**Lesson 5**

**I. Sorting:**

* Use sort or sort\_by.
* <=> called the ‘spaceship’ operator, this method compares two objects, if the left is bigger then the right = 1, if the left is smaller than the right = -1. If objects cant be compared = nil and i get argument error.
* Sort takes the return value of ⇔ (-1,0,1) and sorts. ASCII table used to translate character to numbered/ordered value.
* Order: number, Uppercase, lowercase, use .ord to find place in table
* .sort is the same as .sort do |a, b| a ⇔ b end, use b ⇔ for descending order.
* String is sorted char by char and the char that has more will be larger
* Array compares first element in each arry then second etc, then compares the array as a whole. If the first element in each array are not equal and the second elements return nill, i dont get an argument error because the sort stops there and uses the sort results from the first elements.
* Sort\_by always returns an array, is usually called with a block and the block is used to determine the items to be compared. To sort by symbol i must first convert to string using .to\_s. Ie: sorting using char in index 1
  + [‘cot’, ‘bed’, ‘mat’].sort\_by { |word| word[1] } => [‘mat’, ‘bed’, ‘cot’]
  + Hash: people = { kate: 27, john: 25, mike: 18 }

People.sort\_by { |name, age| age }

=> [[:mike, 18], [:john, 25], [:kate, 27]]

* Mutate by using bang(!) sort! And sort\_by! If not, i will get a sort returned but the original object will not be sorted.
* If sorting numbers as strings “11” would go before “7” because the first 1 in “11” is compared with the first item in “7”, to sort and have “11” come after “7” i must convert to integer .to\_i so that object isnt compared line by line.

**II. Nested Data Structures:**

* Arrays within an array ie arr = [[1, 3], [2]], each array has there own index
  + Selecting within nested array: arr[0][1] => 3
  + Modifying within nested array: arr[0][1] = 5 => [[1, 5], [2]]
  + Inserting within nested array: arr[0] << 3 => [[1, 3, **3**], [2]]
    - Basically arr[0] = [1, 3] << 3
* Hashes can be nested like arr = [{ a: ‘ant’ }, { b: ‘bear’ }]
  + Adding kv pair: arr[0][:c] = ‘cat’
    - Arr now => [{ :a => "ant", :c => "cat" }, { :b => "bear" }]
* Referencing the same object, pointing to the same object:
  + a = [0,1] b = [5] arr = [a,b]
    - A[1] = 8 is same as arr[0][1] = 8, two diff ways to refe same object
    - Remember: variables are POINTERS not actually the object
* Shallow copies (.dup & .clone) they both only copy the object/collection, if that object contains other objects like a nested array, those objects will be *shared*, not copied. If i mutate an object within the copied object, it effects the original.
  + .clone: preserves the frozen state of the object
    - arr1 = ["a", "b", "c"].freeze  
      arr2 = arr1.clone  
      arr2 << "d"  
      # => RuntimeError: can't modify frozen Array

* + .dup: does NOT preserve the frozen state of the object
    - arr1 = ["a", "b", "c"].freeze  
      arr2 = arr1.dup  
      arr2 << "d"  
        
      arr2 # => ["a", "b", "c", "d"]  
      arr1 # => ["a", "b", "c"]
* Freezing an object prevents it from being modified. Only mutable objects can be frozen, integers are already frozen. I can check with the .frozen? method:
  + str = "abc".freeze  
    str << "d"  
    # => RuntimeError: can't modify frozen String
  + Only freezes the object it’s called on, any objects within the frozen object is NOT frozen:
    - arr = [[1], [2], [3]].freeze  
      arr[2] << 4  
      arr # => [[1], [2], [3, 4]]
    - No deep copy/deep freeze: copying all of the nested layers

