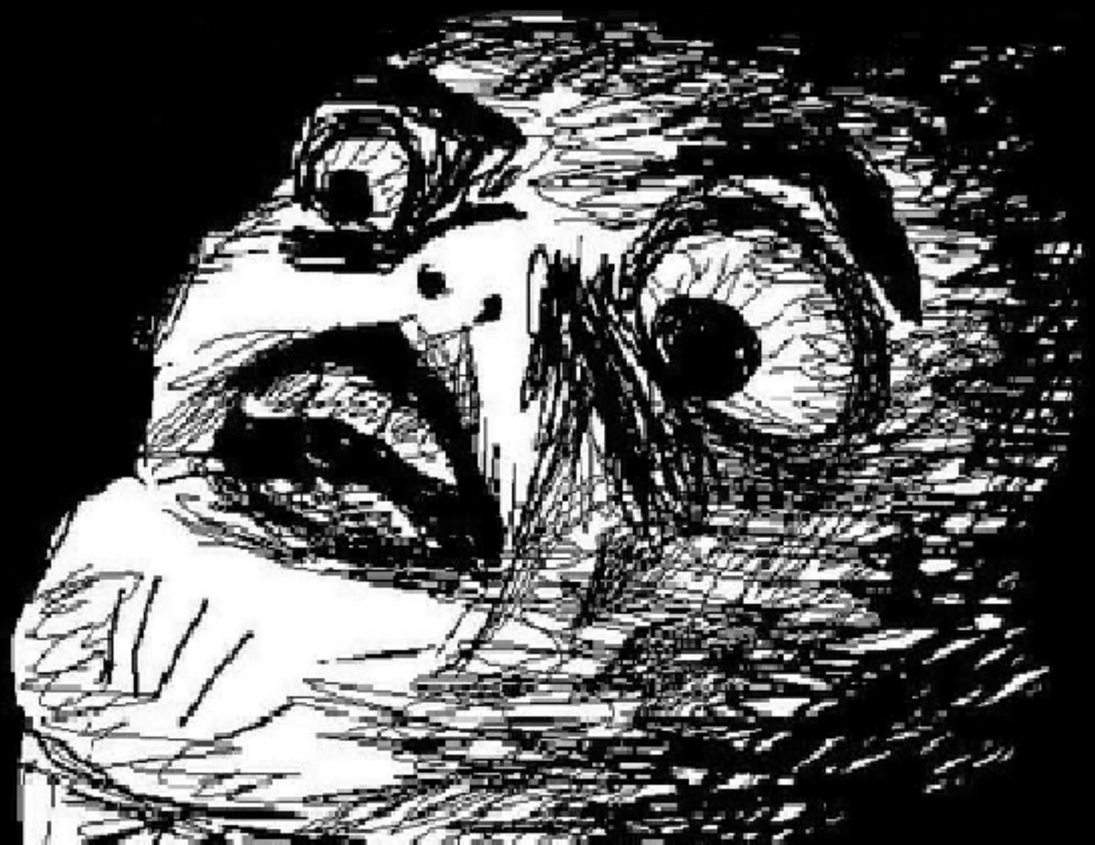
Blocks, Procs and Lambdas

Yes kids, today we are talking about Blocks, Procs and Lambdas...



Don't be scared!

Actually, you have seen them A LOT

Do you remember this?:

```
array_of_things.each do |thing|
  thing.cool_method
end
```

The code between the do and the end is a block.

Isn't it cool to know how to name things?

That's power, man.

Blocks are just chunks of code that you can drop into another method as a parameter.

You declare a block using squiggly braces `{}` if it's on one line or `do ... end` if it's on multiple lines:

```
[1,2,3].each { |num| print "#{num}! " }
  # 1! 2! 3! =>[1,2,3]

[1,2,3].each do |num|
  print "#{num}!"
end
#1! 2! 3! =>[1,2,3]
```

They both do the same.

And what about the variable that we have between the pipes?

This variable is the argument of the block. Just like methods, blocks could have one or several arguments, but they are not mandatory.

Then, when should we use them?

Dumb example:

```
[1, 2, 3].each { puts 'Hello world'}
[1, 2, 3].each { |number| puts number}
```

Blocks are used as arguments for other methods (like `each`), just like the normal arguments that you've seen between the parentheses.

But they always go at the end of the definition because they are too long.

They are nothing special

Neither the `each` method is special.

Matz built it to accept a block as an argument

Let's break the magic

Introducing the *yield* statement

When you write your own methods, you don't need to specify if your method should be able to accept a block. It will just be there waiting for you when you call `yield` inside your method.

