

Ruby advance topics

Recursive Functions

- “Those that keep calling themselves until they hit an end goal”
- Better performance than loops

```
1. def iterative_factorial(n)
2.   (1..n).inject(:*)
3. end
4.
5. def recursive_factorial(n)
6.   # Base case
7.   return 1 if n <= 1
8.
9.   # Recursive call
10.  n * recursive_factorial(n-1)
11. end
```

Recursive Functions

- Homework
 - Fibonacci Sequence with each number multiplied by 100

Enumerable

- Module in Ruby that allow to perform operations, from which classes like arrays and hashes inherit
- Useful methods
 - .each
 - .map – Same as array, but returns the array
 - .inject – Applies the block result element to each item in the array
 - <https://www.delftstack.com/howto/ruby/use-ruby-inject-method/>
 - .any?

Enumerable - .chunk

- Enumerates over the items based on the condition and return an Enumerable. Most often, this Enumerable is used with an each

```
irb(main):006:0> w = [1, 2, 3, 4, 5].chunk { |x| x if x > 2 }  
=> #<Enumerator: #<Enumerator::Generator:0x00007fdc43193a30>:each>  
irb(main):007:0> w.each { |x| puts x }  
3  
3  
4  
4  
5  
5  
=> nil  
irb(main):008:0> 
```

What does chunk receive as a parameter ? -> a block

Returns an enumerator:

```
-> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]  
2.7.5:007 > array.chunk { |x| x if x.even? }  
=> #<Enumerator: #<Enumerator::Generator:0x00007fccd6931cc8>:each>  
2.7.5:008 > 
```

Let's try to access that enumerator

This enumerator does not have special methods

```
=> #<Enumerator: #<Enumerator::Generator:0x00007fccd9842170>:each>  
2.7.5:011 > chunk_result.first  
Traceback (most recent call last):  
 9: from /Users/oskarhinojosa/.rvm/rubies/ruby-2.7.5/bin/irb:23:in `<main>'  
 8: from /Users/oskarhinojosa/.rvm/rubies/ruby-2.7.5/bin/irb:23:in `load'  
 7: from /Users/oskarhinojosa/.rvm/rubies/ruby-2.7.5/lib/ruby/gems/2.7.0/gems/irb-1.2.6/exe/irb:11:in `<top (required)>'  
 6: from (irb):11  
 5: from (irb):11:in `first'  
 4: from (irb):11:in `each'  
 3: from (irb):11:in `each'  
 2: from (irb):11:in `each'  
 1: from (irb):10:in `block in irb_binding'  
NameError (undefined local variable or method ` ' for main:Object)  
2.7.5:012 > 
```

enumerator and enumerable

string of data into smaller lines, for example

each uses more resources, chunk no, because it is part by part?

Enumerable - .each

- Iterates and does the process over each item of the block

```
irb(main):006:0> w = [1, 2, 3, 4, 5].chunk { |x| x if x > 2 }  
=> #<Enumerator: #<Enumerator::Generator:0x00007fdc43193a30>:each>  
irb(main):007:0> w.each { |x| puts x }  
3  
3  
4  
4  
5  
5  
=> nil  
irb(main):008:0> █
```

MAP

returns an array but with modifications

```
2.7.5 :008> w.map { |x| x if x.even? }  
=> [nil, 2, nil, 4, nil, 6, nil, 8, nil, 10, nil, 12, nil, 14, nil, 16, nil, 18, nil, 20, nil, 22, nil, 24, nil, 26, nil, 28, nil, 30, nil, 32, nil, 34, nil, 36, nil, 38, nil, 40, nil, 42, nil, 44,  
nil, 46, nil, 48, nil, 50, nil, 52, nil, 54, nil, 56, nil, 58, nil, 60, nil, 62, nil, 64, nil, 66, nil, 68, nil, 70, nil, 72, nil, 74, nil, 76, nil, 78, nil, 80, nil, 82, nil, 84, nil, 86, nil, 88,  
nil, 90, nil, 92, nil, 94, nil, 96, nil, 98, nil, 100]  
2.7.5 :009> █
```

It returns all values.

Map returns an array, chunk an iterator.

Enumerable - .inject

⊕

- Applies the result from the block to each item in the array into a single result

```
irb(main):024:0>  
irb(main):025:0> w.inject(:*)  
> 120  
irb(main):026:0> w.inject(:+)  
> 15  
irb(main):027:0> w  
> [1, 2, 3, 4, 5]  
irb(main):028:0> █
```

- Need to use the accumulator? No problem.

```
inject (initial_value) { |accumulator, array_item| expression }
```

`:+` is a shortcut, instead of doing the whole block. He is adding the result of each value to the next one

You can also use the other sintaxis with an accumulator variable.

Enumerable - .any?

- Checks if any of the elements meets the given condition.

```
irb(main):036:0>
irb(main):037:0> w.any?(1)
=> true
irb(main):038:0> w.any? { |x| x > 1 }
=> true
irb(main):039:0> █
```

Can be use to check if item exists on array

Enumerable - .all?

