Learn to Code with Ruby

17) Regular Expressions

17.1) start_with? & end_with? ✓ * case specific

Regex: using symbols to tell Ruby how to extract certain substrings from a larger string based on some criteria.

└ eg. puts phrase.downcase.start_with?("the"). → true/false.

```
def custom_start_with?(string, substring)
  string[0, substring.length] == substring ? true : false
                ↳ take 3 char          ↳ return true/false ✓
end
```

```
def custom_end_with?(string, substring)
  string[-substring.length..-1] == substring
end
```

17.2) .include? method.

```
def custom_include?(string, substring)
  len = substring.length
  string.chars.each_with_index do |char, index|
    string[index, len] == substring
  end
end
```

17.3) Regex Intro

- Another Ruby object. Class = Regexp.
- used to match patterns in strings
- created with //, & condition is placed between

- Syntax:
  - =~ return the index position of the first match
    eg. phrase =~ /K/ → index of first 'K' in phrase.
    puts =~ /x/ → nil

17.4) .scan method
- argument: RegExp.; used on string, return an array of all matches

eg. voicemail = "I can be reached at 555-123-4567 or regexman@gmail.com."

p voicemail.scan(/e/) → ["e", "e", "e", "e", "e"]
- .length → 5
- (/[re]/) → ["r", "e", "r", ...] # 'r' or 'e'.
  (/["er"]/) ↗

17.5) .scan method (cont)

p.voicemail.scan(/\d/) → ["5", "5" ~ "7"]
- any single digit (\d)

(/\d+/) → ["555", "123", "4567"]
- keeps going until ~~got a~~ run of digit
- '+': one or more occurrences of whatever is preceding it.

- scan can accept block.
  - voicemail.scan(/\d+/){|digit_match| puts digit_match.length}
    - 3
      3
      4

17.6) The wildcard symbol (.)
- indicates any characters; Any Single Character.
- eg. puts phrase =~ /./ → t
  - phrase.scan(/./) → {
- used when we don't know what character to expect in a specific position but we know there's a character.

└ eg. puts phrase.scan(/.am/) → xam
                                 ram
                                 am
  ↳ whatever character follow by am.
  └ (/.ing/) → ming
               zing
               ring.
  └ (/a.e/) → age
              awe

17.7) Backslash character
eg. puts paragraph.scan(/\./) → find '.'
  └ \d → digits.
  └ \D → non-digits
  └ \s → any whitespaces * + .length
  └ \s+ → any >1 whitespaces * " " ⇒ ""
  └ \S → anything except whitespaces.

17.8) RegEx Anchors - a specific symbol that ties a match to a specific point within a string. (start/end)

eg. poem.scan(/\d/)
  ↳ /\A\d+/ → search at the start of the string for >1 in order ⇒ find ...?
  ↳ /\eer\z/ → anchor to the end of the string & only search for whatever you put a the end of the string.

17.9) Exclude Characters (^)
eg puts sales.scan(/[^aeiou]/) → aeiou...
  ↳ /[^aeiouAEIOU]/ → excludes vowels.
  ↳ └ \s → excludes " + whitespaces
  ↳ └ \S → excludes " + s
  ↳ └ \d → excludes " + digits
  ↳ └ \. → excludes " + '.'
  ↳ + size → 24.

17.10) .sub & .gsub methods with RegEx

~~sub("a", "b")~~

eg. "apple".sub("p", "o") ⇝ aople # first only

"apple".gsub("p", "o") ⇝ aoole # all = find & replace.
↳ global substitution

↳ .sub!() ✓
.gsub!() ✓

puts "(555)-555 1234".gsub(" ", "").gsub("(", "").gsub(")", "").gsub("-", "")
↳ 5555551234

ε,
puts "(555)-555 1234".gsub(/[-\s\(\)]/, "") ⇝ 5555551234
spaces ↳ ( )
-

17.11) Rubular.com - a Ruby regular expression editor ☆☆
↳ get an immediate response of how our regex will be evaluated on a string.
↳ eg. \d{3}-\d{3}-\d{4}

13) Hashes II

13.1) with default value

↳ fruit_prices = Hash.new(0) ⇝ default value
or .new("Not found")
fruit_prices[:banana] = 1.05
⋮
p fruit_prices[:steak] ⇝ 0 ← default value

fruit_prices.default = "Whoops!" ⇝ overwrite default value
↳ "
⇝ "Whoops"

13-2) Convert Hash to Array, vice versa.

p spice_girls.to_a → array of arrays.

