definición de R, así $R \notin R$. Esto presenta una paradoja de pertenencia $R \in R \land R \notin R$.

- d) ¿Explique la relación que tiene el resultado anterior con la paradoja de Russell?

Acá el link directo a la paradoja. Esta plantea la problemática del bardero que afeita a todos los que no pueden afeitar a sí mismo, pero el siendo el único barbero del pueblo no puede afeitarse a sí mismo, pero al mismo tiempo sino se afeita, entonces el barbero del pueblo debería afeitarlo, pero él es el único barbero del pueblo.

En este sentido, a nivel de modelo en conjuntos tenemos la pertenencia y no pertenencia simultáneamente, esto es similar a la problemática planteada en el inciso anterior de $R \in R \land R \notin R$.

Triki Question: ¿Y si ese barbero es *The Rock*? (Calvo)

Bibliografías

Mellado, J, (2018), Scala (4), inmensia, Inmensia.com,
https://inmensia.com/blog/20181203/scala-avanzado/

Variance. (2023). Scala Documentation.
https://docs.scala-lang.org/scala3/book/types-variance.html