Ruby's Enumerable module

David Grayson
Las Vegas Ruby Group
2012-03-21

Enumerable should be familiar!

- Included by Array, Hash, Range, Set, String#chars, String#bytes, maybe ActiveRecord::Relation

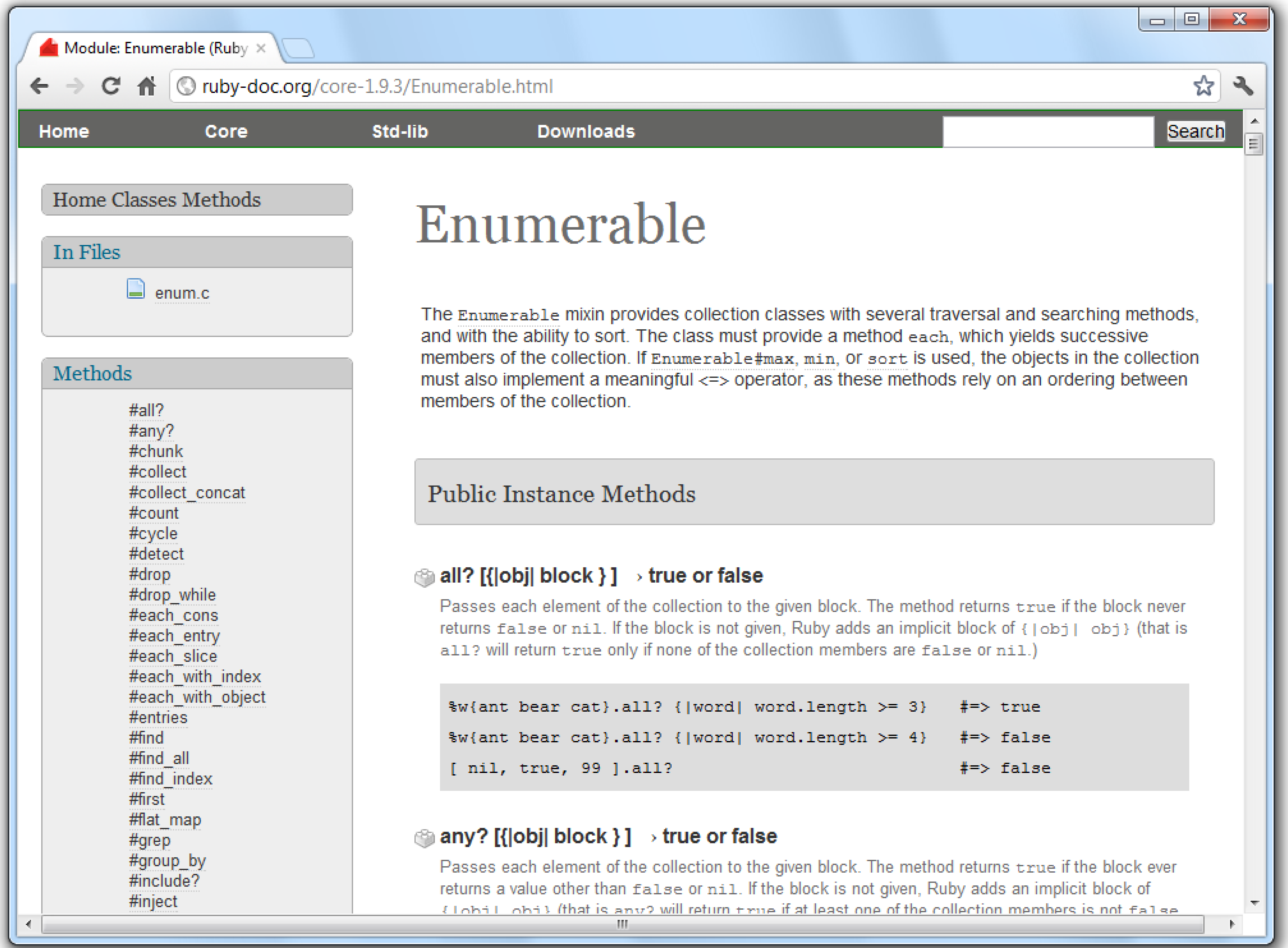
```
object.is_a? Enumerable
```

What is an Enumerable?

- Represents a series of objects.
- Can be lazily generated.
- Can be infinite.