↳ .flatten → [[k1, v1], [k2, v2]]

puts power_rangers.to_h → to hash.

13-3) .sort & .sort_by methods on Hash

p pokemon.sort → [[:k1, v1], [k2, v2]] * Array sorted by key in alphabetical order.

↳ .reverse → [[k3, v3], [...]]

To sort by value instead of keys:

p pokemon.sort_by { |pokemon, type| pokemon }.reverse

" " " type }.reverse

13-4) .key? & .value? method on hash.

puts cars.key?(:key1) → true/false * :symbol as parameter.

↳ .value?("value1") → true/false.

13-5) Hashes as argument of method

```
def calculate_total(info)
  tax_amt = info[:price] * info[:tax]
  tip_amt = " * info[:tip]
  info[:price] + tax_amt + tip_amt.
end
bill = { tip: 0.18, tax: 0.07, price: 24.99 }
p calculate_total_2(bill) → 31.2375
```

13.6) .delete method on Hash

- hash.delete(:key) ≈ pop, can be stored.

13.7) .select & .reject method on Hash

```
hash.select{|key, value| value >= 5}
```

↳ select k-v pair that fits the criteria

```
hash.reject {|key, value| value >= 5}
```

↳ select key pair that not fits the criteria.

13.8) .merge method to combine Hashes.

```
hash1.merge(hash2) => hash_merged.
hash1.merge!(hash2)
```

```
def custom_merge(hash1, hash2)
  new_hash = hash1.dup
  hash2.each do |key, value|
    new_hash[key] = value
  end
end
```

13.9) Challenge: Word Frequency

↳ Given a string, write a method that return a hash where the keys will represent the words in the string & the values will represent its occurrences (split, iterate, => hash)

```
def word_count(string)
  words = string.split(" ") => array of words
  count = Hash.new(0) => set default value as 0
  words.each {|word| count[word] += 1}
  count
end
```

13.10) Quiz :

* hash = {"A" : "a"}
  hash.key?(:A) ⇒ false * ☆ :A ≠ "A"

* hash1 = {Lincoln: 5}
  hash2 = {Lincoln: "5 dollars"}
  hash1.merge(hash2) ⇒ {Lincoln: "5 dollars"}

* sort, sort_by ⇒ multi dimensional array.

15. Intro to the Time object

default = now / 0.

Time.new(yr, mth, date, hr, min, sec)

- puts Time.now ⇒ represent now.
  eg. today = Time.new or Time.now
  p today.class ⇒ Time.
  - used when we want to pass arguments to specify a custom time object with a specific date time. (eg. specific mth, yr, day, hr, min, sec)
  - Without arguments, Time.new = Time.now
- Time.new(2015) ⇒ 2015-01-01 00:00:00 # year.
  - (2015, 5) ⇒ 2015-05-01 " # mth
  - (2015, 5, 18) ⇒ 2015-05-18 " # date
  - (2015, 5, 18, 23) ⇒ 2015-05-18 23:00:00 # hr
  - " ,32) ⇒ " 23:32:00 # min
  - ,21) ⇒ " 23:32:21 # sec

15.2) Instance methods on Time objects

eg. .month .hour
.day .min
.year .sec
.yday : actual # days of the year. eg. 18 Feb ⇒ 31+18 = 49
.wday : day of the week * Sunday = 0 eg. Tuesday ⇒ 2

1.5.3) Predicate / Boolean methods on Time Objects.

- end with ?
- .monday? ; tuesday? ; wednesday? ; thursday? ; friday? ; saturday? ; sunday?
- .dst? (daylight saving Time)

1.5.4) Add & Subtract Time by seconds.

+60 → +60 sec / +1 min

-180 → -180 sec / -3 min

+(60*3) → 3 min

+(60*60) → 1 hr.

+(60*60*24) → 1 day.

- *45 → 45 days.