- All values in the enumerable match the condition or block in specific

```
collection.all?(&:valid?)
```

All the same as any, but the condition will be applied to each one.

```
2: from /Users/oskarhinojosa/.rvm/rubies/ruby-2.7.5/lib
1: from (irb):15
NoMethodError (undefined method `all' for [1, 2, 3]:Array)
Did you mean?  all?
[2.7.5 :016 > [1,2,3].all?(:even?)
=> false
[2.7.5 :017 > [4,2,6].all?(:even?)
=> false
2.7.5 :018 > █
```



Blocks

- Single line of snippets of code
 - Useful information: For betterment they also do multi-line blocks with brackets and use the keyword compact at the end of the block

```
array_of_things.each do |thing|  
  thing if thing.condition?  
end.compact
```

```
[1, 2, 3].map { |n| n * 2 }  
# => [2, 4, 6]
```

They declare an anonymous piece of code.

.compact not required but it is a good practice.

```
3  
4  
5 (1..100).to_a.chunk { |x|  
6  
7 }.compact
```



Remember that ruby is a just in time compilation. .compact is a way to tell the compiler that the block has finished.

Declare your own block



Blocks – Declaring your own blocks

- Options: Use the yield keyword in the method that will be your block
 - Notice how the variable is passed &block

```
1. def explicit_block(&block)  
2.   block.call # same as yield  
3. end  
4.  
5. explicit_block { puts "Explicit block  
   called" }
```

then you can use yield or call.



Blocks – Declaring your own blocks

- When creating methods, the method `block_given?` is available to know if you can use the yield argument

```
1. def do_something_with_block
2.   return "No block given" unless
   block_given?
3.   yield
4. end
```

also we have this method, to check if a block was given.

Yield

Blocks – Declaring your own blocks

- You can use parameters with the yield keyword

```
1. def one_two_three
2.   yield 1
3.   yield 2
4.   yield 3
5. end
6.
7. one_two_three { |number| puts number *
  10 }
8. # 10, 20, 30
```

Con yield si queremos pasar más de un parámetro, usemos un array

Off class notes

<https://scoutapm.com/blog/ruby-enumerator>

<https://blog.carbonfive.com/enumerator-rubys-versatile-iterator/#:~:text=An%20external%20iterator%20is%20controlled,collection%20classes%2C%20Array%20and%20Hash%20>.

Enumeration

Enumeration -> process to transverse over a set of elements

An entity is known to be enumerable when it has a set of elements and knows how to traverse over them.

External vs internal iterator

External Iterator

When you get an iterator and step over it, that is an external iterator

```
for (Iterator iter = var.iterator(); iter.hasNext(); ) {  
    Object obj = iter.next();  
    // Operate on obj  
}
```

Internal Iterator

When you pass a function object to a method to run over a list, that is an internal iterator

```
var.each( new Functor() {  
    public void operate(Object arg) {  
        arg *= 2;  
    }  
});
```

Enumerable

Is a module

From this module comes out methods of search, traversing, etc, that other classes like array, hash or range uses.

Enumerator

Is a class.

`Enumerator` is an `Enumerable` plus external iteration. In this post, we'll take a look at the basics of `Enumerator`s and some of the powerful functionality that they make possible.

`Enumerator` itself is `Enumerable`, so we can use the full power of `Enumerable`.

It add external iterators

```
.  enum = Enumerator.new do |yielder|
!    yielder << "a"
}    yielder << "b"
!    yielder << "c"
}  end
}

!  enum.next # "a"
}  enum.next # "b"
}  enum.next # "c"
}  enum.next # StopIteration: iteration reached an end
```

Create enumerator from enumerable

Creating an Enumerator from an Enumerable

The more common way to create an `Enumerator` is from an `Enumerable` object, specifically an object that defines an `#each` method. `Object#to_enum` (aliased to `Object#enum_for`) is implemented to return a new `Enumerator` that will enumerate by sending `#each` to its receiver.

```
1 array = [1, 2, 3]
2 enum = array.to_enum
3 enum.each {|n| n * 2} # 2 4 6
```

2.rb hosted with ❤ by GitHub

[view raw](#)

Turning Non-Enumerables into Enumerables

Turning Non-Enumerables into Enumerables

So far, we've looked at how to construct `Enumerator` s from scratch and from `Enumerable` s. However, several non-`Enumerable` classes also return `Enumerator` s. Since `Enumerator` is `Enumerable` , this allows you to effectively turn a non-`Enumerable` class into an `Enumerable` one.

`Integer` , for example, returns an `Enumerator` from `#times` , `#upto` , and `#downto` .

```
1  enum = 5.times
2  enum.map { [] }      # [[], [], [], [], []]
3
4  enum = 1.upto 10
5  enum.inject(:+)      # 55
6
7  enum = 5.downto 0
8  enum.map {|n| n * 2} # [10, 8, 6, 4, 2, 0]
```

5.rb hosted with ❤ by GitHub

[view raw](#)

`String` has several iteration methods that will return an `Enumerator` when not given a block.

```
1  enum = "hello".each_char
2  enum.map &:upcase      # ["H", "E", "L", "L", "O"]
3
4  enum = "abc".each_byte
5  enum.map {|n| n}      # [97, 98, 99]
6
7  enum = "one\ntwo\nthree".each_line
8  enum.map {|line| line.chomp.reverse} # ["eno", "owt", "eerht"]
```