Enumerable provides methods:

- #all?
- #any?
- #chunk
- #collect
- #collect_concat
- #count
- #cycle
- #detect
- #drop
- #drop_while
- #each_cons
- #each_entry
- #each_slice
- #each_with_index
- #each_with_object
- #entries
- #find
- #find_all
- #find_index
- #first
- #flat_map
- #grep
- #group_by
- #include?
- #inject
- #map
- #max
- #max_by
- #member?
- #min
- #min_by
- #minmax
- #minmax_by
- #none?
- #one?
- #partition
- #reduce
- #reject
- #reverse_each
- #select
- #slice_before
- #sort
- #sort_by
- #take
- #take_while
- #to_a
- #zip



Enumerable

The `Enumerable` mixin provides collection classes with several traversal and searching methods, and with the ability to sort. The class must provide a method `each`, which yields successive members of the collection. If `Enumerable#max`, `min`, or `sort` is used, the objects in the collection must also implement a meaningful <=> operator, as these methods rely on an ordering between members of the collection.

Public Instance Methods

all? [{|obj| block }] → true or false

Passes each element of the collection to the given block. The method returns `true` if the block never returns `false` or `nil`. If the block is not given, Ruby adds an implicit block of `{ |obj| obj }` (that is `all?` will return `true` only if none of the collection members are `false` or `nil`.)

```
%w{ant bear cat}.all? {|word| word.length >= 3}   #=> true
%w{ant bear cat}.all? {|word| word.length >= 4}   #=> false
[ nil, true, 99 ].all?                             #=> false
```

any? [{|obj| block }] → true or false

Passes each element of the collection to the given block. The method returns `true` if the block ever returns a value other than `false` or `nil`. If the block is not given, Ruby adds an implicit block of `{ |obj| obj }` (that is `any?` will return `true` if at least one of the collection members is not `false`

How to make an Enumerable

- Easy way: just make an Array
- Need to know all values ahead of time.
- Arrays can't be infinite!

How to make an Enumerable

```ruby
class HouseCollection
  include Enumerable
  def each
    yield house
    # ... insert complex code
    yield house
  end
end

enum = HouseCollection.new
```

How to make an Enumerable

```
class HouseCollection
  # class writer forgot to include Enumerable
  def each
    yield house
    # ... insert complex code
    yield house
  end
end

enum = HouseCollection.new.to_enum
```

How to make an Enumerable

```
class HouseCollection
  # class writer forgot to include Enumerable
  def each_house
    yield house
    # ... insert complex code
    yield house
  end
end

enum = HouseCollection.new.to_enum(:each_house)
```

How to make an Enumerable

```ruby
enum = Enumerator.new do |y|
  y << 1
  y << 10
  y << 6
end
```

- Enumerable is a module
- Enumerator is a class that includes Enumerable.

Example Uses of Enumerables

Basic use

```
enum = "a".."f"

enum.to_a      # => ["a", "b", "c", "d", "e", "f"]
enum.entries   # => ["a", "b", "c", "d", "e", "f"]

enum.count                        # => 6
enum.count("b")                   # => 1
enum.count { |s| s <= "c" }  # => 3
```

Iteration

```
enum = 1..6

enum.each { |x| ... }
enum.each_entry { |x| ... }
# yields 1, 2, 3, 4, 5, 6

enum.each_cons(2) { |x, next_x| ... }
# yields [1,2], [2,3], [3,4] ...

enum.each_slice(3) { |x0, x1, x2| ... }
# yields [1,2,3], [4,5,6]

enum.each_with_index { |x, index| ... }
# yields [1, 0], [2, 1], [3, 2] ...

enum.reverse_each { |x| ... }
# yields 6, 5, 4, 3, 2, 1
```

Iteration with #cycle

```
players = ["alex", "bob", "caterina",
           "david", "errol", "fred"]

players.cycle { |player| ... }

# Equivalent to:
while true
  players.each do |player|
    ...
  end
end

# Can also specify number of cycles:
players.cycle(3) { |player| ... }
```

Iteration with Enumerator

```
enumerable = 1..3
enumerator = enumerable.to_enum

p enumerator.next # => 1
p enumerator.next # => 2
p enumerator.next # => 3
p enumerator.next # => StopIteration exception
```

- Enumerable is a module
- Enumerator is a class that includes Enumerable.

Asking questions

```
enum = [2, 5, 7, 10]

enum.include?(5)          # => true
enum.member?(5)           # => true

enum.all? { |x| x < 11 }   # => true
enum.none? { |x| x > 11 }  # => true

enum.any? { |x| x > 6 }    # => true
enum.one? { |x| x.even? }  # => false
```

Sorting

```
enum = [6, -1, 3, -4]

enum.sort   # => [-4, -1, 3, 6]
enum.min    # => -4
enum.max    # => 6
enum.minmax # => [-4, 6]
```

Advanced sorting

```
enum = [6, -1, 3, -4]

enum.sort_by &:abs      # => [-1, 3, -4, 6]
enum.sort_by { |x| x%10 }  # => [3, 6, -4, -1]
```

#min_by, #max_by, and
#minmax_by also available!