Ex.

```
def find_day_of_year_by_number(number)        → #day.
  current_date = Time.new(2016,1,1)
  one_day = 60*60*24
  until current_date.yday == number
    current_date += one_day
  end
  current_date
end
p find_day_of_year_by_number(~~number~~ 150)  → 2016-05-29 ...
                             (366)            → 2016-12-31 ...
```

1.5.5) Comparable methods on a Time Object

>=, >, <, <=, ==, !=

.between?(date1, date2) → exclusively?

15.6) Convert Time object to others objects.

- .yday ; .wday ; .mday
- .to_s
- .ctime → Tue Feb 15 00:00:00 ... → Timezone
- .to_a → [0, 0, 0, 15, 2, 2000, 2, 46, false, "EST"]
  (hr, min, sec, day, mth, yr, wday, yday, not DST)

15.7) Converting Time Object to Formatted String

- 'search: ruby strftime' to customize how it looks. * ruby strftime
- %b, %B, %d, %j, %m, %w, %x, %y, %Y
- p Time.now.strftime("%B %d, %Y") => December 29, 2019...

~~"%B %d..~~

15.8) .parse & .strptime methods

* require 'time'
* puts Time.parse("2016-01-01") => 2016-01-01 00:00 -....
  - .class => Time
* puts Time.strptime("03-04-2000", "%m-%d-%Y")
  - 2000-03-04 00:00:00

18) Classes

18.1) Intro to Classes.

- blueprints for creating objects; * array -> storing things in sequence; hash -> storing connections
- The goal of object oriented design is to use code to model real-life things as classes
- Object is a custom container for data that made from a class.

- objects group functionalities into single unit
- Instance variables (attributes) are variable belonged to that object;
  - attributes provide info about the object's current state
- Instance methods interact with the object, belonged to that object.
- Encapsulation is a OOP concept that restricts direct access to an object.
- Object is an instance of the class; class is the blueprint, pattern, template, outline of an object.
- Instantiation is the process of creating object from a class.
- A class is called an abstract type because it hides the complexity of interacting with the data, much like a method abstracts the complexity of a process.

18.2) Review .class method

~~18.3)~~ ↳ Fixnum, Float, String, Array, Hash, TrueClass, FalseClass, NilClass, Range, Regexp, Proc, Proc (lambda), Time

18.3) .superclass & .ancestors methods on a class.

↳ class itself is another object.

↳ Every class in Ruby (except top class - BasicObject) has ≥1 superclass.

↳ eg. Numeric
- Integer
  - Fixnum
  - Bignum
- Float

↳ .superclass ⇝ 1 class above

↳ .ancestors ⇝ an array of all superclasses inherited

↳ eg. 5.class.superclass ⇝ Integer
- .superclass ⇝ Numeric
- .superclass ⇝ Object
- .superclass ⇝ BasicObject

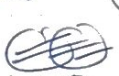
5.class.ancestors ⇝ [Fixnum, Integer, Numeric, Comparable, Object, Kernel, BasicObject]
(Comparable, Kernel: modules)

18.4) .methods method.

↳ return an array of all of the ~~objects~~ methods available instance methods.

eg. puts fixnum_methods & float_methods

A1 * | A2 ⇝



A1 - A2 ⇝

eg. 5 methods.sort

18.5) → Create a class

Ruby
- Classes
  - gadget.rb

```
class Gadget
end
```

* Gadget.superclass → Object.

```
phone = Gadget.new
phone.class → Gadget.
puts phone.is_a?(Gadget) → true
puts phone.respond_to?(:class) → true
                (:is_a?) → true
```

(:is_a?) ← method name; ↑ symbol *

18.6) Object Aliases
- * cmd+/ to hide panel (Atom IDE)
- * .object_id → ...
- a = Gadget.new
  b = a → * pointing to the same object, same location in memory.
  b.object_id == a.object_id → true
  b == a → true
- c = Gadget.new
  c == a → false although looks same.

