RUBY (2/2)

- Objects
- Modules
- Inheritance / Mixin
- Error handling
- Block / Proc / Lambda
- Regex
- Gem
- Metaprogramming

OBJECTS

In Ruby, everything is an object (`5.times { print "We *love* Ruby -- it's outrageous!" }`)

```ruby
class Counter
  attr_accessor :counter
  attr_accessor :initial_value

  def initialize(initial_value: 0)
    self.initial_value = initial_value
    self.counter = initial_value
  end

  def increment!
    self.counter += 1
  end

  def decrement!
    self.counter -= 1
  end

  def reset!
    self.counter = initial_value
  end

  def count
    self.counter
  end
end
```

METHODS

- Instance / Class
- Public / Private
- Chainable
- Args
- Return value

```
['hello', 'world'].join(', ').upcase
=> "HELLO, WORLD"
```

MONKEY PATCHING

```ruby
class String
  def anagram
    split('').shuffle.join('')
  end
end

puts "neo".anagram
# => one
```

MODULES

Modules are about organizing your code

```ruby
module API
  class User
    attr_accessor :name

    def initialize(name:)
      self.name = name
    end

    def self.random # class method
      new(name: ('a'..'z').to_a.shuffle[0,8].join)
    end

    def send(message, to:)
      Message.new(from: self, to: to, message: message).send
    end
  end

  class Message
    attr_accessor :from_user
    attr_accessor :to_user
    attr_accessor :message

    def initialize(from:, to:, message:)
      self.from_user = from
      self.to_user = to
      self.message = message
    end

    def send
      puts "sending `#{self.message}` from #{from_user.name} to #{to_user.name}"
    end
  end
end
```

```ruby
teacher = API::User.new(name: 'Hugues')
classroom = API::User.new(name: 'Classroom')
teacher.send("hello", to: classroom) # indirectly use the Message class
# => sending `hello` from Hugues to Classroom
teacher.send("hello", to: API::User.random) # indirectly use the Message class
# => sending `hello` from Hugues to ...
```

INHERITANCE

```ruby
module API
  class SuperUser < Message
    def initialize
      super('Super User')
    end
  end
end
```

```ruby
super = API::SuperUser.new
classroom = API::User.new(name: 'Classroom')
teacher.send("hello", to: classroom)
# => sending `hello` from Super User to Classroom
```

MIXINS

```ruby
module API
  module MessageSender
      def send(message, to:)
        Message.new(from: self, to: to, message: message).send
      end
  end
end

module API
  class User
    include MessageSender

    attr_accessor :name

    def initialize(name:)
      self.name = name
    end
  end
end
```

ERROR HANDLING

```ruby
class SlackAPI
  def self.get_conversations(channel_id:)
    uri = URI('https://slack.com/api/conversations.list?limit=50')
    req = Net::HTTP::Get.new(uri)
    req['Authorization'] = "Bearer xoxp-2486113197334-2492860403907-2492926538098-76ac2d6b0dcc5d6a24b3c72889355468"
    res = Net::HTTP.start(uri.hostname, uri.port, use_ssl: true) { |http| http.request(req) }
    response = res.body
    raise StandardError('Error') unless response
    JSON.parse(response)['channels']
  end
end

begin
  channels = SlackAPI.get_conversations(channel_id: 'abc')
rescue
  puts "something went wrong"
end
```

BLOCK / PROC / LAMBDA

Different flavors of anonymous functions.

BLOCK

Ruby blocks are anonymous functions that can be passed into methods

```ruby
class Array
  def my_map(&block)
    new_values = []
    for element in self
      new_values << block.call(element)
    end
    new_values
  end
end
```

```ruby
['hello', 'world'].my_map { |a| a.capitalize }
=> ["Hello", "World"]
```

LAMBDA

lambda are re-usable (named) blocks

```ruby
cipher = lambda { |a| a.split('').shuffle.join('') } # !! return
cipher.call('hello') # => lhloe
cipher.call # ArgumentError (wrong number of arguments (given 0, expected 1))
['hello', 'world'].map(&cipher) # => ["olehl", "lword"]
```

- **return from a lambda return from the lambda**

PROC

proc are re-usable (named) blocks

```
cipher = Proc.new { |a| a.split('').shuffle.join('') } # !! implicit return
cipher.call('hello')
cipher.call # NoMethodError (undefined method `split' for nil:NilClass)
['hello', 'world'].map(&cipher) # => ["olehl", "lword"]
```

- return from a proc return from the current scope

REGEX

A regular expression is a sequence of characters that specifies a search pattern. Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation – https://en.wikipedia.org/wiki/Regular_expression

```
"Do you like cats?" =~ /like/ # => true
"Do you like cats?".match(/like/) # => true
"The year was 1979.".scan(/\d+/) # => "1979"
!!"hugues@xdev.fr".match(/\A[\w.+-]+@\w+\.\w+\z/) # => true
```

GEM / PACKAGE MANAGER

A gem is a library which installed as a package

```
gem install httparty
irb
> require 'httparty'
> result = HTTParty.get('https://slack.com/api/conversations.list?limit=50', headers: { Authorization: "Bearer xoxp-2486113197334-2492860403907-2492926538098-76ac2d6b0dcc5d6a24b3c72889355468"} )
> puts result.parsed_response
```

METAPROGRAMMING & DSL

Writing ruby in Ruby

```ruby
class Message
  TYPES = %w(text video image audio) # => ['text', 'video', 'image', 'audio']
  attr_accessor :type

  TYPES.each do |_type|
    define_method("#{_type}?") do
      _type == type
    end
  end

  def initialize(type:)
    self.type = type
  end
end

Message.new(type: 'text').text? #=> true
Message.new(type: 'video').text? #=> false
Message.new(type: 'video').video? #=> true
```

- `method_missing`, `define_method`, 'delegate', ...

Learn more -> https://www.toptal.com/ruby/ruby-metaprogramming-cooler-than-it-sounds

DSL

A Domain-Specific Language, or DSL, is “a programming language of limited expressiveness focused on a particular domain”
— https://thoughtbot.com/blog/writing-a-domain-specific-language-in-ruby