TTPS opción Ruby

Práctica 2

En esta segunda práctica del taller aplicaremos lo visto sobre el lenguaje Ruby, analizando distintas situaciones con los elementos fundamentales del mismo: los métodos, las clases y los módulos, los bloques, y los enumeradores.

Métodos

1. Implementá un método que reciba como parámetro un arreglo de números, los ordene y devuelva el resultado. Por ejemplo:

   ```
   ordenar_arreglo([1, 4, 6, 2, 3, 0, 10])
   # => [0, 1, 2, 3, 4, 6, 10]
   ```

2. Modificá el método anterior para que en lugar de recibir un arreglo como único parámetro, reciba todos los números como parámetros separados. Por ejemplo:

   ```
   ordenar(1, 4, 6, 2, 3, 5, 0, 10, 9)
   # => [0, 1, 2, 4, 5, 6, 9, 10]
   ```

3. Suponé que se te da el método que implementaste en el ejercicio anterior para que lo uses a fin de ordenar un arreglo de números que te son provistos en forma de arreglo. ¿Cómo podrías invocar el método? Por ejemplo, teniendo la siguiente variable con los números a ordenar:

   ```
   entrada = [10, 9, 1, 2, 3, 5, 7, 8]
   # Dada `entrada', invocar a #ordenar utilizando sus valores para ordenarlos
   ```

4. Escribí un método que dado un número variable de parámetros que pueden ser de cualquier tipo, imprima en pantalla la cantidad de caracteres que tiene su representación como `String` y la representación que se utilizó para contarla.

   > Nota: Para convertir cada parámetro a string utilizá el método #to_s presente en todos los objetos.

   Por ejemplo:

```
longitud(9, Time.now, 'Hola', {un: 'hash'}, :ruby)
# Debe imprimir:
# "9" --> 1
# "2015-09-07 21:04:10 +0000" --> 25
# "Hola" --> 4
# {:un=>"hash"} --> 13
# ruby --> 4
```

5. Implementá el método `cuanto_falta?` que opcionalmente reciba como parámetro un objeto `Time` y que calcule la cantidad de minutos que faltan para ese momento. Si el parámetro de fecha no es provisto, asumí que la consulta es para la medianoche de hoy.

   Por ejemplo:

   ```
   cuanto_falta? Time.new(2015, 12, 31, 23, 59, 59)
   # Debe retornar la cantidad de minutos que faltan para las 23:59:59 del 31/12/
   2015
   cuanto_falta?
   # Debe retornar la cantidad de minutos que faltan para la medianoche de hoy
   ```

6. Analizá el siguiente código e indicá qué problema(s) puede tener.

```
# Tira un dado virtual de 6 caras
def tirar_dado
 rand 1..6
end

# Mueve la ficha de un jugador tantos casilleros como indique el dado en un ta
blero virtual de 40 posiciones.
# Si no se recibe la cantidad de casilleros, aprovecho el valor por defecto pa
ra ese parámetro para evitar tener que
# llamar a #tirar_dado dentro del cuerpo del método.
def mover_ficha(fichas, jugador, casilleros = tirar_dado)
 fichas[jugador] += casilleros
 if fichas[jugador] > 40
   puts "Ganó #{jugador}!!"
   true
 else
   puts "#{jugador} ahora está en el casillero #{fichas[jugador]}"
   fichas[jugador]
 end
end

posiciones = { azul: 0, rojo: 0, verde: 0 }

finalizado = false
until finalizado
 ['azul', 'rojo', 'verde'].shuffle.each do |jugador|
   finalizado = mover_ficha(posiciones, jugador)
 end
end
```

7. Modificá el código anterior para, acorde a tu análisis, corregir los problemas que pueda tener.

> Nota: asumí que el juego debe terminar al momento que el primer jugador supera la posición 40 en el tablero.

Clases y módulos

1. Modelá con una jerarquía de clases algo sencillo que represente la siguiente situación:
   - Tres tipos de vehículo: `Auto`, `Moto` y `Lancha`
   - Los tres tipos arrancan usando una llave.
   - El auto, adicionalmente, requiere que no esté puesto el freno de mano y que el cambio esté en punto muerto. La moto, por otra parte, requiere una patada (sin la llave). La lancha arranca con la llave y listo.
   - El arranque de los tres vehículos se prueba en un taller. La especificación de `Taller` es la siguiente:

```
class Taller
def probar(objeto)
 objeto.arrancar
end
end
```

Suponé que, posteriormente, el taller necesita probar una motosierra. ¿Podrías hacerlo? ¿Cómo? ¿Qué concepto del lenguaje estás usando para poder realizar esto?

