Tarea 2: Ruby

Carlo Miguel Herrera Di Giacinto (18-10451)

Parte I. Implementación

1. Propiedades y herencia

- Considere una clase Circulo, con un´unico campo radio.
    - Defina la clase en cuestión, con el campo propuesto.
    - Implemente *setters* y *getters* para el campo radio de Circulo.
    - Implemente un método `initialize` (constructor) para Circulo que reciba un número e inicialice el radio del círculo con dicho número.
        - En caso de que el número propuesto sea negativo, se debe arrojar un error con el mensaje: 'Radio invalido'.
    - Implemente un método `area` para Circulo que retorne el área del círculo.

```
# Clase Circulo
class Circulo
    attr_accessor :radio

    def initialize(radio)
        if (radio < 0)
            raise "Radio inválido."
        else
            @radio = radio
        end
    end

    def radio=(radio)
        if (radio < 0)
            raise "Radio inválido."
        else
            @radio = radio
        end
    end

    def radio
        @radio
    end

    def area
        area =  3.1416 * @radio * @radio
    end
end
```

- Considere una subclase Cilindro de Circulo, que agrega un´unico campo altura.
    - Defina la subclase en cuestión, con el campo adicional propuesto.
    - Implemente *setters* y *getters* para el campo altura de Circulo.
    - Implemente un método `initialize` (constructor) para Cilindro que reciba dos números e inicialice el radio y altura del cilindro con dichos números.
        - En caso de que el radio propuesto sea negativo, se debe arrojar un error con el mensaje: 'Radio invalido'.
        - En caso de que la altura propuesta sea negativo, se debe arrojar un error con el mensaje: 'Altura invalida'.
    - Implemente un método volumen para Cilindro que retorne el volumen del cilindro.

```
# Subclase Cilindro de la clase Circulo
class Cilindro < Circulo
    attr_accessor :altura

    def initialize(radio, altura)
        super(radio) # Utilizamos el initialize de la superclase para inicializar el radio

        if (altura < 0)
            raise "Altura inválida."
        else
            @altura = altura
        end
    end

    def altura
        @altura
    end

    def altura=(altura)
        @altura = altura
    end

    def volumen
        volumen = 3.1416 * @radio * @radio * @altura
    end

end
```

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

- Defina métodos `dolares`, `yens`, `euros`, `bolivares` y `bitcoins` sobre la clase `Float` que convierta el flotante en dólares, yens, euros, bolívares y bitcoins, respectivamente.
- Defina un método en sobre la clase `Moneda` (y sus subclases, por ende) que reciba un átomo entre `:dolares`, `:yens`, `:euros`, `:bolivares` y `:bitcoins` y convierta la moneda en aquella representada por el átomo propuesto.

  Por ejemplo: `15.dolares.en(:euros)` debe evaluar en 14.16 euros.
- Defina un método `comparar` sobre la clase Moneda, que reciba otra Moneda y las compare.
  - Debe devolver `:menor` si la primera moneda es menor que el argumento.
  - Debe devolver `:igual` si la primera moneda es igual que el argumento.
  - Debe devolver `:mayor` si la primera moneda es mayor que el argumento.

  Por ejemplo: `50.bolivares.comparar(2.dolares)` debe evaluar en `:menor`

Definicion de la clase Moneda

```
class Moneda
    attr_reader :valor

    def initialize(valor)
        @valor = valor
    end

    def en(divisa)
        divisa.cambia(self)
    end

    def cambia(moneda)
        raise NotImplementedError "Este metodo debe ser implementado por cada subclase"
    end
end
```

