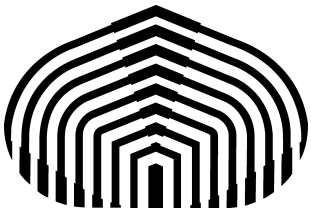

UNIVERSIDAD SIMÓN BOLÍVAR
INGENIERÍA DE LA COMPUTACIÓN
DEPT.COMPUTACIÓN Y TECNOLOGÍA DE LA INFORMACIÓN
CI3661: LABORATORIO DE LENGUAJES DE PROGRAMACIÓN I

Tarea de Ruby

PROFESOR:

Fernando Lovera
.

ALUMNOS:

Andres A. Buelvas V. 13-10184
Miguel C. Canedo R. 13-10214

Implementación

Círculo y Cilindro

(3 pts) - Propiedades y herancia. Considere una clase Circulo, con un unico campo radio.

a. (0.25 pts) – Defina la clase en cuestion, con el campo propuesto.

b. (0.25 pts) – Implemente setters y getters para el campo radio de Circulo.

c. (0.5 pts) – Implemente un metodo initialize (constructor) para Circulo que reciba un numero e inicialice el radio del cirrculo con dicho numero. En caso de que el número propuesto sea negativo, se debe arrojar un error con el mensaje: 'Radio invalido'.

d. (0.5 pts) – Implemente un metodo area para Circulo que retorne el area del cırculo.

Considere una subclase Cilindro de Circulo, que agrega un unico campo altura:

a. (0.25 pts) – Defina la subclase en cuestion, con el campo adicional propuesto.

b. (0.25 pts) – Implemente setters y getters para el campo altura de Circulo.

c. (0.5 pts) – Implemente un metodo initialize (constructor) para Cilindro que reciba dos numeros e inicialice el radio y altura del cilindro con dicho numero. 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'.

d. (0.5 pts) – Implemente un método volumen para Cilindro que retorne el volumen del cilindro.

Solución:

```ruby
class Circulo

  # Constructor de la clase Circulo
  def initialize radio
    if radio < 0
      raise 'Radio Invalido'
    else
      @radio = radio
    end
  end
```

```ruby
  # Metodo que permite colocar el radio al Circulo
  def set_radio radio
    if radio < 0
      raise 'Radio Invalido'
    else
      @radio = radio
    end
  end

  # Metodo que permite obtener el radio del Circulo
  def get_radio
    @radio
  end

  # Metodo que calcula y retorna el area del Circulo
  def area
    Math::PI * (@radio**2)
  end
end

# Clase Cilindro que representa a un Cilindro
class Cilindro < Circulo
  # Constructor de la Clase Cilindro
  def initialize radio, altura
    super radio
    if altura < 0
      raise 'Altura Invalido'
    else
      @altura = altura
    end
  end

  # Metodo que permite colocar la altura al Cilindro
  def set_altura altura
    if altura < 0
      raise 'Altura Invalido'
    else
      @altura = altura
    end
  end

  # Metodo que retorna la altura del Cilindro
  def get_altura
    @altura
  end
  # Metodo que calcula y retorna el volumen del Cilindro
  def volumen
    self.area*@altura
  end
end
```

Monedas Internacionales

(2 pts) – Defina una clase Moneda con subclases Dolar, Yen, Euro, Bolivar y Bitcoin.

a. (0.5 pts) – Defina metodos dolares, yens, euros, bolivares y bitcoins sobre la clase Float que convierta el flotante en dolares, yens, euros, bolıvares y bitcoins, respectivamente.

b. (1 pt) – Defina un metodo en sobre la clase Moneda (y sus subclases, por ende) que reciba un atomo entre: dolares, :yens, :euros, :bolivares y :bitcoins y convierta la moneda en aquella representada por el atomo propuesto. Por ejemplo: 15.dolares.en(:euros) debe evaluar en 12.72 euros.

c. (1 pt) – Defina un metodo 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: 100000.bolivares.comparar(2.dolares) debe evaluar en :menor. Nota: Use doble despacho para averiguar los tipos del argumento pasado. No pregunte por el tipo explıcitamente.

