Jose Revete
19-10040

Tarea 2 (Ruby)

1. **Propiedades y herencia.**
    a. **Considere una clase Circulo, con un único campo radio.**

R:

```ruby
# Clase Circulo
# a)
class Circulo
  # c)
  def initialize(number)
    puts number
    if number < 0
      raise ArgumentError, 'Radio invalido'
    end
    @radio = number
  end

  # b)
  def radio
    @radio
  end

  def radio=(radio)
    if radio < 0
      raise ArgumentError, 'Radio invalido'
    end
    @radio = radio
  end

  # d)
  def area
    Math::PI * @radio * @radio
  end
end
```

    b. **Considere una subclase Cilindro de Círculo, que agrega un único campo altura.**

R:

```ruby
# Clase Cilindro
# a)
class Cilindro < Circulo
  # c)
  def initialize(radio, altura)
    if radio < 0
      raise ArgumentError, 'Radio invalido'
    elsif altura < 0
      raise ArgumentError, 'Altura invalida'
    end
    @radio = radio
    @altura = altura
  end

  # b)
```

```ruby
  def altura
    @altura
  end

  def altura=(altura)
    if altura < 0
      raise ArgumentError, 'Altura invalida'
    end
    @altura = altura
  end

  # d)
  def volumen
    circulo = Circulo.new(@radio)
    circulo.area * @altura
  end
end

—----------------------------------------
# Pruebas
  circulo = Circulo.new(5)
  puts circulo.radio
  puts circulo.area
  circulo.radio = 10
  puts circulo.radio
  puts circulo.area
  circulo.radio = -10 # Debería lanzar una excepción
  puts circulo.radio
  puts circulo.area
  puts '--------------------'

  cilindro = Cilindro.new(5, 10)
  puts cilindro.radio
  puts cilindro.altura
  puts cilindro.volumen
  cilindro.radio = 10
  cilindro.altura = 20
  puts cilindro.radio
  puts cilindro.altura
  puts cilindro.volumen
  cilindro.radio = -10 # Debería lanzar una excepción
  cilindro.altura = -20 # Debería lanzar una excepción
  puts cilindro.radio
  puts cilindro.altura
  puts cilindro.volumen
```

2. **Defina una clase Moneda con subclases Dolar, Yen, Euro, Bolivar y Bitcoin.**

