Práctica 2

8. ¿Qué diferencia hay entre el uso de include y extend a la hora de incorporar un módulo en una clase?
   - 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?
   - Si en cambio quisieras usar un módulo para agregar métodos de clase, ¿qué forma usarías en ese caso?

   - Include
     - Agrega los métodos del módulo como métodos de instancia. Estarán disponibles en los objetos creados a partir de la clase.
   - Extend
     - Agrega los métodos del módulo como métodos de clase. Estarán disponibles directamente en la clase que los incorpora, sin necesidad de instanciar objetos.
   - Si quisieras usar un módulo para agregar métodos de instancia a una clase usuaria include.

```ruby
module Saludable
  def saludar
    "Hola!"
  end
end

class Persona
  include Saludable
end

p = Persona.new
puts p.saludar    # => "Hola!"
```

   - Si quisieras usar un módulo para agregar métodos de clase usaría extend.

```ruby
module Identificable
  def tipo
    "Soy una clase"
  end
end

class Usuario
  extend Identificable
end
```

```ruby
puts Usuario.tipo  # => "Soy una clase"
```

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

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?
    a. puts A.value
        i. A
        ii. Llama al método de clase self.value del módulo A y dentro de A.value, VALUE hace referencia a la constante más cercana en scope, A::VALUE = 'A'
    b. puts A::B.value
        i. B
        ii. Llama al método de clase self.value de la clase B dentro de A y dentro de B.value, VALUE busca en el scope de B donde la constante más cercana es B::VALUE = 'B'
    c. puts C::D.value
        i. D
        ii. Llama al método de clase self.value de la clase D dentro de C y dentro de D.value, VALUE hace referencia a la constante más cercana en scope, D::VALUE = 'D'
    d. puts C::E.value
        i. global
        ii. Llama al método de clase self.value del módulo E dentro de C. Como E no tiene VALUE definido, busca en los scopes externos. El Primer scope externo es C que tampoco tiene VALUE definido como constante. El siguiente scope es el global con VALUE = 'global'. Ruby permite que un módulo acceda a constantes externas si no hay definición local.
    e. puts F.value
        i. global
        ii. F hereda de C. Llama a self.value de C porque F no redefine el método self.value. Dentro de C.value, VALUE busca constante en scope de C. C no tiene VALUE definido, entonces busca en scope externo que es el global con VALUE = 'global'.
        iii. Aunque F define VALUE = 'F' como constante, esto no afecta al método heredado C.value, porque dentro de C.value, el scope

de la constante es donde se definió el método, no la clase que hereda.

2. ¿Qué pasaría si ejecutases las siguientes sentencias? ¿Por qué?
    a. puts A::value
        i. A
        ii. :: se usa para resolver constantes y scope, no es la forma estándar de llamar métodos, pero se puede hacer (aunque depende mucho de la version de Ruby).
    b. puts A.new.value
        i. error
        ii. No se puede hacer .new sobre un modulo
    c. puts B.value
        i. error
        ii. B no está definida en el scope global, está dentro de A. B sola no existeputs C::D.value
    d. puts C::D.value
        i. D
        ii. Llama al método de clase self.value de la clase D dentro de C y dentro de D.value, VALUE hace referencia a la constante más cercana en scope, D::VALUE = 'D'
    e. puts C.value
        i. global
        ii. C.value llama al método de clase self.value de C. Dentro de C.value, VALUE no está definido dentro de C por lo que se va al scope externo que es el global.
    f. puts F.superclass.value
        i. global
        ii. F < C, la F.superclass es C y C.value devuelve 'global'.

:: Principalmente se usa para acceder a constantes, clases o módulos dentro de otro módulo/clase. También puede llamar métodos, pero solo si son métodos de clase o de módulo.

. Se usa para llamar métodos ya sea métodos de clase o métodos de instancia (sobre objetos)

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

    Es un objeto que permite iterar elementos de manera perezosa donde en lugar de generar o procesar todos los elementos de una colección de una sola vez, produce cada elemento bajo demanda. Se obtiene a partir de un Enumerator común con el método .lazy

```ruby
# Enumerador normal
(1..Float::INFINITY).map { |x| x * 2 }    # se cuelga: intenta generar infinitos elementos

# Lazy enumerator
(1..Float::INFINITY).lazy.map { |x| x * 2 }.first(5)
# => [2, 4, 6, 8, 10]
```

    Ventajas:

    - Eficiencia en memoria: un enumerador normal genera y guarda todos los elementos en memoria antes de devolver el resultado, en cambio el lazy enumerator solo genera lo necesario en el momento, evitando usar memoria extra.
    - Posibilidad de trabajar con colecciones infinitas: un enumerador común no puede manejar rangos infinitos (1..Float::INFINITY), porque intenta calcular todo, en cambio un lazy enumerator los maneja sin problema, ya que procesa solo hasta el límite que se le pide.
    - Optimización de rendimiento: si se encadenan operaciones (map, select, reject, etc.), en un enumerador normal se evalúa toda la colección en cada paso. Por su parte, con lazy cada elemento pasa por toda la cadena de transformaciones antes de pasar al siguiente, lo cual puede ser más eficiente cuando solo necesitás una parte del resultado.