Solución:

```
# Clase Float (Flotante)
class Float

  # Metodo que convierte el flotante respectivo en un 'Dolar'
  def dolares
    Dolar.new self
  end

  # Metodo que convierte el flotante respectivo en un 'Yen'
  def yens
    Yen.new self
  end

  # Metodo que convierte el flotante respectivo en un 'Euro'
  def euros
    Euro.new self
  end

  # Metodo que convierte el flotante respectivo en un 'Bolivar'
  def bolivares
    Bolivar.new self
  end

  # Metodo que convierte el flotante respectivo en un 'Bitcoin'
```

```
  def bitcoins
    Bitcoin.new self
  end
end

# Clase Moneda que representa a una moneda cualquiera
class Moneda

  attr_accessor :valor

  # Constructor de la Clase Moneda
  def initialize arg
    @valor = arg
  end

  # Metodo que calcula que reciba un atomo entre: :dolares, :yens,
  # :euros, :bolivares y :bitcoins y convierta la moneda en aquella
  # representada por el atomo propuesto.
  def en id
    case id
      when :dolares
        self.enAux :dolares

      when :yens
        self.enAux :yens

      when :euros
        self.enAux :euros

      when :bolivares
        self.enAux :bolivares

      when :bitcoins
        self.enAux :bitcoins
    end
  end

  # Metodo que permite comparar dos monedas
  def comparar otraMoneda
    otraMoneda.compararAux self
  end
end

# Subclase Dolar que representa al tipo de moneda 'Dolar'
class Dolar < Moneda

  # Constructor de la Clase 'Dolar'
  def initialize arg
    super arg
  end

  # Metodo auxiliar del metodo comparar que se encuentra en la clase
```

```
    Moneda que
  # dice si la moneda preguntada es mayor, menor o igual a la moneda con
    la
  # cual se comparo
  def compararAux otraMoneda

    otroValor = otraMoneda.en(:dolares).valor # Valor de la segunda
    moneda

    if otroValor < @valor
      :menor
    elsif otroValor == @valor
      :igual
    else
      :mayor
    end
  end

  # Metodo Auxiliar del metodo 'en' de la clase Moneda que realiza los
    calculos
  # y devuelve el nuevo valor en el cambio propuesto
  def enAux id
    case id
      when :dolares
        Dolar.new@valor*1
      when :yens
        Yen.new@valor*106.56
      when :euros
        Euro.new@valor*0.80
      when :bolivares
        Bolivar.new@valor*216164.85
      when :bitcoins
        Bitcoin.new@valor*0.000108
    end
  end
end

# Subclase Yen que representa al tipo de moneda 'Yen'
class Yen < Moneda

  # Constructor de la Clase 'Yen'
  def initialize arg
    super arg
  end

  # Metodo auxiliar del metodo comparar que se encuentra en la clase
    Moneda que
  # dice si la moneda preguntada es mayor, menor o igual a la moneda con
    la
  # cual se comparo
  def compararAux otraMoneda
```

```
    otroValor = otraMoneda.en(:yens).valor # Valor de la segunda moneda

    if otroValor < @valor
      :menor
    elsif otroValor == @valor
      :igual
    else
      :mayor
    end
  end

  # Metodo Auxiliar del metodo 'en' de la clase Moneda que realiza los
    calculos
  # y devuelve el nuevo valor en el cambio propuesto
  def enAux id
    case id
      when :dolares
        Dolar.new@valor*0.00938
      when :yens
        Yen.new@valor*1
      when :euros
        Euro.new@valor* 0.00757
      when :bolivares
        Bolivar.new@valor*2027.62
      when :bitcoins
        Bitcoin.new@valor*0.0000010
    end
  end
end

# Subclase Euro que representa al tipo de moneda 'Euro'
class Euro < Moneda

  # Constructor de la Clase 'Euro'
  def initialize arg
    super arg
  end

  # Metodo auxiliar del metodo comparar que se encuentra en la clase
    Moneda que
  # dice si la moneda preguntada es mayor, menor o igual a la moneda con
    la
  # cual se comparo
  def compararAux otraMoneda

    otroValor = otraMoneda.en(:euros).valor # Valor de la segunda moneda

    if otroValor < @valor
      :menor
    elsif otroValor == @valor
      :igual
    else
```

```
      :mayor
    end
  end

  # Metodo Auxiliar del metodo 'en' de la clase Moneda que realiza los
    calculos
  # y devuelve el nuevo valor en el cambio propuesto
  def enAux id
    case id
      when :dolares
        Dolar.new@valor*    1.23
      when :yens
        Yen.new@valor*132.10
      when :euros
        Euro.new@valor*1
      when :bolivares
        Bolivar.new@valor*268169.76
      when :bitcoins
        Bitcoin.new@valor*0.00013
    end
  end
end

# Subclase Bolivar que representa al tipo de moneda 'Bolivar'
class Bolivar < Moneda

  # Constructor de la Clase Bolivar
  def initialize arg
    super arg
  end

  # Metodo auxiliar del metodo comparar que se encuentra en la clase
    Moneda que
  # dice si la moneda preguntada es mayor, menor o igual a la moneda con
    la
  # cual se comparo
  def compararAux otraMoneda

    otroValor = otraMoneda.en(:bolivares).valor # Valor de la segunda
    moneda

    if otroValor < @valor
      :menor
    elsif otroValor == @valor
      :igual
    else
      :mayor
    end
  end

  # Metodo Auxiliar del metodo 'en' de la clase Moneda que realiza los
    calculos
```

```
  # y devuelve el nuevo valor en el cambio propuesto
  def enAux id
    case id
      when :dolares
        Dolar.new@valor*0.00000462609
      when :yens
        Yen.new@valor*0.00049298978
      when :euros
        Euro.new@valor*0.00000372898
      when :bolivares
        Bolivar.new@valor*1
      when :bitcoins
        Bitcoin.new@valor*0.00000000049961772
    end
  end
end

# Subclase Bitcoin que representa al tipo de moneda 'Bitcoin'
class Bitcoin < Moneda

  # Constructor de la Clase Bitcoin
  def initialize arg
    super arg
  end

  # Metodo auxiliar del metodo comparar que se encuentra en la clase
    Moneda que
  # dice si la moneda preguntada es mayor, menor o igual a la moneda con
    la
  # cual se comparo
  def compararAux otraMoneda

    otroValor = otraMoneda.en(:bitcoins).valor # Valor de la segunda
    moneda

    if otroValor < @valor
      :menor
    elsif otroValor == @valor
      :igual
    else
      :mayor
    end
  end

  # Metodo Auxiliar del metodo 'en' de la clase Moneda que realiza los
    calculos
  # y devuelve el nuevo valor en el cambio propuesto
  def enAux id
    case id
      when :dolares
        Dolar.new@valor*9315.86
      when :yens
```

```
        Yen.new@valor*992528.49
      when :euros
        Euro.new@valor*7862.07
      when :bolivares
        Bolivar.new@valor*2013761479.52
      when :bitcoins
        Bitcoin.new@valor*1
    end
  end
end
```