18.7) Instance variables & .initialize
- variables that belong to a specific object, define object's properties/attributes, make up the object's state; each object can have a custom state; state of an object can change over time.
- begin with @ symbol (eg. @name)
  - is called sigil, a special character that denotes the var's scope.
- without it, it'd be interpreted as a local var to the method it's used in.
- Initialize method.
  - called immediately when an object is instantiated from a class with the .new method.
  - assign values to instance variables; define the object's state
  - if not defined, the object is initialized with no state
  - The object's state can be altered later.

```
class Gadget ( username, password, p-n)
  └ def initialize
      @username = username   "User #{rand(1..100)}"
      @password = password   "topsecret"
      @production_number = p-n   "#{("a"..."z").to_a.sample}-#{rand(1..999)}"
    end
end
                                  eg. k-21
phone = Gadget.new
puts phone.inspect  ⇝ #<Gadget:...>
p. phone
```

* initialize is a private method, never meant to be called.

p.phone.instance_variables ⇝ [@username, @password, @production_number]

18.8) Instance Methods

└ methods belonged exclusively to the object.

```
class Gadget
  └ def info          * instance variable:
      "Gadget #{@production_number} has an username of #{@username}"
    end

puts phone.info ⇝ Gadget ...
```

es. receiver of instance methods:

eg. puts phone.methods - Object.methods ⇝ info.

18.9) Override the .to_s method.

```
def to_s  ⇝ to_s is overriden from class Object
end
```

18.10) The self keyword in an Instance method.

- within an instance method body, the self keyword will refer to the current object.
- needed because you can't predict what variable it is stored.

  eg. phone = Gadget.new
  ↑ self

  - Referring to the current object, treat it like a receiver & call a method on it.

  eg. def info
  "It's object id is #{self.object_id} and is made of #{self.class}, having a superclass of #{self.superclass}."
  end

  - variables are really just placeholders of the object, but we need some kind of way to refer - that specific (name, label) object in memory & we have no clue what the user is going to use as their variables ~~or~~.
    - self.

- kind of tricky in Ruby ∴ it refers to different things in diff. contexts
- self refers to the object at hand when it's used in an instance method.
- outside of the instance method, refer to ~~it's~~ the Class itself.

18.11) Getter Methods. (Reader method)

- Instance variables can't be ~~are~~ accessed by default (OOP principle - encapsulation)
  - Your data should be protected, hidden, unless you give specific permission.

~~- An object state~~
- to protect data
- Getter methods: methods that get the value of an instance variable.
  - write-only field: a field that you can only write to. (eg. password...)
    read-only field: eg. production_number
    r&w field: eg. username ~~then~~
  - Instance methods have access ~~& can call~~ to instance to all instance variables, & can call other instance methods.

- eg. def production_number → same name ★
  @production_number
  end

18.12) Setter method (Writer method)

- Responsible for setting a new value for an instance variable
- Modify the object state

eg. def password=(password) ← ★ best practice to indicate that's a writer method (visual signal)
  @password = password → argument/param, disappear a.s.a the method is done.
  ↑ instance variable.
end

def username=(new_username)
  @username=new_username
end

eg. phone.username=("Kyle") ✓

18.13) Shortcut Accessor Method.

attr_accessor :username
attr_reader :production_number
attr_writer :password

replace all above. (setter & getter methods)

attr_writter :password, :username

★ phone.username = "Kyle" ★

18.14) Add parameters to initialize method

def initialize(username, password)
  @username = username
  @password = password
  @production_number = #{("a".."z").to_a.sample} - #{rand(1..999)}
end

phone = Gadget.new("Kyle", "Lehele") ✓

14) Blocks, Procs, & Lambdas

14.1) Intro to Blocks.

- A collection of code; must be attached to a method; alter the execution of the method; Block is NOT an argument / parameter to the method; Blocks can be defined with {} or do end; A block can get or update the value of local variables within the block.
- Methods vs. Blocks
  - Methods can be invoked over & over; Block will only be called once, then disappear; Block isolates actions away from the method
  - Methods take care of what is common, consistent, and constant each time, Blocks take away some of the functionality that is custom or unique to that specific method invocation.
- eg. .each method

```
[3,5,7,9].each {|num| puts num**2}
```

(|num| → Block variable)

- .each → Method
  - its core functionality is to iterate over each element within the array
- {|num| puts num**2} → Block
  - specifies what we want to do with each actual element within an array.
  - * Take care some of the exclusivity.