R:

```ruby
# b) Clase Moneda
class Moneda
  def initialize(number)
    if number < 0
      raise ArgumentError, 'Numero invalido'
    end
    @number = number
  end

  # b) Metodo que convierte una moneda a otra
  def en(clase)
    case clase
```

```ruby
    when :dolares
      Dolar.new(@number)
    when :yens
      Yen.new(@number)
    when :euros
      Euro.new(@number)
    when :bolivares
      Bolivar.new(@number)
    when :bitcoins
      Bitcoin.new(@number)
    else
      raise ArgumentError, 'Clase invalida'
    end
  end

# c) Metodo que compara dos monedas
def comparar(otra_moneda)
  # Determinar la clase actual como símbolo
  clase_actual = case self.class.name
                 when "Yen" then :yens
                 when "Euro" then :euros
                 when "Bitcoin" then :bitcoins
                 when "Bolivar" then :bolivares
                 when "Dolar" then :dolares
                 else
                   raise ArgumentError, 'Clase invalida'
                 end

  moneda_convertida = otra_moneda.en(clase_actual)

  # Comparar los valores de las monedas
  if @number > moneda_convertida.number
    :mayor
  elsif @number < moneda_convertida.number
    :menor
  else
    :igual
  end
end

  def number
    @number
  end

  def to_s
    "#{@number}"
  end
end

# Clases de las monedas
# b) Los datos de conversion son actuales
class Dolar < Moneda
  def en(clase)
    case clase
    when :dolares
      Dolar.new(@number)
    when :yens
      Yen.new(@number * 154.46)
    when :euros
      Euro.new(@number * 0.95)
    when :bolivares
      Bolivar.new(@number * 45.98)
    when :bitcoins
```

```
      Bitcoin.new(@number * 0.00001)
    else
      raise ArgumentError, 'Clase invalida'
    end
  end

  def to_s
    "#{@number} Dolares"
  end
end

class Yen < Moneda
  def en(clase)
    case clase
    when :dolares
      Dolar.new(@number * 0.0065)
    when :yens
      Yen.new(@number)
    when :euros
      Euro.new(@number * 0.0062)
    when :bolivares
      Bolivar.new(@number * 0.29850)
    when :bitcoins
      Bitcoin.new(@number * 0.000000022)
    else
      raise ArgumentError, 'Clase invalida'
    end
  end

  def to_s
    "#{@number} Yenes"
  end
end

class Euro < Moneda
  def en(clase)
    case clase
    when :dolares
      Dolar.new(@number * 1.05)
    when :yens
      Yen.new(@number * 161.72)
    when :euros
      Euro.new(@number)
    when :bolivares
      Bolivar.new(@number * 48.28)
    when :bitcoins
      Bitcoin.new(@number * 0.0000105)
    else
      raise ArgumentError, 'Clase invalida'
    end
  end

  def to_s
    "#{@number} Euros"
  end
end

class Bolivar < Moneda
  def en(clase)
    case clase
    when :dolares
      Dolar.new(@number * 0.0217)
    when :yens
```

```
      Yen.new(@number * 3.35)
    when :euros
      Euro.new(@number * 0.0206)
    when :bolivares
      Bolivar.new(@number)
    when :bitcoins
      Bitcoin.new(@number * 0.000000217)
    else
      raise ArgumentError, 'Clase invalida'
    end
  end

  def to_s
    "#{@number} Bolivares"
  end
end

class Bitcoin < Moneda
  def en(clase)
    case clase
    when :dolares
      Dolar.new(@number * 99170.55)
    when :yens
      Yen.new(@number * 15237248.87)
    when :euros
      Euro.new(@number * 94192.49)
    when :bolivares
      Bolivar.new(@number * 4559861.89)
    when :bitcoins
      Bitcoin.new(@number)
    else
      raise ArgumentError, 'Clase invalida'
    end
  end

  def to_s
    "#{@number} Bitcoins"
  end
end

# a) Clase Float, para agregar metodos a los numeros
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
```

```ruby
# Pruebas
puts 1.0.dolares.en(:yens)
puts 2.0.yens.en(:dolares)
puts 3.0.euros.en(:bolivares)
puts 8.0.euros.en(:euros)
puts 9.0.bolivares.en(:bolivares)
puts 10.0.bitcoins.en(:bitcoins)
puts 15.0.dolares.en(:euros)
puts 50.0.bolivares.comparar(2.0.dolares)
```

3. **Bloques e iteradores.**

R:

```ruby
# Función que recibe dos arreglos y devuelve el producto cartesiano de ambos arreglos.
def producto_cartesiano(arreglo1, arreglo2)
  for i in 0..arreglo1.length-1
    for j in 0..arreglo2.length-1
      yield [arreglo1[i], arreglo2[j]]
    end
  end
end

# Pruebas
arreglo1 = [:a, :b, :c]
arreglo2 = [4, 5]

producto_cartesiano(arreglo1, arreglo2) do |par|
  puts par.inspect
end
```

Investigación:

1. **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>?**

R: En general, no habría ninguna relación de herencia entre Lista<A> y Lista<B>, pero es posible el caso en el que exista relación únicamente para la lectura de estas listas (caso poco probable en cuanto a los lenguajes que lo permiten) o que permita que Lista<A> sea tratada como Lista<B> (aun menos probable en cuanto a los lenguajes que lo permiten).

