Práctica 2: Clases, módulos, métodos, bloques, enumeradores y excepciones

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

Métodos

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

```ruby
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:

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

3. Suponé que se te da el método que implementaste en el ejercicio 2 para que lo uses a fin de ordenar un conjunto 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:

```ruby
entrada = [10, 9, 1, 2, 3, 5, 7, 8]
# Dada `entrada', invocá a #ordenar utilizando los valores que
    contiene la variable
ordenar(entrada) # <= Esto no funciona. Corregí esta invocación
    para que funcione.
```

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.

   Por ejemplo:

```ruby
longitud(9, Time.now, 'Hola', {un: 'hash'}, :ruby)
# Debe imprimir:
# "9" --> 1
# "2025-09-14 13:22:10 +0000" --> 25
# "Hola" --> 4
# {:un=>"hash"} --> 13
# ruby --> 4
```

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

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(2032, 12, 31, 23, 59, 59)
# => Retorna la cantidad de minutos que faltan para las 23:59:59
    del 31/12/2032
cuanto_falta?
# => Retorna la cantidad de minutos que faltan para la medianoche
    de hoy
```

Clases y módulos

6. Modelá con una jerarquía de clases la siguiente situación:

   Los usuarios finales de una aplicación tienen los atributos básicos que permiten identificarlos (usuario, clave, email - los que consideres necesarios), y un rol que determina qué operaciones pueden hacer. Los roles posibles son: Lector, Redactor, Director y Administrador. Cada usuario gestiona Documentos según su rol le permita, acorde a las siguientes reglas:

   - Los Lectores pueden ver cualquier Documento que esté marcado como público.
   - Los Redactores pueden hacer todo lo que los Lectores y además pueden cambiar el contenido de los Documentos que ellos crearon.
   - Los Directores pueden ver y cambiar el contenido de cualquier documento (público o privado, y creado por cualquier usuario), excepto aquellos que hayan sido borrados.
   - Los Administradores pueden hacer lo mismo que los directores y además pueden borrar Documentos.

   Utilizando el siguiente código para la clase `Documento`, implementá las clases que consideres necesarias para representar a los usuarios y sus roles, completando la funcionalidad aquí presente:

```
class Documento
  attr_accessor :creador, :contenido, :publico, :borrado

  def initialize(usuario, publico = true, contenido = '')
    self.creador = usuario
    self.publico = publico
    self.contenido = contenido
    self.borrado = false
  end

```

```
11    def borrar
12      self.borrado = true
13    end
14
15    def puede_ser_visto_por?(usuario)
16      usuario.puede_ver? self
17    end
18
19    def puede_ser_modificado_por?(usuario)
20      usuario.puede_modificar? self
21    end
22
23    def puede_ser_borrado_por?(usuario)
24      usuario.puede_borrar? self
25    end
26  end
```

7. Luego de implementar el ejercicio anterior, modificalo para que los usuarios implementen el método `#to_s` que debe retornar el atributo usuario (o email, según hayas decidido utilizar) y el rol que posee. Por ejemplo:

```
1  lector.to_s
2  # => "elhector@example.org (Lector)"
3  administrador.to_s
4  # => "admin@example.org (Administrador)"
```

8. ¿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 incorporar el módulo a la clase?
    2. Si en cambio quisieras usar un módulo para agregar métodos de clase, ¿qué forma usarías en ese caso?

9. 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.

10. 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, su uso podría ser el siguiente:

```ruby
class Greeter
  include Countable # Incluyo el Mixin

  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.
>
> Tip: investigá `Module#alias_method` y `Module#included`.

11. 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`**.

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

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

12. 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?

13. 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:

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

14. Analizá el siguiente script e indicá:

```ruby
VALUE = 'global'

module A
  VALUE = 'A'

  class B
    VALUE = 'B'

    def self.value
      VALUE
    end

    def value
      'iB'
    end
  end

  def self.value
    VALUE
  end
end

```

```ruby
class C
  class D
    VALUE = 'D'

    def self.value
      VALUE
    end
  end

  module E
    def self.value
      VALUE
    end
  end

  def self.value
    VALUE
  end
end

class F < C
  VALUE = 'F'
end
```

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 C::D.value`
    5. `puts C.value`
    6. `puts F.superclass.value`

Bloques

15. Escribí un método `da_nil?` que reciba un bloque, lo invoque y retorne si el valor de retorno del bloque fue `nil`. Por ejemplo:

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

16. Implementá un método que reciba como parámetros un `Hash` y un `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:

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

17. 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 `RuntimeError`, debe imprimir en pantalla `"Hay algo mal que no anda bien"`, y retornar `:rt`.
    - Si la excepción es de clase `NoMethodError`, debe imprimir `"Y este método?"` más el mensaje original de la excepción que se produjo, y retornar `:nm`.
    - Si se produce cualquier otra excepción, debe imprimir en pantalla `"Y ahora?"`, y relanzar la excepción que se produjo.

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

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

Enumeradores

18. ¿Qué son los *lazy enumerators*? ¿Qué ventajas ves en ellos con respecto al uso de los enumeradores que no son *lazy*?

    > Tip: Analizalo pensando en conjuntos grandes de datos.

19. Implementá una clase `Palabra` que funcione de la siguiente manera:

    - La clase se instancia con un argumento obligatorio (un `String`) que será la palabra que represente.

- Si la palabra que representa contiene caracteres y al menos un espacio, la instanciación debe arrojar una excepción `NoEsUnaPalabra` con el mensaje `"<palabra> no es una palabra"` (donde `<palabra>` es el valor recibido como argumento en el constructor.
- Si la palabra que representa es un `String` vacío (`""`, `" "` son dos ejemplos de `String`s vacíos), debe arrojar una excepción `EsUnStringVacio` con el mensaje `"Es un string vacío"`.
- La clase debe implementar los siguientes métodos de instancia:
  - `#vocales` que debe retornar las vocales que contiene la palabra que representa, sin repeticiones.
  - `#consonantes` que debe retornar las consonantes que contiene la palabra, sin repeticiones.
  - `#longitud` que debe retornar la cantidad de caracteres que tiene la palabra.
  - `#es_panvocalica?` que debe retornar un valor booleano indicando si la palabra es panvocálica (o pentavocálica), es decir si contiene las 5 vocales.
  - `#es_palindroma?` que debe retornar un valor boolean indicando si la palabra es un palíndromo, es decir si se lee igual en un sentido que en otro, teniendo al menos 3 letras.
  - `#gritando` que debe retornar la palabra que representa en mayúsculas.
  - `#en_jaquer` que debe retornar la palabra que representa con las vocales cambiadas por números (`"a"` por `"4"`, `"e"` por `"3"`, `"i"` por `"1"`, `"o"` por `"0"` y `"u"` por `"2"`).

> Tip: para simplificar la implementación, podés asumir que las palabras no tendrán acento.