Class Enumerator < Object

1.9 Relies on: each, <=>

Enumerator allows you to capture the concept of an enumeration as an object. This allows you to store enumerations in variables, pass them as parameters, and so on.

Some of the methods in the Enumerable module can return an Enumerator object.

```
enum = (1..10).take_while {|num| num % 5 != 0 }
enum.class   # =>   Array
enum.to_a    # =>   [1, 2, 3, 4]
```

You can also create enumerators with the method Kernel#to_enum (or via its alias, Kernel#enum_for). By default, these methods look for an each method in the object you're enumerating, but this can be overridden by passing them the name of a method (and possibly parameters to be used) that invokes a block for each item to be enumerated.

```
str = "quick brown fox"
case what_to_process        # set elsewhere to :by_word
when :by_bytes
  enum = str.to_enum(:each_byte)
when :by_word
  enum = str.to_enum(:scan, /\w+/)
end
enum.each {|item| p item}
```

produces:

```
"quick"
"brown"
"fox"
```

Mixes in

Enumerable:

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

Class methods

new

Enumerator.new {| *yielder* | *block* } → *enum*
Enumerator.new(*obj*, *method=:each*, ⟨ args ⟩*) → *enum*

The first form constructs an enumerator based on the block. The block is passed an object of class Enumerator::Yielder. You can use the << or yield methods of this yielder to supply values to be returned by the enumerator. This process is performed lazily (similar to the way that fibers can be used to generate sequences).

```
def multiples_of(n)
  Enumerator.new do |yielder|
```

```
    number = 0
    loop do
      yielder.yield number
      number += n
    end
  end
end
twos = multiples_of(2)
threes = multiples_of(3)
5.times do
  puts "#{twos.next} #{threes.next}"
end
```

produces:

```
0 0
2 3
4 6
6 9
8 12
```

The second form builds an enumerator based on the given method of *obj*. Any additional arguments are passed to this method. The method then yields successive values to a block—these values are picked up and used by the new enumerator. You should avoid this form in favor of the equivalent to_enum call.

```
enum_poor = Enumerator.new(1..10, :each_slice, 3)
enum_poor.to_a   # =>  [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10]]

enum_good = (1..10).enum_for(:each_slice, 3)
enum_good.to_a   # =>  [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10]]
```

Instance methods

each

enum.each { | *item, ...* | *block* } →

Calls the block for each item in the enumeration. This does not create an intermediate array. Instead, the original iterating method (the one used when creating the enumerator) is called passing it the block passed to this method. The block receives as many parameters as the original method passes.

```
def dump_enum(enum)
  enum.each { |item| p item }
end
enum = (1..10).enum_for(:each_slice, 3)
dump_enum(enum)
```

produces:

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
[10]
```

Note that because Enumerator defines each and includes Enumerable, all the enumerable methods are available too.

```
enum = "quick brown fox".enum_for(:scan, /\w+/)
enum.minmax   # =>   ["brown", "quick"]
```


each_with_index

enum.each_with_index {| *item, ..., index* | *block* } →

Same as each but appends an index argument when calling the block. Returns a new Enumerator if no block is given.

```
enum = (1..10).enum_for(:each_slice, 3)
enum.each_with_index do |subarray, index|
  puts "#{index}: #{subarray}"
end
```

produces:

```
0: [1, 2, 3]
1: [4, 5, 6]
2: [7, 8, 9]
3: [10]
```

each_with_object

enum.each_with_object(*memo*) {| *item, memo* | *block* } → *memo* or *enumerator*

1.9

Calls *block* for each item in *enum*, passing it the item and the parameter passed initially to each_with_object. Returns an Enumerator object if no block is given.

```
animals = %w(cat dog wombat).to_enum
hash = animals.each_with_object({}) do |item, memo|
  memo[item] = item.upcase.reverse
end
hash   # =>   {"cat"=>"TAC", "dog"=>"GOD", "wombat"=>"TABMOW"}
```

next

enum.next →

Returns the next item in the enumeration. Raises StopIteration if you call it past the last item. Internally this is implemented using fibers and so cannot be called across threads.

```
array = [ 1, 2, 3, 4 ]
e1 = array.to_enum
e2 = array.to_enum
e1.next   # =>   1
e1.next   # =>   2
e2.next   # =>   1
```

If the underlying method called by the enumerator has side effects (such as moving your position while reading a file), those side effects will be triggered. For this reason, next breaks the abstraction provided by Enumerator.

```
f = File.open("testfile")
enum1 = f.to_enum(:each_byte)
enum2 = f.to_enum
enum1.next   # =>  84
enum1.next   # =>  104
enum2.next   # =>  "is is line one\n"
f.gets       # =>  "This is line two\n"
enum2.next   # =>  "This is line three\n"
```


rewind

enum.rewind → *enum*

Resets the sequence of values to be returned by next.

```
array = [ 1, 2, 3, 4 ]
e1 = array.to_enum
e2 = array.to_enum
e1.next   # =>  1
e1.next   # =>  2
e2.next   # =>  1
e1.rewind
e1.next   # =>  1
e2.next   # =>  2
```

Has no effect if the underlying method of the enumerator has side effects and therefore cannot be rewound.

```
enum = File.open("testfile").to_enum
enum.next   # =>  "This is line one\n"
enum.next   # =>  "This is line two\n"
enum.rewind
enum.next   # =>  "This is line one\n"
```

with_index

enum.with_index {| *item, ..., index* | *block* } →

Synonym for each_with_index.

with_object

enum.with_object(*memo*) {| *item, memo* | *block* } → *memo* or *enumerator*

1.9

Synonym for each_with_object.