The `yield` statement says "run the block right here"

Let's see how it works opening the guts of a customized `each`:

```
class Array
  def my_each
    i = 0
    while i < self.size
        yield(self[i])
        i+=1
    end
    self
  end
end
```

There are 3 important things here:

1- It's built inside the class `Array`, so we can call the customized method on a real array like this:

```
[1, 2, 3].my_each {|num| print num}# => 1 2 3
```

2- Inside the method, `self` refers to the same array on which our customized `each` is being applied.

```
class Array
  def my_each
    i = 0
    while i < self.size
        yield(self[i])
        i+=1
    end
    self
  end
end
```

Quick question: Do you remember what the official `each` method returns?

If you don't remember, take a look at the last `self`. There you've the answer!

3- Finally, with the `while` loop we are iterating through `self`, using the `yield` statement to call the block in the place we want.

So, when `yield` makes its magic, our method will look like this:

```
class Array
  def my_each
    i = 0
    while i < self.size
        print "#{self[i]}!"    # Our block got "subbed in" here
        i+=1
    end
    self
  end
end
```

Remember: `yield` says "*run the block right here*"

One final consideration. Take a look at this chunk of code of `my_each`:

```
yield(self[i])
```

That's the way we give the needed arguments to the method. Just as if the block were a normal method.

Exercise 1

Create a `my_each_with_index` method in the same way as we have done with `my_each`.

Solution 1

```
class Array
  def my_each_with_index
    i = 0
    while i < self.size
        yield(self[i])
        i+=1
    end
    self
  end
end
```

Exercise 2

Create `#my_select` in the same way, though you may use `#my_each` in your definition (but not #each).

Solution 2

```
class Array
  def my_select
      return self unless block_given?
      new_array = []
      my_each { |i| new_array << i if yield(i) }
      new_array
    end
end
```

Exercise 3

Create:

- `#my_all?`
- `#my_any?`
- `#my_none?`
- `#my_count`
- `#my_map`
- `#my_inject`

Exercise 4

Test your `#my_inject` by creating a method called `#multiply_els` which multiplies all the elements of the array together by using `#my_inject`.

Example:

```
multiply_els([2,4,5]) #=> 40
```

When should we use blocks?

When creating methods where you want to optionally be able to override how they "work" internally by supplying your own block

The `sort` method is a clear example of that. You can pass it a block to decide how it's going to sort the elements.

For that purpose, if you want to ask whether a block was passed to a method (to only yield in that case), use `block_given?`.

```
def my_sort
  ...  #here goes the loop code like the each
  if block_given?
    yield
  else
    ... #here goes the usual sort code
  end
end
```

Re-using blocks

But... What if we want to use the same block multiple times?

Remember our DRY mantra!

Introducing the Procs

That's a job for Procs, aka Procedures!

A Proc is just a block that you save into a variable

```
my_proc = Proc.new { |arg1| print "#{arg1}! " }
```

Introducing the Procs

You can use that block of code (now called a Proc) as an input to a function by prepending it with an apersand & symbol

```
[1,2,3].each(&my_proc) #=> 1! 2! 3!
```

It's the same procedure than passing the block like you did before!

When you create your own function to accept procs, the guts need to change a little bit because you'll need to use `#call` instead of `yield` inside.

```
def my_greeting(generic_proc)
  greeting = "Hello. We are using a #{generic_proc.class}"
  generic_proc.call()
end

my_greeting(&my_proc) #=> Hello. We are using a Proc!
```

Most of the time, using a block is more than enough, specially in your early projects.

When you begin to feel the need for using a Proc, you'll have Procs there waiting for you.

Lambdas

In Ruby, we have another fellow that is really similar to a Proc.

It's called Lambda

Lambdas work almost like Procs.

```
my_lambda = lambda {puts "Hello lambda!"}

my_lambda.call
```

But they are much stricter than Procs about you passing them the correct number of arguments (you'll get an error if you pass the wrong number).

```ruby
my_proc = Proc.new { |param| puts "Hi from Proc with #{param}!!!"}
my_proc.call

# => Hi from Proc with !!!


my_lambda = lambda { |param| puts "Hello from lambda with #{param}!"}
my_lambda.call

# => wrong number of arguments (0 for 1) (ArgumentError)
```

Exercise

Modify the `#my_map` method you coded before to receive a proc instead of a block.

Exercise

Fizz Buzz Pro!!!!!

Go to the Prework's Fizz Buzz and refactor it trying to use Lambdas or Procs for the conditions :)