(Almost Always) too advanced sorting

~~`countries.sort { |c1,c2| c1.code <=> c2.code}`~~

```
countries.sort_by :&code
```

```
friends.sort { |a, b| arm_wrestle(a, b) }
```

#min, #max, and #minmax
can also take a block

Searching for one element

```
names = ["judd", "russ", "david", "paul", "ryan"]

names.find   { |n| n[1] == "a" } # => "david"
names.detect { |n| n[1] == "a" } # => "david"

names.find_index { |n| n[1] == "a" }  # => 2
names.find_index("david")             # => 2
```

Filtering by value

```
names = ["judd", "russ", "david", "paul", "ryan"]

names.select { |n| n[1] == "u" }
  # => ["judd", "russ"]

names.reject { |n| n.length < 5 }
  # => ["david"]

names.grep(/u/)
  # => ["judd", "russ", "paul"]

[1, 4.0, nil, Object, 5].grep(Integer)
  # => [1,5]
```

Filtering by position in series

```
days = ["mon", "tue", "wed",
  "thu", "fri", "sat", "sun"]

p days.first     # => "mon"
p days.first(2)  # => ["mon", "tue"]

p days.drop(5)   # => ["sat", "sun"]
p days.drop_while { |x| x != "sat" }
   # => ["sat", "sun"]

p days.take(2)   # => ["mon", "tue"]
p days.take_while { |x| x != "wed" }
   # => ["mon", "tue"]
```

Dividing into subsets: chunk

```
hand = ["7H", "AS", "KS", "JS", "9H"]

p hand.chunk{|c| c[1]}.each { |suit, cards| }
  # yields "H", ["7H"]
  #        "S", ["AS, "KS", JS"]
  #        "H", ["9H"]
```

- Order matters; chunks are consecutive
- `nil` and `:_separator` drop the element.
- `:_alone` puts the element in its own chunk.

Dividing into subsets: group_by

- Order does not matter!

```
hand = ["7H", "AS", "KS", "JS", "9H"]

hand.group_by { |c| c[1] }
  # => {
  #    "H"=>["7H", "9H"],
  #    "S"=>["AS", "KS", "JS"]
  # }
```

Dividing into subsets: partition

```
players = ["alex", "bob", "caterina",
           "david", "errol", "fred"]

teams = players.partition { |p| players.index(p).even? }
# => [ ["alex", "caterina", "errol"],
#      ["box", "david", "fred"] ]



# Cooler way:
teams = players.partition.with_index do |p, index|
  index.even?
end
```

Dividing into subsets: slice_before

- Block returns “true” => start of new chunk

```
(3..11).slice_before{ |n| n%5 == 0}.each{ |s| ... }
  # yields [3, 4]
  #        [5, 6, 7, 8, 9],
  #        [10, 11]
```

inject (a.k.a. reduce)

- Combines all the elements together.

```
enum = 1..4

enum.inject(:+)       # 1+2+3+4 => 10
enum.inject(0.5, :*) # 0.5*1*2*3*4 => 12.0

enum.inject { |memo, x| ... }
enum.inject(initial) { |memo, x| ... }
```

zip

- zips 2 or more enums together into one

```
team1.zip(team2) do |player1, player2|
  play_chess player1, player2
end
```

map and flat_map

```
require 'set'
names = Set.new ["richard hoppes"
                 "nicholas shook"]

p names.map &:upcase
# => ["RICHARD HOPPES", "NICHOLAS SHOOK"]

p names.map &:split
# => [["richard", "hoppes"], ["nicholas", "shook"]]

p names.flat_map &:split
# => ["richard", "hoppes", "nicholas", "shook"]
```

Alternate names: #collect, #collect_concat

Ruby 2.0: Enumerable::Lazy

- In 1.9, lots of enumerable functions return arrays => can't be lazy
- In 2.0:

```
a = [1,2,3,4,2,5].lazy.map { |x| x * 10 }.
  select { |x| x > 30 }  # => no evaluation

a.to_a # => [40, 50]
```

