Clases, bloques, enumeradores y excepciones

Clases

Creamos la clase

```
class BookInStock
end
```

Lo probamos

```
a_book = BookInStock.new
another_book = BookInStock.new
```

Manejo de clases

Observaciones

- Se crean dos objetos diferentes de la clase `BookInStock`.
- Podríamos decir en esta primer instancia que son el mismo libro, o iguales porque **nada los distingue**.
- Lo solucionamos obligando que la inicialización indique aquellos datos que distinga al libro.

Manejo de clases

Variables de instancia

```
class BookInStock
  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
  end
end
```

Manejo de clases

- El método `initialize` es especial en Ruby
- Cuando se invoca el método `new`, Ruby aloca memoria para alojar un objeto no inicializado y luego invoca al método `initialize` **pasándole cada parámetro que fue enviado a `new`**.
- Entonces `initialize` nos permite configurar el estado inicial de nuestros objetos.

Manejo de clases

En el método initialize

- Se utilizan variables de instancia: comienzan con @.
- Las variables `@isbn` e `isbn` son diferentes.
- Se realiza una pequeña validación implícita:
  - El método `Float` toma un argumento y lo convierte a `float`, terminando el programa si falla la conversión

Manejo de clases

Implementamos to_s

```
class BookInStock
  def to_s
    "ISBN: #{@isbn}, price: #{@price}"
  end
end
```

Manejo de clases

Objetos y sus atributos

- Un objeto como el mostrado anteriormente no permite que nadie acceda a sus variables.
- Si bien es algo positivo encapsular, si no permitimos acceder a los datos que mantienen el estado del objeto, el mismo se vuelve inútil.
- A las *ventanas* de acceso a los objetos las denominaremos **atributos**.
- Modificaremos nuestra clase de `BookInStock` con el fin de agregar atributos para `isbn` y `price` así podemos contabilizarlos.

Manejo de clases

Getters

```
class BookInStock
  def isbn
    @isbn
  end

  def price
    @price
  end
end
```

Manejo de clases

A los atributos anteriores se los denomina **accesor** porque mapean directamente con las variables de instancia. Ruby provee un shortcut: **`attr_reader`**.

```ruby
class BookInStock
  attr_reader :isbn, :price
  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
  end
end
```

- *Notar que se utilizan **símbolos**.*
- *`attr_reader` **no define variables de instancia**, sólo los métodos de acceso.*

Bloques

El código define un generador de la secuencia de Fibonacci hasta un máximo dado, y usa yield para que un bloque externo decida qué hacer con cada número (en este caso, imprimirlo).

```
def fib_up_to(max)
  i1, i2 = 1, 1
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end

fib_up_to(1000) {|f| print f, " " }
```

Uso del valor retornado de un bloque

```ruby
class Array
  def my_find
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end

(1..200).to_a.my_find {|x| x%5 == 0}

(1..200).to_a.my_find {|x| x == 0}
```

Ejemplos bloques

Escribí un método **da_nil?** que reciba un bloque, lo invoque y retorne si el valor de retorno del
bloque fue nil

Ejemplos

```ruby
def da_nil?
  resultado = yield
  resultado.nil?
end
```

Los iteradores

- Las clases que implementan colecciones, como `Array` *hacen lo que hacen mejor:*
  - Acceder a los elementos que contienen.
- El comportamiento de qué hacer con cada elemento lo delegan a la aplicación:
  - Permitiendo que nos concentremos sólo en un requerimiento particular.
  - En los casos anteriores (`find`), sería encontrar un elemento para el cual el criterio sea verdadero.

each y collect

- El iterador `each` es el más simple.
  - Solo invoca `yield` para cada elemento.
- El iterador `collect` también conocido como `map`.
  - Invoca `yield` para cada elemento. El resultado lo guarda en un nuevo arreglo que es **retornado**.

```
[ 1, 3, 5, 7, 9 ].each {|i| puts i }

['k','h','m','t','w'].collect {|x| x.succ }
```