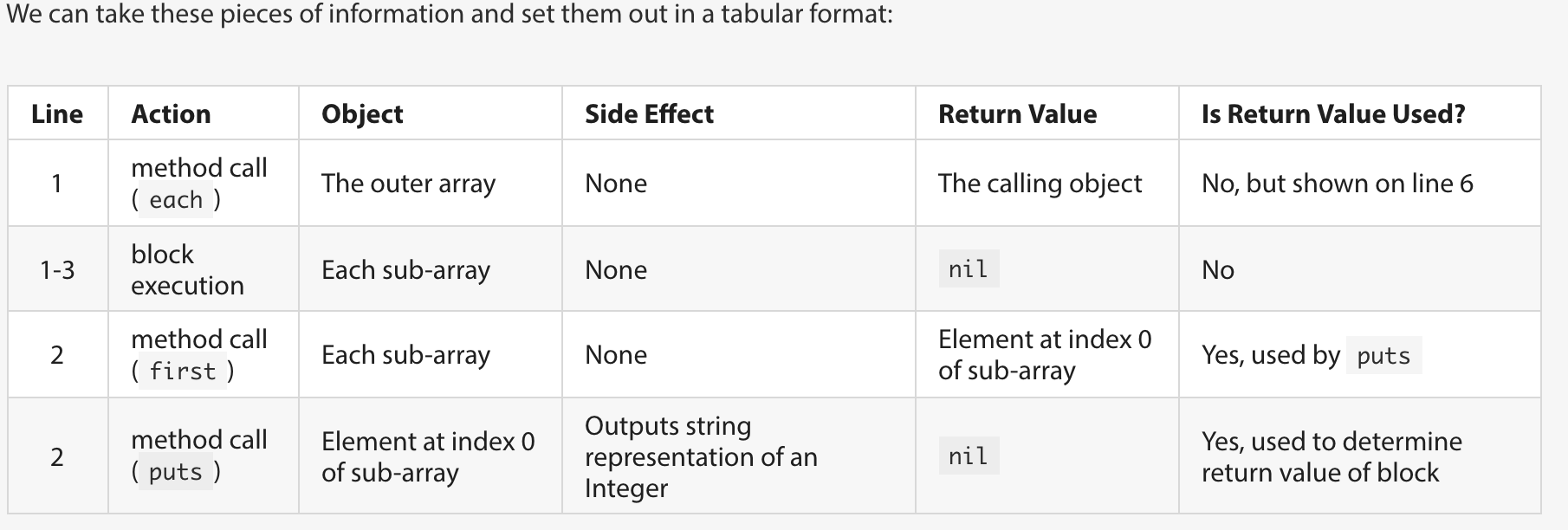
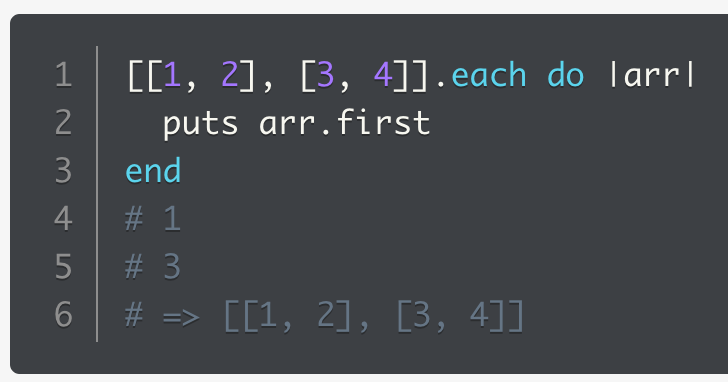
**Iterating**

**MAP**: Creates a **new array** containing the values returned by the block. Iterates through the array.

**EACH:** Returns the **original** object that was called

**SELECT:** Returns a new array containing all elements of ary for which the given block returns a **true** value.

**II. Working with Blocks:**

* Break down code line by line, approach it systematically and figure out what the return value and side effect is of each step.
* Keep track of the following when evaluating blocks of code
  + What is the type of action being performed (method call, block, conditional, etc..)?  
    What is the object that action is being performed on?  
    What is the side-effect of that action (e.g. output or destructive action)?  
    What is the return value of that action?  
    Is the return value used by whatever instigated the action?
* Nested arrays: I need to first access the nested arrays before I can ‘select’ values, use map or each, then next block can be ‘select’

[[8, 13, 27], ['apple', 'banana', 'cantaloupe']].**map** do |arr| *map to access nested objects*  
 arr.**select** do |item| *selecting from within nested obj*  
 if item.to\_s.to\_i == item # if it's an integer  
 item > 13  
 else  
 item.size < 6  
 end  
 end  
end  
# => [[27], ["apple"]]

One of the main reasons ***map*** is used in this example is not only to iterate over the array and access the nested arrays, but to **return a new array** containing the selected values. If we used ***each instead*** we wouldn't have the desired return value, and would need an extra variable to collect the desired results.