14.2) The yield keyword

- yield: transfer control from the method to the block that is attached to the method call.
- When we have the 'yield' keyword, the method pauses execution & wait until the block is done executing whatever is within it.
- Allows a layer of customization to how the method operates.
- eg.

```
def pass_control
  puts "a"
  yield
  puts "b"
end
pass_control { puts "hehe" }
```

→ a, hehe, b

* when use 'yield', Ruby expects a block after the method, else error.

```
pass_control do
  puts "hoho"
  puts "hehe"
end
```

→ a, hoho, hehe, b

- Basically temporarily exiting the method to do whatever is in the block.
- * Blocks implicitly return the last evaluation back to the method that calls them. (because the return is implicit, you should not include 'return' keyword whenever you're writing a block → else error)
- eg.
```
def who_am_i
  adjective = yield
  puts "I am very #{adjective}"
end
who_am_i {"handsome"}
```
→ I am very handsome.

eg
```
def multiple_pass
  puts "a"
  yield *
  puts "b"
  yield *
end
multiple_pass {puts "hoho"}
```
→ a
hoho *
b
hoho *

```
if result = multiple_pass {puts "hoho"}
  puts result → nil   (return : puts) → nil *
* without puts → "hoho"
```

∴ yield transfer control ≥1 times from inside a method to a block that's going to follow the method.

14.3) Procs I

- A new type of object, function as a saved block.
- Whenever you have a block that you want to reuse over & over again and you don't want to rewrite it, → create a Proc!

eg. ⌜ a = [1,2,3,4,5]
b = [6,7,8,9,10]
c = [11,12,13,14,15]
a_cubes = a.map {|num| num**3}
b " b "
c " c "

why not methods? What if **2, 4, 5? We don't want to create separate methods over & over again for each of these processes... (?)
We need a way to keep that general functionality with / method & keep the rest of it in... (?)
⇨ Proc.

Block vs Proc?
└ Block is not an object; it cannot have a method be called on it; can't save it to a variable; gone as soon as the method calling it is done executed, block & block variables ceased to exist, It is temporary.
└ Proc: an object that functions like a saved block, designed to be reused; think of them as a consistent procedure; does not care about its arguments and it's entirely self-contained.

⌜ Proc.new {|number| number ** 3}
cubes = ← * to let Ruby knows it's a Proc, not variable.
a.map(&cubes)

squares = Proc.new {|number| number**2}
b.map(&squares)
a_cubes, b_cubes, c_cubes = [a,b,c].map {|array| array.map(&cubes)}

eg. ⌜ ages = [10, 60, 25, 78, 83]
is_old = Proc.new {|age| age > 55}
p ages.select(&is_old) → [60, 78, 83]
" .reject(") → [10, 25]

14.4) The block_given? method.

```
if block_given?
  yield
end
```

↳ yield if block_given?

14.5) Yielding with argument. ☆☆

```
def speak_the_truth
  yield("Kyle") if block_given?
end
speak_the_truth {|name| puts "#{name} is brilliant!"}
```

↳ argument

```
def speak_the_truth(name)
  yield(name) if block_given?
end
speak_the_truth("Kyle") {|name| puts "#{name} is brilliant"}
```

↳ param as argument

```
def speak_the_truth(name)
  yield(name) if block_given?
end
speak_the_truth("Kyle") do |name, age|
  p name
  p age
end
```

↳ invalid argument

→ extra block argument

```
def num_eval(n1, n2, n3)
  puts "a"
  yield(n1, n2, n3)
  puts "b"
end
num_eval(5, 10, 15) {|n1, n2, n3| n1*n2*n3}
```

↳ arguments → ☆() optional

14.5) A custom .each method

```
def custom_each(array)
  i=0
  while i < array.length
    yield(array[i])
    i+=1
  end
end

custom_each([1,2,3,4,5]) do |num|
  puts "The square of #{num} is #{num**2}."
end
```

14.6) Procs II.

~~def greeter~~ hi = Proc.new { puts "Hi there." }

hi.call → Hi there. (execute the Proc itself, not frequently used)

5.times(&hi)

eg.
```
def greet
  puts "a"
  yield
end
greet(&hi)
```
→ a Hi there.