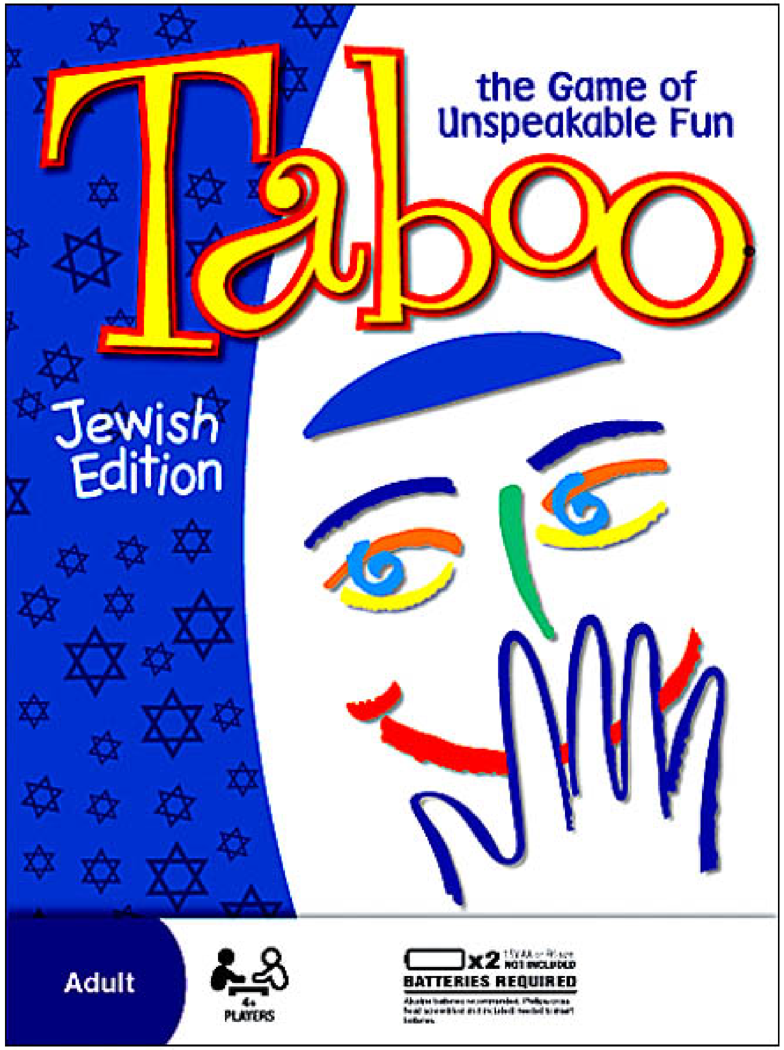
http://blog.railsware.com/2012/03/13/ruby-2-0-enumerablelazy/

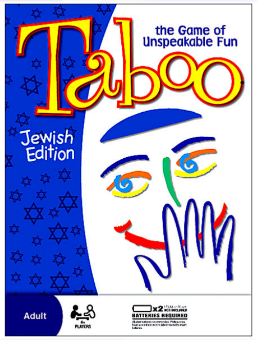
Fibbonacci enumerator

```
def fibbonacci(a=0, b=1)
  return enum_for(:fibbonacci,a,b) if !block_given?
  yield a
  yield b
  while true
    a, b = b, a + b
    yield b
  end
end

fibbonacci.first(10)
  # => [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

fibbonacci.each_cons(2) { |x, y| puts y.to_f/x }
  # => approaches Golden Ratio, 1.61803399
```

The End

Taboo game


Taboo
the Game of Unspeakable Fun
Jewish Edition
Adult
4+ PLAYERS
x2
BATTERIES REQUIRED


Taboo
the Game of Unspeakable Fun
Jewish Edition
Adult

Taboo: take turns

```
while true
  players.each do |player|
    # ...
  end
end


players.cycle do |player|
  # ...
end


players.cycle(3) do |player|
  # ...
end
```

~~Taboo: scoring turns~~

```
def tu        player, goal_word, t        ds)
```

~~-1 if   aid any of   o_words!~~

~~+1 if any o   goal_word~~

~~Otherwise, 0~~

```
end
```

~~all?, #none?, and #one? also available!~~

Taboo: winners

```ruby
teams = [team0, team1]

winning_team = teams.max_by &:score
losing_team = teams.min_by &:score


losing_team, winning_team = teams.minmax_by &:score


losing_team, winning_team = teams.sort_by &:score
```

Linked List Example

```ruby
class Node
  attr_accessor :value, :next_node

  def initialize(value)
    @value = value
  end
end
```

Linked List Example

```ruby
class LinkedList
  include Enumerable
  attr_accessor :next_node   # first node of list

  def each
    n = self
    while n = n.next_node
      yield n.value
    end
  end

  def initialize(values)
    values.inject(self) do |last_node, value|
      last_node.next_node = Node.new(value)
    end
  end
end
```

Linked List Example

```ruby
list = LinkedList.new(1..7)
p list.count         # => 7
p list.to_a          # => [1, 2, 3, 4, 5, 6, 7]
p list.entries       # same as #to_a
p list.inject(:+)    # => 28
```

Fibbonacci: each_with_index

```ruby
fibbonacci.each_with_index do |f, index|
  puts "#{index}: #{f}"
  break if f > 10
end
```

Output:

```
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
```

Fibonacci: each_cons

```
fibbonacci.each_cons(2) do |x, y|
  puts "%10f %2d %2d" % [y.to_f/x, x, y]
end
```

Output:

```
       Inf  1  0
  1.000000  1  1
  2.000000  2  1
  1.500000  3  2
  1.666667  5  3
  1.600000  8  5
  1.625000 13  8
  1.615385 21 13
  1.619048 34 21
  1.617647 55 34
...
```