

Lambdas

Lambdas

- A way to define a Proc and its parameters with a special syntax

```
1. say_something = -> { puts "This is a  
lambda" }
```

- Defining it with variables

```
irb(main):051:0> place_something = -> (x, y, z) { puts x, y, z }  
=> #<Proc:0x00007fdc408e8928@irb:51 (lambda)>  
irb(main):052:0> 
```

Procedure and lambda

Lambda is a procedure, a procedure with special syntax and context.

With lambdas you use the context of the own lambda.

Procs

- Different than lambdas when there are arguments.

```
irb(main):087:0>  
irb(main):088:0> t = Proc.new { |x, y| puts x, y }  
=> #<Proc:0x00007fdc43173398@irb:88>  
irb(main):089:0> t.call(1, 2)  
1  
2  
=> nil  
irb(main):090:0> 
```

- If you use procs, you use the context of the method where you are calling it.
- If you use procs, you create your own context

Procs

- In the following example, we return in the call_proc method, not the proc itself.

```
1. def call_proc
2.   puts "Before proc"
3.   my_proc = Proc.new { return 2 }
4.   my_proc.call
5.   puts "After proc"
6. end
7.
8. p call_proc
9. # Prints "Before proc" but not "After
   proc"
```

If you are using a procedure inside a function and you call a return, this return is going to end the function.

You can still return something if you simply put the variable or value without return as the last line of the procedure.

procedures -> outside context

lambda -> inner context.

Ruby Introspection & GC Agenda

- Introspection
 - What is introspection
 - The Object Space
 - Looking inside objects
 - Calling methods dynamically
 - A Note about performance
- The Garbage Collector
 - Introduction
 - Mark and Sweep Algorithm
 - Heap Compaction
 - Performance Issues and managing our own memory

What can a program see within itself?

- What objects contain
- Current class hierarchy
- Contents and behaviors of objects
- Information on methods

Introspection

2

- With the information the program knows about themselves, they can call methods at runtime and modify themselves while running

The Object Space

2

- Where all objects inhabit in Ruby
- Objects alive and not yet collected by the Garbage Collector
 - This means an object is alive as long as it has any references pointing to it

In the object space is the list of all objects that are alive.

The Object Space

- What does it contain by default?
- Useful Tip: You can also use the method `.each_object` without an argument (class)

```
irb(main):001:0> ObjectSpace
=> ObjectSpace
irb(main):002:0> ObjectSpace.each_object(Numeric) { |x| p x }
(0+1i)
9223372036854775807
-166676550149418238933563278187521080161
NaN
Infinity
1.7976931348623157e+308
2.2250738585072014e-308
=> 7
```

The object space is a module/class.

To access the object, we use the `each_object` method.

The Object Space

- Please note that the `ObjectSpace` doesn't know about `Fixnum`, `true`, `false`, or `nil`.

```
irb(main):099:0> ObjectSpace.count_objects[:TOTAL]
=> 35054
irb(main):100:0> b = nil
=> nil
irb(main):101:0> ObjectSpace.count_objects[:TOTAL]
=> 35054
irb(main):102:0> █
```

Looking Inside Objects

- You can get the methods available for any variable with `.methods`

```
1  r = 1..10 #=> 1..10
2  list = r.methods #=> begin, end, bsearch.....
3  list.length #=> 125
4  list[0..3] #=> [begin end bsearch to_a]
5
```

To know if a class has a method:

```
?, :respond_to?, :object_id, :send, :__send__, :!, :!=
2.7.5 :008 > a.respond_to? :delete_prefix
=> true
2.7.5 :009 > 
```

Calling methods dynamically

- Determining the objects direct class and class hierarchy

```
class Example
  MYCONSTANT = 'hello this is a test'

  def self.this_is_a_class_method
    puts 'ok'
  end

  def this_is_a_public_method
    puts 'ok'
  end

  protected

  def this_is_a_protected_method
    puts 'ok'
  end

  private

  def this_is_a_private_method
    puts 'ok'
  end
end
```

```
Example.private_instance_methods
#=> [:this_is_a_private_method, :irb_binding, ...]
Example.protected_instance_methods
#=> [:this_is_a_protected_method]
Example.public_instance_methods
#=> [:this_is_a_public_method, :instance_variable_defined?, ...]
Example.singleton_methods #=> [:this_is_a_class_method]
Example.constants #=> [:MYCONSTANT]
```

(singleton methods are the class methods).

Calling the methods dynamically

- Send
- Eval
- Method.call

Send

- Sending a symbol (as best practice, although it also accepts strings)
- Remember: Symbols represent methods (identifiers)

```
"Ruby on Rails!".send(:length) #=> 14
"Ruby on Rails!".send("gsub", /Rails/, "wheels") #=> "Ruby on wheels!"
"Ruby on Rails!".send(:gsub, /Rails/, "wheels") #=> "Ruby on wheels!"
```

Send is calling the method itself, but it receives a symbol

```
=> true
2.7.5 :009 > a.send(:reverse)
=> "raksO"
2.7.5 :010 > █
```

The Method Object

- Method class that creates a method that is not bound to a class
- This means it can be used anywhere

```
def example_function(a)
  2 * a
end

method(:example_function).call(2) #=> 4
```

Llamar métodos que no están dentro de una clase.