14.7) Pass a Ruby Method as Proc ☆

helpful for select, reject, map

```
p ["1","2","3"].map { |num| num.to_i }
p ["1","2","3"].map(&:to_i)
p [16, 14, 25].map(&:to_s)
```

[1,2,3,4,5].select(&:even?) = [1,2,3,4,5].select {|num| num.even?}

[1,2,3,4,5].reject(&:odd?) = [1,2,3,4,5].reject {|n| n.odd?}

14.8) Methods with Proc Parameters.

```
def talk_about(name, &mypro)
  puts "a"
  my_prc.call(name)
end

proc1 = Proc.new do |name|
  puts "#{name} is handsome"
}
talk_about("Kyle", &proc1)   → Kyle is handsome
```

14.9) Lambdas intro

↳ ≈ Proc, from Proc class

```
lambda1 = lambda {|n| n**2}   ✓
[1,2,3].map(&lambda1)         ✓ * need & in .map
```

↳ more popular ∵ more similar across programming languages

1st diff: How they treat wrong number of arguments.

Proc	Lambda
ignore	error

↳ eg.
```
some_proc = Proc.new {|a,b| "#{a} meets #{b}."}
p some_proc.call("Kyle","Lily")   → Kyle meets Lily.
p some_proc.call("Kyle")          → Kyle meets nil.

some_lambda = lambda {|a,b| "#{a} meets #{b}."}
some_lambda.call("Kyle")          → error.
```

2nd diff: How they return value.

```
def diet
  status = lambda {return "You gave in"}
  status.call
  "You completed the diet"
end

result = diet
p result   → You completed the diet.
```

← this is return more popular

```
def dief
  status = Proc.new { return "a" }   <- this is returned.
  status.call
  "b"                                -> not proceeded
end
```

∴ lambdas returns control back to the method, Proc returns within methods; lambdas care about # arguments received, Proc don't.
When defining methods, lambdas as argument require '&', Proc do. * map, both need '&'.

(14.10) Lambda Efficiency Example
↳ ↑ scalability.

(9) Modules & Mixins

(9.1) Intro

∴ Modules -> a toolbox or container of methods & constants

~~↳ Module methods and constants can be used as needed~~

↳ Modules create namespaces for methods with the same name; cannot be used to create instances; can be mixed into classes to add behavior

↳ Syntax & Style

↳ Module names written in UpperCamelCase; Constant in modules -> ALL CAPS; Access module methods with dot operator (.); access module constant with the :: symbol; -> scope resolution operator

```
module LengthConversions
  WEBSITE = "www.google.com"
  def self.miles_to_feet(miles)        -> self
    miles * 5280
  end
  def self.miles_to_inches(miles)
    feet = miles_to_feet(miles)
    feet * 12
  end
end

def self.miles_to_centimeters(miles)
  inches = miles_to_inches(miles)
  inches * 2.54
end

puts LengthConversions::WEBSITE
puts LengthConversions.miles_to_feet(100)
```

refer to

19.2 Modules with identical Methods.

```
module Square
  def self.area(side)
    side * side
  end
end

module Rectangle
  def self.area(length, width)
    length * width
  end
end

module Circle
  PI = 3.14
  def self.area(radius)
    PI * radius * radius
  end
end

puts Square.area(5)      → 25
puts Rectangle.area(5,10) → 50
puts Circle(5)           → 78.5
```

19.3 Import Modules into the Current File

* store 1 module per Ruby file

```
require "./rectangle.rb"   → require_relative "rectangle"
require "./square"         → .rb removed /  ./ remove
require "./circle"

puts Circle.area(10)
puts Rectangle.area(13,9)
```

19.4) The Math module

```
puts Math::PI
puts Math.sqrt(9)  → 3
puts Math.sin(90)  → 1
puts Math.cos(0)   → 1
```

19 5) Mixins . Part 1

- a module that's mixed into a class
- Why mix in modules to classes?
  - Diff classes need similar functionalities, even with diff superclass.
  - eg. Comparison Operators are needed for both Numeric & String class.
- DRY → Do not Repeat Yourself.

```
class OlympicMedal
  include Comparable    # .between?, <=>, >=, <, >, <= ...
  def initialize(type, weight)    > MEDAL_VALUES = { Gold: 3, Silver: 2, Bronze: 1 }
    @type = type                    attr_reader :type
    @weight = weight
  end
  def <=>(other)
    if MEDAL_VALUES[type] < MEDAL_VALUES[other.type]
      -1
    elsif MEDAL_VALUES[type] == MEDAL_VALUES[other.type]
      0
    else
      1
    end
  end
end
bronze = OlympicMedal.new("Bronze", 5)
silver = OlympicMedal.new("Silver", 10)
gold = OlympicMedal.new("Gold", 3)
```