i. **¿Qué decisión toma el lenguaje Java? Explique dicha decisión.**

R: Java trata los tipos genéricos como **invariantes**, es decir que Lista<A> y Lista<B> son tipos independientes, sin importar la relación de herencia entre A y B. Por lo tanto, no tienen ninguna relación de herencia entre sí, incluso si B es subclase de A

ii. **¿Qué decisión toma el lenguaje Scala? Explique dicha decisión.**

R: En Scala, la relación de herencia entre Lista[A] y Lista[B] depende de cómo se declare la clase genérica. Por ejemplo, con declaración de **covarianza** (+T) permite que Lista[B] sea un subtipo de Lista[A] si B es subtipo de A, lo cual es útil para estructuras inmutables que solo leen datos, pero no puedes agregar elementos. También está la **contravarianza** (-T), que permite que Lista[A] sea un subtipo de Lista[B] si B es subtipo de A, esto es ideal para consumidores de datos, pero cuenta con el limitante de la lectura a tipos generales. La **invariancia** (el cual se asigna por defecto) no establece relación entre Lista[A] y Lista[B], incluso si hay herencia entre A y B.

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

R: En el caso de Bottom, encontramos que el tipo inferido de la lista vacía podría darse de dos manera: List<Bottom> (sin contexto explícito), dado que Bottom es el tipo más genérico y compatible con cualquier clase; o que el tipo sea determinado por el contexto en el que se use la lista, ajustándose al tipo esperado.

b. **Considere ahora que el lenguaje de programación en el que se está trabajando es funcional. Cómo es puramente orientado a objetos, las funciones también deben ser objetos. Suponga otra clase cualquiera C.**
   i. **¿Qué relación de herencia, de haberla, tienen las funciones con firmas A → C y B → C? Justifique su respuesta.**

R: Sabiendo que B es subclase de A, entonces sabemos que para A → C, esta función que recibe A, puede trabajar con cualquier instancia de A, incluidas las instancias de B, por lo que obtendremos que B → C es subclase de A → C.

ii. **¿Qué relación de herencia, de haberla, tienen las funciones con firmas C → A y C → B? Justifique su respuesta.**

R: Nuevamente, al ser B subclase de A, entonces C → B es subclase de C → A, dado que en la función que retorna B se espera un resultado más específico que igualmente se puede encontrar en A y más.

---

2. **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?**

R: Con respecto a la jerarquía de herencia, la clase **BasicObject** es la clase padre de todas las clases de Ruby. Por lo tanto, sus métodos están disponibles para todos los objetos a menos que se anulen explícitamente. Por otro lado, para la instanciación sería **Class**, donde incluso la propia clase **Class** es una instancia de sí misma.

b. **Si A y B son las raíces encontradas para las jerarquías, respectivamente:**

i. **¿Cuál es el resultado de aplicar A.class?**

R: El resultado seria **Class**

ii. **¿Ćal es el resultado de aplicar A.superclass?**

R: El resultado sería **nil**

iii. **¿Cuál es el resultado de aplicar B.class?**

R: El resultado seria **Class**

iv. **¿Cuál es el resultado de aplicar B.superclass?**

R: El resultado sería **Module**

c. **Sea R = {A | A.class != A}. Si consideramos R como un valor más, ¿cuál debería ser el resultado de aplicar R.class? ¿Será cierto que R ∈ R?**

R: Si consideramos a R como un valor más, entonces necesariamente debería tener una clase asociada, por lo que tendremos 2 casos:

- Si **R.class == R**, entonces R no puede estar en R, por lo que genera una paradoja.
- Si **R.class != R**, entonces R debe estar en R, por lo que sería otra paradoja.

Por lo que podemos concluir que no hay una respuesta a la pregunta.

d. **Explique la relación que tiene el resultado anterior con la paradoja de Russell.**

R: La paradoja de Russell habla de el conjunto de todos los conjuntos que no contienen a sí mismos, **S={A | A no pertenece A}**, por lo que encontramos relación con este ejemplo dado que si A no pertenece a A, entonces, debe pertenecer a A, y si A pertenece a A, entonces no debe pertenecer a A.