Otros usos de iteradores

- Los iteradores no sólo se usan con array y hash.
- Su lógica es muy utilizada en clases de entrada / salida para retornar líneas sucesivas o bytes.

```
f = File.open("testfile")
f.each { |line| puts "The line is: #{line}"}
f.close
```

```
f = File.open("testfile")
f.each_with_index do |line, index|
  puts "Line #{index} is: #{line}"
end
f.close
```

inject

- Este iterador tiene un nombre *raro*.
- Permite acumular un valor a lo largo de los miembros de una colección.
- Recibe un parámetro que es el valor inicial para comenzar a acumular.
  - Si no se especifica **toma el primer elemento de la colección**.

```
[1,3,5,7].inject(0) {|sum, element| sum+element}
[1,3,5,7].inject    {|sum, element| sum+element}
[1,3,5,7].inject(1) {|prod, element| prod*element}
[1,3,5,7].inject    {|prod, element| prod*element}
```

Problema de iteradores

- Los iteradores son muy cómodos pero:
  - Son parte de la colección y no una clase a parte.
  - En otros lenguajes (como Java), las colecciones no implementan sus iteradores, sino que son clases separadas (como por ejemplo la interfaz Iterator de Java).
  - Es complicado iterar dos colecciones simultáneamente.

Enumeradores

- La solución: clase `Enumerator`.
- Se obtiene de una colección con el método `to_enum` o `enum_for`.

```
a = [ 1, 3, "cat" ]
h = { dog: "canine", fox: "lupine" }
# Create Enumerators
enum_a = a.to_enum
enum_h = h.to_enum
enum_a.next   # => 1
enum_h.next   # => [ :dog, "canine" ]
enum_a.next   # => 3
enum_h.next   # => [ :fox, "lupine" ]
```

Enumerators e iteradores

Si un iterador se utiliza sin bloque, entonces retorna un Enumerator

```
a = [1,2,3].each
a.next
```

El método loop con enumeradores

- Ejecuta el código que se encuentra dentro del bloque.
- Se puede salir con break cuando se cumple una condición.
- Si hay iteradores, `loop` terminará cuando el Enumerator se quede sin valores.

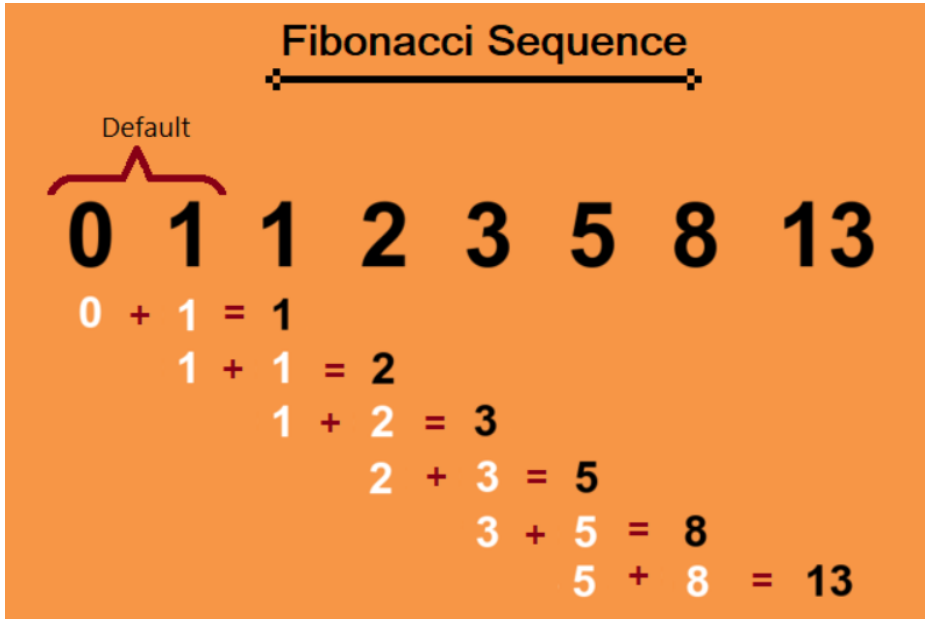
```ruby
loop { puts "Hola" }
```

```ruby
i=0
loop do
  puts i += 1
  break if i >= 10
end
```

```ruby
short_enum = [1, 2, 3].to_enum
long_enum = ('a'..'z').to_enum
loop { puts "#{short_enum.next} - #{long_enum.next}" }
```

Ejemplo enumeradores

Escribí un enumerador que calcule la serie de Fibonacci.