(9.6) Mixin, Part II

mixin — A mixin is a module that injects additional behavior into a class
- Mixins allow us to mimic inheritance from >1 class
- A class that includes a module has access to its methods & constants.
- Constants & methods mixed into a class do not need to be prefixed with the module name.

- Modules (Mixins) vs. Inheritance
  - Class inheritance should be used is-a relationship
    - everything should be a type of the thing above it (eg. Fixnum is a type of Integer).
  - Modules should be used for a has-a relationship. It adds functionality.
    'eg. a String has a Comparable feature set.
    - Class has a Module's tools. (modules.)

- Method Lookup Path
  - last modules included will be used first.

```
module Purchaseable  ~ often end as 'ble' 'able'.
  def purchase(item)
    "#item is purchased"
  end
end

class Bookstore
  include Purchaseable
end

popular = Bookstore.new
p popular.purchase("Book1")  ✓

class Supermkt
  include Purchaseable
end.

shop1 = Supermkt.new
p shop1.purchase("meat") ✓

class CornerMart < Supermkt
end

mart1 = CornerMart.new
p mart1.purchase("item")  ✓
```

19.7) .ancestors method in depth

- .ancestors ⇝ array of super classes. * Call on Class

eg. Bookstore.ancestors ⇝ [Bookstore, Purchaseable, ...]

order of method lookup path.

* CornerMart.ancestors ⇝ [CornerMart, Supermkt, Purchaseable, ...]

19.8) The prepend keyword

~~eg. class Books~~ ↳ used as include, but at highest method lookup order.

≈ !important in CSS.

```
eg. class Bookstore
      prepend Purchaseable
    end
```

* Bookstore.ancestors

↳ [Purchaseable, Bookstore, ...]

19.9) The extend keyword.

' add modules methods into Class level rather than object level;

↳ methods for Class, not its instances.

```
eg. class Bookstore
      extend Purchaseable
    end

    p popular.purchase(-) ⇝ error
    p Bookstore.purchase ⇝ ✓
```

* can be used with 'self'

```
eg. module Announcer
      def who_am_i
        "The name of this class is "#{self}"
      end
    end
    class Dog
      extend Announcer
    end

    Dog.who_am_i ✓ ... is Dog.
    dog1 = Dog.new
    dog1.who_am_i ✗ ⇝ error
```

(19.10) The Enumerable Module

```
class ConvenienceStore
  include Enumerable
  attr_reader :snacks
  def initialize
    @snacks = []
  end
  def add_snack(snack)
    snacks << snack
  end

  def each
    snacks.each do |snack|
      yield snack
    end
  end
end

store1 = ConvenienceStore.new
store1.add_snack("snack-1")
                 "       2
                 ^       3

store1.each {|snack| puts snack} ⇝ test if Enumerable module works.
p store1.any? {|snack| snack.length > 10} ⇝ false.
        ↳ part of Enumerable methods.
p store1.all? {|snack| snack.length > 3} ⇝ true

p store1.map {|snack| snack.upcase} ⇝ ["SNACK-1", ... ]
p store1.select {|snack| snack.downcase.include?("s")} ⇝ x.
p store1.reject {|snack| snack.upcase.include?("J")}
```

required everytime including Enumerable.

20) Classes II

20.1) Private methods.

- To make a method private (only usable by Class variables), simply + 'private' above a method.
  - only accessible to us, programmer. ≈ backend...