Producto Cartesiano

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] esta en el resultado del ejemplo anterior, pero [4. :a] no. El orden interno de las tuplas es importante.

Solución:

```
# Clase ProductoCartesiano
class ProductoCartesiano

  # Constructor de la Clase ProductoCartesiano
  def initialize arreglo1, arreglo2
    @arreglo1 = arreglo1
    @arreglo2 = arreglo2
  end

  # Metodo que llevara a cabo el producto cartesiano y lo imprimira en
    pantalla
  def producto
```

```
    @arreglo1.each { |i| @arreglo2.each { |j| yield [i,j] } }
  end
end
```

Nota : Para que corra el método 'producto' de la Clase 'ProductoCartesiano' se debe correr como en el siguiente ejemplo:

```
A = ['a','b','c']
B = [4,5,6]
prod = ProductoCartesiano.new A, B
prod.producto do
  |x| print x
  puts
```

DFS

(4 pts) - Ud. debe implantar un DFS que pueda ser utilizado por dos clases, empleando mixins. Tu DFS debe ofrecer lo siguiente:

- find(start, predicate) que comienza la busqueda DFS a partir del objeto start hasta encontrar el primer objeto que cumpla el predicado predicate y la retorna.Debes devolver nil si nadie cumple el predicado.
- path(start, predicate) comienza un recorrido DFS a partir de start, hasta encontrar el primer objeto que cumpla el predicado predicate. Te devuelve el camino desde start hasta el nodo encontrado (un arreglo de objetos). Si nadie cumple el predicado, se retorna un arreflo vacio.