Ejemplo enumeradores

```ruby
fibonacci = Enumerator.new do |caller|
  i1, i2 = 1, 1
  loop do
    caller.yield i1
    i1, i2 = i2, i1+i2
  end
end

6.times { puts fibonacci.next }
```

Excepciones

- Las excepciones permiten empaquetar en un objeto información sobre un error.
- El objeto `Exception` se propagará hacia arriba en la pila de ejecución hasta que el sistema detecte código que sepa manejar dicha excepción.

Excepciones

- Ruby define una jerarquía de excepciones que son subclase de `Exception`.
- Al lanzar una excepción, es posible hacerlo con cualquiera de las subclases de `Exception` o con una clase propia.
- Toda excepción tiene asociado un mensaje y una traza de ejecución.
  - Si definimos excepciones propias, podemos agregar información específica

Excepciones

Analizamos el siguiente código

```ruby
require 'open-uri'

web_page = URI.open("https://www.ruby-lang.org/en/documentation/")
output = File.open("ruby.html", "w")
while line = web_page.gets
  output.puts line
end
output.close
```

- ¿Qué sucede si ocurre un error en la mitad de la transferencia?
- No queremos guardar una página por la mitad en el archivo de salida

Excepciones

Agregamos el manejador de excepción

```
page = 'unip'
file_name = "#{page}.html"
output = File.open(file_name, "w")
begin
  web_page = URI.open("https://www.ruby-lang.org/en/#{page}")
  while line = web_page.gets
    output.puts line
  end
  output.close
rescue Exception
  STDERR.puts "Failed to download #{page}: #{$!}"
  output.close
  File.delete(file_name)
  raise
end
```

Excepciones

- Cuando sucede una excepción se ubica una referencia al objeto con la excepción asociada en la variable global **`$!`**.
- Luego de cerrar y eliminar el archivo, se invoca a `raise` sin parámetros, que relanza la excepción en **`$!`**.

Como funciona rescue

- La decisión de qué `rescue` utilizar, es similar al caso de un `case`.
- Cada `rescue` compara la excepción lanzada con cada uno de los parámetros nombrados:
  - La comparación es: `parámetro == $!`
  - Esto significaría que si el tipo de la excepción lanzada coincide con el del parámetro.

Como funciona rescue

- Si se omiten parámetros, se compara con `StandardError`.
- Si no machea con ningún parámetro, sale del bloque `begin/end` buscando en el método que invocó un manejador para la misma, y así siguiendo hacia arriba en la pila.
- Casi siempre usaremos nombre de clases como parámetros a `rescue`, pero podemos usar expresiones que retornen una subclase de `Exception`.

Como funciona rescue

- Varias veces necesitamos ejecutar determinado código al finalizar un método de forma segura, es decir independientemente de si se produce un error en la mitad:
  - Por ejemplo, tenemos un archivo abierto, que necesitamos cerrar antes de finalizar el bloque.
- El código bajo `ensure` se ejecutará siempre, haya sido una ejecución exitosa o con algún problema.

Excepciones

Un ejemplo ensure

```
f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
ensure
  f.close
end
```

Excepciones

- El `else` aplica cuando ninguno de los `rescue` manejan la excepción.
- Tener cuidado porque `ensure` ejecutará siempre, incluso cuando no se produce un error.

```
f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
else
  puts "Congratulations-- no errors!"
ensure
  f.close
end
```

Excepciones

`Volver a empezar...`

- A veces podemos corregir una causa de excepción.
- Para estos casos, podemos usar `retry` para volver a ejecutar el bloque `begin/end`.
- Es muy factible caer en loops infinitos.

Excepciones

Podemos lanzar excepciones usando el método `Kernel.raise`

```
raise
raise "bad mp3 encoding"
raise InterfaceException, "Keyboard failure", caller
```

- La primer forma **relanza una excepción** si la hubiere, o **`RuntimeError`** si no. Usualemnte dentro de `rescue` para el primer caso.
- El segundo ejemplo, lanza **`RuntimeError`** con el mensaje indicado.
- El tercer ejemplo, utiliza el primer parámetro para crear un excepción con el segundo parámetro usado como mensaje y la pila de ejecución en el tercer parámetro. **`Kernel.caller`** genera la traza de ejecución.

Tarea para la próxima clase

Implementa 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.
#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.
#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.

Que excepción podríamos agregar?