20.2) Protected Methods

- Review of methods
  - Public - allow interaction & be called by any other object
  - Private: can only be called within the object; only the current object is able to receive the method.
- Protected methods
  - can be called within the same family, class of objects; used to compare objects of the same class.
  - primarily used to compare 2 objects, able to call a method from another object but you don't want it to be a public reader method

eg

```
def compare_car_with(car)
  self.value > car.value ? "Your car is better!" : "Your car is worse"
end

protected
def value
  @value
end
```

} this method can't be called outside of the object

eg. civic = Car.new(1,10000)
civic.value ⇝ error
civic.compare_car_with(car2) ✓

```
class Car
  def initialize(age, miles)
    base_value = 20000
    age_deduction = age * 1000
    miles_deduction = (miles / 10.to_f)
    @value = base_value - age_deduction - miles_deduction
  end
```

20.3) Add Validation to Setter Method

└ eg. check password pattern.

```
def password=(new_password)
  if validate_password(new_password)
    @password = password
  end
private
def validate_password(new_password)
  new_password.is_a?(String) && new_password.length >= 6 && new_password =~ /\d/
end
```

{ @password = password 하는 것.

phone = Gadget.new("keyk", "programming123")

phone.password = 123 → 이렇게 happens as criteria not fulfilled

phone.password = "test"

phone.password = "computer" ✓

phone.password = "computer 123" ✓

↑ spaces ok. 오오 ...

∴ attr_writer remove ∵ we have custom setter. ✓.

20.4) Prefer Instance Methods to instance variables

└ general practice.

```
def to_s
  "Gadget #{@production_number} has the username #{@username}
```

* @ remove ; #{production_number} = #{self.production_number}

method called on 'self'.

20.5) Calling multiple setter methods within 1 method

eg. ⌜ def reset(username, password)
  self.username = username
  self.password = password
  self.apps = [ ]
end

⇒ * use 'self' keyword to provide those new values

private
attr_writer :apps ⇒ can be only invoked by reset method (within class).

methods

↳ to group together various functionalities such as resetting multiple instance variables or assigning new values to them in single step.

↳ Represents the most secure & non-duplicative way of accomplishing operations like this.

20.6) Structs

↳ A mini class with less functionality, a Class with only instance variables;

↳ More complex objects with more complex methods / operations should be be represented as Classes (eg. validation),

↳ Framework for an object ⇒ Struct, simply representing an object with constant state, without much methods / various changes undergone.

eg. ⌜ module AppStore
  Struct.new(:name, :developer, :version)
  App =
  APPS = [
    App.new(:Chat, :facebook, 2.0)
    App.new(:Twitter, :twitter, 5.8)
  ]

```
def self.find_app(name)
  APPS.find{|app| app.name == name}
end
end
```

At gadget.rb:

- require_relative 'app_store'
- def install_app(name)
    app = AppStore.find_app(name)
    @apps << app unless @apps.include?(app) *
  end
- def delete_app(name)
    app = apps.find {|installed_app| installed_app.name == name}
    apps.delete(app) unless app.nil? *
  end

- g = Gadget.new("Kyla", "password")
- p g.apps
- g.install_app(:Chat) ✓
- p g.apps ⇝ [#<struct AppStore::App name=:Chat, ...>]
- g.delete_app(:Chat) ✓

20.1) Monkey Patching

- The process of modifying / adding features to a predefined class to an existing class; even we have a class definition, we can later reopen it & add new things. (eg. new instance methods); allows you to make changes to its own existing classes

eg. ⌜ class Array

```
  def sum      # ~~meth~~ doesn't exist in Array methods (won't override)
    total = 0
    self.each { |el| total += element if element.is_a?(numeric) }
    total
  end
end
p [1, "H", 2, "a", 3].sum  ⇝ 6
```

eg ~~ouch class~~ ⌜ class String

```
  def alphabet_sum
    alphabet = ("a".."z").to_a
    sum = 0
    self.downcase.each_char do |character|
      if alphabet.include?(character)
        numeric_value = alphabet.index(character) + 1
        sum += numeric_value
      end
    end
    sum
  end
end
puts "abc".alphabet_sum ⇝ 6
puts "zzz".alphabet_sum ⇝ 78
puts "HELLO. WORLD".alphabet_sum ⇝ 124.
```

20.8) Monkey Patching 2.

⌜ class Hash

```
  def identify_duplicate_values
    values = []
    dupes = []
    self.each_value do |value|
      values.include?(value) ? dupes << value : values << value
    end
    dupes.uniq
  end
end
```