CLASSES OBJECTS METHODS

CLASSES

Syntax

simple class definition

- class ClassName

  end

defining class with attribute accessors

- class Person
  attr_accessor :name , :age
  attr_reader :name , :age
  attr_writer :name , :age

    def initialize

    @name = "person name"
    @age = 0

    end

  end

defining a class constructor

- class GiftBox
  attr_accessor :id , :description

    def initialize (id: , description: "electronic")

    @id = id
    @description = description

    end

  end

why do we have classes ? it is so we are able to define and create objects.
Think of everything as an object.

OBJECTS

creating objects

Simple syntax to creating a object

- object_name = ClassName.new

Create object using it's constructor (initialize method)

- xmas_gift = GiftBox.new (id: 2734 , description: "toy")

Create multiple objects and pass it into an array

- array = [ ]
  count = 0

  while count < 5
  array << GiftBox.new
  count += 1
  end

EXAMPLE : CLASSES , OBJECTS

```
require_relative 'person'
require_relative 'gift_box'

class GiftStop

    @gift_types = %w/electric sweets drinks books apparel/
    @persons = %w/Alex Iryna Bella Steffano Yujing/

    @visited_persons = []
    @gifts_given = []

      count = 0
      while count < 5
        @visited_persons << Person.new
        @visited_persons[count].name = @persons[count]
        @gifts_given << GiftBox.new(id: count , description:
@gift_types[count] )
        count += 1
       end

    @visited_persons.each_with_index { |person, idx| puts "#
{person.name} was given a # {@gifts_given[idx].description}" }



 end
```

in the above example lets assume that people visited the ' giftstop ' in the order of the array index. and the gift given is in the same order (index is the same)

CLASSES OBJECTS METHODS

METHODS

Syntax

simple method definition

- def method_name

  end

methods with parameters

- def print_visits(param1 , param2)

    print " #{param1} and #{param2} "

  end

or

- def print_visits(param1: , param 2:)

    print " #{param1} and #{param2} "

  end

methods with default values for the parameters

- def print_visits(param1: "visitors" , param2: 25)

    print " #{param1} and #{param2} "

  end

The above method declaration allows the user to decide which parameter he/she is inputting values for.

Class methods

- def **self.**print_visits(param1: "visitors" , param2: 25)

    print " #{param1} and #{param2} "

  end

class methods can be used to call methods without instantiating objects.

EXAMPLE : CLASSES , OBJECTS , METHODS

creating a new class with a method (print_visits)

- class Utility

  def print_visits (visitors: , gifts:)

  visitors.each_with_index { |person, idx| puts "#{person.name} was given a #{ gifts[idx].description} " }

  end

end

refactoring earlier 'GiftStop' class using utility class and methods

```
require_relative 'person'
require_relative 'gift_box
require_relative 'utility'

class GiftStop

  @gift_types = %w/electric sweets drinks books apparel/
  @persons = %w/Alex Iryna Bella Steffano Yujing/

  @visited_persons = []
  @gifts_given = []

    count = 0
    while count < 5
      @visited_persons << Person.new
      @visited_persons[count].name = @persons[count]
      @gifts_given << GiftBox.new( id: count ,  description: @gift_types[count] )
      count += 1
    end

  helper = Utility.new

  helper.print_visits(visitors: @visited_persons , gifts: @gifts_given)

end
```

Observe how using methods improved code readability and reusability.
since we use the utility class object only to invoke its methods we can also define print_visits as a class method

def **self.**print_visits (visitors: , gifts:)

and use it like :

Utility.print_visits(visitors: @visited_persons , gifts: @gifts_given)