Solución:

```
# Modulo DFS donde se generalizan dos metodos de recorrido para las
    clases
# ArbolBinario y GrafoDirigido.
module DFS
  # Metodo que se encarga de buscar en la estructura, comenzando desde '
    start',
  # el primer elemento que cumpla el 'predicate' y lo devuelve.
  # En caso de que nadie cumpla el 'predicate', retorna 'nil'.
  def find(start, predicate, visited = [])
    visited << start     # Marcamos a 'start' como visitado.
    encontrado = nil

    # Si 'start' cumple la condicion del predicate, lo retornamos a el.
    if predicate.call(start.dato)
```

```
      return start.dato
    end

    # Recorremos todos los nodos o vertices que le siguen a 'start'.
    start.siguientes.each do |n|
      next if visited.include?(n)        # Si fue visitado se ignora.
      if n
        encontrado = find(n, predicate, visited)    # Se sigue la
    recursion ahora con 'n' como 'start'.
        break if encontrado
      end
    end

    # Retornamos el vertice o nodo encontrado, o 'nil'.
    encontrado
  end

  # Metodo que se encarga de buscar en la estructura, comenzando desde '
    start',
  # el primer elemento que cumpla el 'predicate' y devuelve el camino
    hacia el mismo.
  # En caso de que nadie cumpla el 'predicate', retorna 'nil'.
  def path(start, predicate, visited = [], p = [])
    pEncontrado = nil
    visited << start    # Marcamos a 'start' como visitado.

    # Agregamos el vertice 'start' al camino.
    pathNuevo = p.clone()
    pathNuevo << start

    # Si 'start' cumple la condicion del predicate, se retorna el camino
    encontrado hasta el.
    if predicate.call(start.dato)
      return pathNuevo
    end

    # Recorremos todos los nodos o vertices que le siguen a 'start'.
    start.siguientes.each do |n|
      next if visited.include?(n)     # Si fue visitado se ignora.
      if n
        pEncontrado = path(n, predicate, visited, pathNuevo)    # Se
    sigue la recursion ahora con 'n' como 'start'.
        break if pEncontrado
      end
    end

    # Retornamos el camino encontrado, o 'nil'.
    pEncontrado
  end
end

# Clase ArbolBinario representa un Arbol Binario de Busqueda sencillo.
```

```ruby
# Incluye los metodos del Modulo DFS.
class ArbolBinario
  include DFS

  attr_accessor :root

  # Clase Nodo que se encarga de almacenar los datos de un Arbol Binario
  # ademas de sus hijos en dicho Arbol.
  class Nodo
    attr_accessor :dato, :izq, :der

    # Metodo inicilizador de la clase Nodo.
    def initialize r
      @dato = r     # Dato que almacena el Nodo.
      @izq = nil     # Hijo Izquierdo del Nodo.
      @der = nil    # Hijo Derecho del Nodo.
    end

    # Metodo que devuelve los hijos de Nodo empaquetados en un arreglo.
    def siguientes
      [@izq, @der]
    end

    # Metodo que devuleve la representacion en String de la clase Nodo.
    def to_s
      "#{dato}"
    end
  end

  # Metodo Inicilizador de la clase ArbolBinario.
  def inicialize
    @root = nil # Raiz del ArbolBinario.
  end

  # Metodo que inserta de manera ordenada un nuevo elemento en el Arbol
    Binario.
  def insertar (nuevo, nodo = @root)
    if @root
      if nodo.dato >= nuevo
        if nodo.izq
          insertar(nuevo, nodo.izq)
        else
          nodo.izq = Nodo.new(nuevo)
        end
      else
        if nodo.der
          insertar(nuevo, nodo.der)
        else
          nodo.der = Nodo.new(nuevo)
        end
      end
    else
```

```
      @root = Nodo.new(nuevo)
    end
  end

  # Metodo que devuleve la representacion en String de la clase
    ArbolBinario.
  def to_s
    nodos = [@root]
    s = "Arbol Binario: "
    while !nodos.empty?
      s = s + nodos[0].to_s + " "

      nodos << nodos[0].izq if nodos[0].izq
      nodos << nodos[0].der if nodos[0].der

      nodos = nodos.drop(1)
    end

    s
  end

end

# Clase GrafoDirigido representa un Grafo donde sus lados posee una
    direccion unica.
# Incluye los metodos del Modulo DFS.
class GrafoDirigido
  include DFS

  attr_accessor :vertices

  # Clase Vertice que se encarga de almacenar los datos de un Grafo
    Dirigido,
  # ademas de los vertices a los que se dirige.
  class Vertice
    attr_accessor :dato, :vecinos

    # Metodo inicilizador de la clase Veritice.
    def initialize(dato)
      @dato = dato     # Dato que almacena el Vertice.
      @vecinos = []    # Vertices a los que se dirige este.
    end

    # Metodo que devuelve el arreglo de Vertices a destinos desde este.
    def siguientes
      @vecinos
    end

    # Metodo que devuleve la representacion en String de la clase Vertice
    .
    def to_s
      "#{dato}"
```

```
    end

  end

  # Metodo inicilizador de la clase GrafoDirigido.
  def initialize
    @vertices = []
  end

  # Metodo que agrega un Vertice al Grafo.
  def agregarVertice(dato)
    @vertices << Vertice.new(dato)
  end

  # Metodo que agrega una Arista desde el Vertice 'origen'
  # hasta Vertice 'destino'.
  def agregarArista(origen, destino)
    desde = vertices.index { |v| v.dato == origen }
    hasta = vertices.index { |v| v.dato == destino }

    vertices[desde].vecinos << vertices[hasta]
  end

  # Metodo que devuleve la representacion en String de la clase
    GrafoDirigido.
  def to_s
    s = "Grafo Dirigido: "
    vertices.each { |v| s += v.to_s }
    s
  end
end
```

Nota : Se adjunta un .tgz con las implementaciones realizadas y corridas pruebas de los mismos.