2. ¿Qué diferencia hay entre el uso de `include` y `extend` a la hora de incorporar un módulo en una clase?

    1. Si quisieras usar un módulo para agregar métodos de instancia a una clase, ¿qué forma usarías a la hora de incluirlo en la clase?
    2. Si en cambio quisieras usar un módulo para agregar métodos de clase, ¿qué forma usarías a la hora de incluir el módulo en la clase?

3. Implementá el módulo `Reverso` para utilizar como *Mixin* e incluilo en alguna clase para probarlo. `Reverso` debe contener los siguientes métodos:

    1. `#di_tcejbo` : Imprime el `object_id` del receptor en espejo (en orden inverso).
    2. `#ssalc` : Imprime el nombre de la clase del receptor en espejo.

4. Implementá el Mixin `Countable` que te permita hacer que cualquier clase cuente la cantidad de veces que los métodos de instancia definidos en ella es invocado. Utilizalo en distintas clases, tanto desarrolladas por vos como clases de la librería standard de Ruby, y chequeá los resultados. El Mixin debe tener los siguientes métodos:

    1. `count_invocations_of(sym)` : método de clase que al invocarse realiza las tareas necesarias para contabilizar las invocaciones al método de instancia cuyo nombre es `sym` (un símbolo).
    2. `invoked?(sym)` : método de instancia que devuelve un valor booleano indicando si el método llamado `sym` fue invocado al menos una vez en la instancia receptora.
    3. `invoked(sym)` : método de instancia que devuelve la cantidad de veces que el método identificado por `sym` fue invocado en la instancia receptora.

    Por ejemplo:

```
# Ejemplo de uso de Countable
class Greeter
  # Incluyo el Mixin
  include Countable

  def hi
    puts 'Hey!'
  end

  def bye
    puts 'See you!'
  end

  # Indico que quiero llevar la cuenta de veces que se invoca el método #hi
  count_invocations_of :hi
end

a = Greeter.new
b = Greeter.new

a.invoked? :hi
# => false
b.invoked? :hi
# => false

a.hi
# Imprime "Hey!"

a.invoked :hi
# => 1
b.invoked :hi
# => 0
```

> Nota: para simplificar el ejercicio, asumí que los métodos a contabilizar no reciben parámetros.
>
> Tips: investigá `Module#alias_method` y `Module#included`.

5. Dada la siguiente clase *abstracta* `GenericFactory`, implementá subclases de la misma que permitan la creación de instancias de dichas clases mediante el uso del método de clase `.create`, de manera tal que luego puedas usar esa lógica para instanciar objetos sin invocar directamente el constructor `new`.

```
class GenericFactory
  def self.create(**args)
    new(**args)
  end

  def initialize(**args)
    raise NotImplementedError
  end
end
```

6. Modificá la implementación del ejercicio anterior para que `GenericFactory` sea un módulo que se incluya como *Mixin* en las subclases que implementaste. ¿Qué modificaciones tuviste que hacer en tus clases?

7. Extendé las clases `TrueClass` y `FalseClass` para que ambas respondan al método de instancia `opposite`, el cual en cada caso debe retornar el valor opuesto al que recibe la invocación al método. Por ejemplo:

```
false.opposite
# => true
true.opposite
# => false
true.opposite.opposite
# => true
```

8. Analizá el script Ruby presentado a continuación e indicá:

    1. ¿Qué imprimen cada una de las siguientes sentencias? ¿De dónde está obteniendo el valor?
        1. `puts A.value`
        2. `puts A::B.value`
        3. `puts C::D.value`
        4. `puts C::E.value`
        5. `puts F.value`

    2. ¿Qué pasaría si ejecutases las siguientes sentencias? ¿Por qué?
        1. `puts A::value`
        2. `puts A.new.value`
        3. `puts B.value`
        4. `puts D.value`
        5. `puts C.value`

Bloques

1. Escribí un método `da_nil?` que reciba un bloque, lo invoque y retorne si el valor de retorno del

bloque fue `nil`. Por ejemplo:

```
da_nil? { }
# => true
da_nil? do
  'Algo distinto de nil'
end
# => false
```

2. Implementá un método que reciba como parámetros un `Hash` y `Proc`, y que devuelva un nuevo `Hash` cuyas las claves sean los valores del `Hash` recibido como parámetro, y cuyos valores sean el resultado de invocar el `Proc` con cada clave del `Hash` original.