Definicion de las subclases de Moneda

```
# Subclase Dolares
```

```
class Dolar < Moneda

    def initialize(valor)
        super(valor)
    end

    def to_s
        "#{@valor} USD"
    end

    def cambia(divisa)
        divisa.de_dolar(self)
    end

    def de_dolar(:dolares)
        self
    end

    def de_euro(:euros)
        Dolar.new(euro.valor * 1.04829)
    end

    def de_yen(:yenes)
        Dolar.new(yen.valor / 152.27074)
    end

    def de_bolivar(:bolivares)
        Dolar.new(bolivar.valor / 46.7537 )
    end

    def de_bitcoin(:bitcoins)
        Dolar.new(bitcoin.valor * 92871.2)
    end
end

# Subclase Euros
class Euro < Moneda

    def initialize(valor)
        super(valor)
    end

    def to_s
        "#{@valor} EUR"
    end

    def cambia(divisa)
        divisa.de_euro(self)
    end

    def de_dolar(:dolares)
        Euro.new(dolar.valor / 1.04829)
    end

    def de_euro(:euros)
        self
    end

    def de_yen(:yenes)
        Euro.new(yen.valor / 159.597)
    end

    def de_bolivar(:bolivares)
        Euro.new(bolivar.valor / 49.077359 )
    end
```

```
    def de_bitcoin(:bitcoins)
        Euro.new(bitcoin.valor * 88761.54)
    end
end

# Subclase Yenes
class Yen < Moneda

    def initialize(valor)
        super(valor)
    end

    def to_s
        "#{@valor} JPY"
    end

    def cambia(divisa)
        divisa.de_yen(self)
    end

    def de_dolar(:dolares)
        Yen.new(dolar.valor * 152.288)
    end

    def de_euro(:euros)
        Yen.new(euro.valor * 159.597)
    end

    def de_yen(:yenes)
        self
    end

    def de_bolivar(:bolivares)
        Yen.new(bolivar.valor * 3.257239)
    end

    def de_bitcoin(:bitcoins)
        Yen.new(bitcoin.valor * 14125895.62)
    end
end

# Subclase Bolivares
class Bolivar < Moneda

    def initialize(valor)
        super(valor)
    end

    def to_s
        "#{@valor} VES"
    end

    def cambia(divisa)
        divisa.de_bolivar(self)
    end

    def de_dolar(:dolares)
        Bolivar.new(dolar.valor * 46.7537)
    end

    def de_euro(:euros)
        Bolivar.new(euro.valor * 49.077359)
    end

    def de_yen(:yenes)
```

```
        Bolivar.new(yen.valor / 3.257239)
    end

    def de_bolivar(:bolivares)
       self
    end

    def de_bitcoin(:bitcoins)
        Bolivar.new(bitcoin.valor * 4342072.22344)
    end
end

# Subclase Bitcoins
class Bitcoin < Moneda

    def initialize(valor)
        super(valor)
    end

    def to_s
        "#{@valor} BTC"
    end

    def cambia(divisa)
        divisa.de_bitcoin(self)
    end

    def de_dolar(:dolares)
        Bitcoin.new(dolar.valor / 92871.2)
    end

    def de_euro(:euros)
        Bitcoin.new(euro.valor / 88761.54)
    end

    def de_yen(:yenes)
        Bitcoin.new(yen.valor / 14125895.62)
    end

    def de_bolivar(:bolivares)
        Bitcoin.new(bolivar.valor / 4342072.22344)
    end

    def de_bitcoin(:bitcoins)
        self
    end
end
```

Extendemos la clase Float para incluir metodos constructores de Moneda

```
module Divisas
    def dolares
        Dolar.new(self)
    end

    def euros
        Euro.new(self)
    end

    def yenes
        Yen.new(self)
    end

    def bolivares
        Bolivar.new(self)
    end

    def bitcoins
        Bitcoin.new(self)
    end
end

class Float
    include Divisas
end
```

3. Bloques e Iteradores

Dadas dos colecciones (de tipos posiblemente diferentes), se desea calcular el producto cartesiano de los elementos generados para cada una de ellas.

Por ejemplo: El producto cartesiano de `[:a, :b, :c]` y `[4, 5]` debe generar:

```
[:a, 4]
[:a, 5]
[:b, 4]
[:b, 5]
[:c, 4]
[:c, 5]
```

> Nota 1: No importa el orden en que se devuelvan los elementos, sino que todos los elementos aparezcan.

> Nota 2: El elemento `[:a, 4]` está en el resultado del ejemplo anterior, pero `[4. :a]` no. El orden interno de las tuplas es importante.

```
# Definición del iterador del producto cartesiano.
def cart_prod(a, b)
    a.each do |elema|
        b.each do |elemb|
            yield [elema, elemb]
        end
    end
end

# Llamada que imprime cada elemento yieldeado.
cart_prod(x, y) do |item|
    p item
end
```

Parte II. Investigación