Passing a method with the method name as a symbol, allows you to pass it as a variable.

```
2.7.5 :001 >
2.7.5 :002 >
2.7.5 :003 >
2.7.5 :004 >
2.7.5 :005 > def example_function
2.7.5 :006 >   puts "ok"
2.7.5 :007 > end
=> :example_function
2.7.5 :008 > method(:example_function).call
ok
=> nil
2.7.5 :009 > method(:example_function)
=> #<Method: main.example_function() (irb):1>
2.7.5 :010 > v = method(:example_function)
=> #<Method: main.example_function() (irb):1>
2.7.5 :011 > class Example
2.7.5 :012 >   def self.example(arg)
2.7.5 :013 >     arg.call
2.7.5 :014 >   end
2.7.5 :015 > end
=> :example
2.7.5 :016 > Example.example(v)
ok
=> nil
2.7.5 :017 > █
```

It is similar to passing it as a procedure.

Eval

- Takes a string and execute it
- Explicitly evaluates it into the current context

```
a = '2.send(:class)'
eval a
```

Receives a string and evaluates it with the current context.

THIS IS CALLED METAPROGRAMMING

A few benchmarks

- Click

```
require 'benchmark'
include Benchmark

test = "My not so long string"
method_var = test.method(:length)

n = 10000000

Benchmark.bm(10) { |x|
  x.report("call") { n.times { method_var.call } }
  x.report("send") { n.times { test.send(:length) } }
  x.report("eval") { n.times { eval("test.length") } }
  x.report("norm") { n.times { test.length } }
}.compact

=begin
| | | | | | | user      system      total      real
call      0.648847  0.002007  0.650854 ( 0.653695)
send      0.636859  0.000771  0.637630 ( 0.638562)
eval     53.932909  0.083692 54.016601 ( 54.091896)
norm      0.334988  0.000221  0.335209 ( 0.335498)
=end
```

The performance of the other methods is slower.

The benchmark is very used

Want to know about Benchmarks?



- Benchmark is a great Ruby Module for managing and knowing about performance
- Visit <https://ruby-doc.org/stdlib-3.0.0/libdoc/benchmark/rdoc/Benchmark.html>

System Hooks

- Technique that lets you trap events
- Most common event is time of creation
- Technique allowed by the *alias* method
 - Creates new aliases of the methods

System Hooks: An example

- Click to

```
class Object
  def timestamp
    return @timestamp
  end
  def timestamp=(aTime)
    @timestamp = aTime
  end
end

class Class
  alias_method :old_new, :new
  def new(*args)
    result = old_new(*args)
    result.timestamp = Time.now
    return result
  end
end

class Test
end

Test.new.timestamp #=> Dependable of execution time but mine was 2022-08-12 14:25:13 -0500
```

How to edit a class?

With `alias_method` it creates almost a shadow copy

System hook, a way to add additional properties to the variables we create.

How do we add a timestamp to a class?

.....

Runtime Callbacks

- Being notified when one of the following happens

Event	Callback Method
Adding an instance method	<u>Module#method_added</u>
Adding a singleton method	<u>Kernel::singleton_method_added</u>
Subclassing a class	<u>Class#inherited</u>
Mixing in a module	<u>Module#extend_object</u>

Runtime Callbacks

- Singleton methods – When a class has methods
- Extend object when extend is used inside a specific object (variable)
- Inherited – When a class inherits from another class
- Method added – When we modify methods of a module at runtime

Event	Callback Method
Adding an instance method	<u>Module#method_added</u>
Adding a singleton method	<u>Kernel::singleton_method_added</u>
Subclassing a class	<u>Class#inherited</u>
Mixing in a module	<u>Module#extend_object</u>

A way to adding properties to an object. USED THE MOST FOR THAT

```
module Test
  def self.extend_object(o)
    puts "Added Test to #{ o.class } type"
  end
end

(3).extend(Test) #=> Added Test to Integer type
```

Additional Word of Knowledge: Mixins

- Multiple inheritance in a class through Modules

```
module A
  def a
  end

  def b
  end
end

module B
  def d
  end

  def e
  end
end

class Test
  include A
  include B
end

my_test = Test.new
my_test.a
my_test.b
my_test.d
my_test.e
```

We cannot do multiple inheritance.

A module can have a class inside of it, but it can only be used inside of the module.

Ruby Garbage Collector

- Works through malloc and free, later on calloc and realloc were also used

Removes things that are not useful from the memory automatically.

The Mark Phase

- Ruby Crawls through the objects and marks the ones still in use
- Colors
 - White: Unmarked – Still in use
 - Grey: Marked but may point to white object. Live but not scanned
 - Black: Marked and don't point to any white objects

The Sweep Phase

- Unmarked objects are returned to the free list
 - Separate spaces by young and old objects
 - Incremental garbage collection since Ruby 2.2
 - Now, it works in blocks
 - Spend same time in GC but in shorter and more frequent bursts
 - Protected and unprotected objects concept (from write)

End of garbage collector phases...

Heap Compaction since Ruby 2.7

- Thanks to Mark Evans using Tmalloc
 - Tmalloc – Fast memory allocation implementation of C's malloc by Google

Memory Loops: Options

- Debugging: BEST PRACTICE. ALWAYS
- Using Jemalloc – Always the second option

Jemalloc

useful gem to debug memory loops, and memory usage?

MEMORY LOOP EQUALS MEMORY LEAK


```

1
2  class MyClass
3
4      def public_method
5          my_method
6      end
7
8      protected
9
10     def my_method
11         puts "im protected"
12     end
13 end
14
15
16 module MyModule
17     class Hello
18         def initialize
19             puts "new class"
20         end
21     end
22
23     def method
24         Hello.new
25     end
26 end
27
28 def ok
29     a = 2
30     b = Proc.new { a *= 2 }
31     b.call
32 end
33
34 def ok1
35     c = -> { return 3 }
36     c.call
37 end
38
39 MyClass.public_instance_methods[0]

```

Manera correcta

Para llamar un método de un módulo, solo se puede llamar dentro de un módulo. Estas están protegidas dentro del scope del módulo.