   Por ejemplo:

   ```
   hash = { 'clave' => 1, :otra_clave => 'valor' }
   procesar_hash(hash, ->(x) { x.to_s.upcase })
   # => { 1 => 'CLAVE', 'valor' => 'OTRA_CLAVE' }
   ```

3. Implementá un método que reciba un número variable de parámetros y un bloque, y que al ser invocado ejecute el bloque recibido pasándole todos los parámetros que se recibieron encapsulando todo esto con captura de excepciones de manera tal que si en la ejecución del bloque se produce alguna excepción, proceda de la siguiente forma:

   - Si la excepción es de clase `RuntimeException`, debe imprimir en pantalla `"Algo salió mal..."`, y retornar `:rt`.
   - Si la excepción es de clase `NoMethodError`, debe imprimir `"No encontré un método: "` más el mensaje original de la excepción que se produjo, y retornar `:nm`.
   - Si se produce cualquier otra excepción, debe imprimir en pantalla `"¡No sé qué hacer!"`, y relanzar la excepción que se produjo.

   En caso que la ejecución del bloque sea exitosa, deberá retornar `:ok`.

   > Tips: Leer sobre las sentencias `raise` y `rescue`.

Enumeradores

1. Si no lo hiciste de esa forma en la práctica 1, escribí un enumerador que calcule la serie de Fibonacci.

2. ¿Qué son los *lazy enumerators*? ¿Qué ventajas les ves con respecto al uso de los enumeradores que no son *lazy*?

   > Tip: Analizalo pensando en conjuntos grandes de datos.

3. Extendé la clase `Array` con el método `randomly` que funcione de la siguiente manera:

    - Si recibe un bloque, debe invocar ese bloque con cada uno de los elementos del arreglo en orden aleatorio.
    - Si no recibe un bloque, debe devolver un enumerador que va arrojando, de a uno, los elementos del arreglo en orden aleatorio.

4. Suponé que tenés la clase `Image` detallada más abajo para realizar procesamiento de imágenes. Esta clase representa en sí misma una imagen y dispone de métodos para aplicarle diversos filtros (`filter_a` , `filter_b` , `filter_c` , `filter_d` , `filter_e` y `filter_f`) generando y retornando una nueva instancia de `Image` cada vez que se los invoca. Por ejemplo, el siguiente código toma una imagen inicial y retorna otra instancia de `Image` que representa la imagen original con los filtros `A` , `C` y `E` aplicados:

    ```
    image = Image.new
    image.filter_a.filter_c.filter_e
    ```

    Dada la siguiente implementación de la clase `Image` , se te pide que la modifiques para que su uso consuma menos recursos (principalmente procesamiento) haciendo que los cálculos de los filtros se hagan únicamente cuando se pida la información de la cabecera de la imagen (representada en este caso mediante el método `Image#header_bytes`).

```
require 'matrix'

class Image
  attr_accessor :data, :size

  def initialize(data = nil, size = 1024)
    self.size = size
    self.data = data || Matrix.build(size) { Math::PI }
  end

  def header_bytes
    Matrix.rows([data.first(size)])
  end

  # Distintos filtros de imágenes:

  def filter_a
    Image.new data.map { |e| e ** 1.2 }
  end

  def filter_b
    Image.new data.map { |e| e ** 1.4 }
  end

  def filter_c
    Image.new data.map { |e| e ** 1.8 }
  end

  def filter_d
    Image.new data.map { |e| e ** 2 }
  end

  def filter_e
    Image.new data.map { |e| e ** 2.2 }
  end

  def filter_f
    Image.new data.map { |e| e ** 2.4 }
  end

  #- Fin de filtros

  def all_filters
    ('a'..'f').inject(self) do |pipe, type|
      pipe.public_send "filter_#{type}"
    end
  end
end
```

Las modificaciones que hagas deberían hacer que se cumpla lo siguiente:

```
image = Image.new
image.filter_a.filter_c.filter_e              # => Esto no realiza ningún cálc
ulo.
image.filter_a.filter_c.filter_e.header_bytes # => Esto sí realiza cálculos pa
ra obtener la info de la cabecera.
```

> Tip 1: Para este ejercicio es útil desactivar el `echo` de `irb` (si lo usás para probar el ejercicio). Para esto, podés escribir en la consola: `ruby irb_context.echo = false` Tip 2: La solución es vaga.