12 Ruby: Blocks

Blocks are one of the most powerful tools in the Rubyist's arsenal. They are also one of the hardest concepts for beginners to grasp, so prepare for some confusion, and discuss these exercises with your mentor.

So what is a block?

Blocks allow you to *pass* a series of instructions to a method as a special kind of argument.

You've already had some exposure to blocks in the `describe` and `it` blocks of the **Reading** `RSpec` **Tests** checkpoint, and when you used `each` in the **Loops** checkpoint.

As a refresher, here's an example of the `each` method:

```
items = [1, 2, 3, 4]
items.each do |item|
  p item + 3
end

#=> 4
#=> 5
#=> 6
#=> 7
```

Here's how this works:

- The `each` method is called on a collection (here, `items`).
- It's then passed a block, delineated by the `do` and `end`.

- The block has a block argument, `item`, declared inside "pipes." `item` represents the current element of the collection.
- When you call `each` on a collection, it iterates over each element in the collection and "calls" the block on it, with the element as the block argument.

> Blocks are basically anonymous chunks of code - methods without names - that can be passed as arguments to a method.

In the above example, our block gives the instructions to "print the value of the item we're iterating over, plus 3." We could just as easily pass a block with a different set of instructions, say, "print 'odd' if the value is odd." In the case of `each`, the block lets us take a collection and, essentially, say, "do some stuff with each element of the collection."

> Blocks allow us to dissociate the specific behavior of what we want to do as part of a loop (or other behavior) from the process of looping itself. Think of how we'd have to do this without a block.

If you have a simple block, like the example above, you can replace `do` and `end` with curly brackets. For example:

```
items = [1, 2, 3, 4]
items.each { |item| p item + 3 }

#=> 4
#=> 5
#=> 6
#=> 7
```

This is entirely equivalent to:

```
items.each do |item|
  p item + 3
end
```

Let's explore the variable in between the pipes (`|item|`), the **block argument**. In the example above, the `item` block argument represents an element from the `items` array. With every iteration of the `each` method, the `item` block argument becomes a different element from the `items` array. In other words:

- on the 1st iteration of the `each` method, `item` = `1`
- on the 2nd iteration `item` = `2`
- on the 3rd iteration `item` = `3`
- on the 4th iteration `item` = `4`

After the 4th iteration, the `each` method returns because there are no more elements to iterate over.

> Nothing about `each` specifies that it adds 3 to every element in its collection. That behavior is specified by the block. The `each` statement *yields* control to the block for each element in the array.

Block Returns

Blocks can also return values, and in the case of `each`, the return value of calling the block is the collection at the end of iteration. Because our above call to `each` didn't *change* the collection over which it iterated, the return value is the original `items` array: `[1,2,3,4]`.

`each` *does not alter* the collection it iterates. We use `each` when we want to do something *with* an array, not *to* it. See this distinction in action:

```
first_array = [1,2,3,4]

second_array = first_array.each do |item|
  item + 1000
end

p first_array
#=> [1,2,3,4] # Each does not alter the collection on which it's called

p second_array
#=> [1,2,3,4] # It also *returns* the original collection
```

> This means that `each` is useful **only** for its "side effect", such as printing. In Rails, `each` is often used to loop through and print items in collections.

Controlling Each's Return

Let's take this a little further and create a method to demonstrate how a block is executed. Consider the following example, where we define a method which takes an array and returns a new array with all of the original values incremented by 3:

```
def new_array_plus_3(items)

  # create an empty array to store the results of the block
  new_items = []

  # iterate over the items array, add 3 to each element
  # and store the results in the new_items array
  items.each { |item| new_items << item + 3 }

  # return the newly populated array
  new_items
end

array = [1,2,3,4]

p array
#=> [1, 2, 3, 4]

p new_array_plus_3(array)
#=> [4, 5, 6, 7]

p array
#=> [1, 2, 3, 4]
```

> We're still just using `each` for its side effects - here, "shoveling" values into a new array - but we then store the result of those side effects, and return them from our method. This is just for demonstration. As you'll see shortly, other block methods are better suited to the above purpose.

Other Block Methods

Ruby has many methods that take blocks. You've been introduced to the `each` and `each_with_index` Array methods, and RSpec's `describe`, and `it`. There is a whole world of other block methods in Ruby. We'll cover a few basic examples here so you can begin to get a feel for them.

> Don't worry about understanding these perfectly just yet. You'll have a chance to work with most

> of them in the exercises.

Most of below are collection methods:

Map

You'll recall that the `each` method iterates through a collection, does something with each member, and returns the original collection. The `map` method instead returns a new collection constructed from the previous one, based on the logic in the block. While `each` performs behaviors with a collection and returns that collection untouched, `map` does the former and then returns a *new* collection based on the return from the block code.

Specifically, it loops through the primary collection, filling a new collection with the elements from the first one, modified based on the block instructions.

This makes our `new_array_plus_3` method much briefer and easier:

```
def new_array_plus_3(items)
  items.map{ |item| item + 3 }
end

array = [1,2,3,4]

new_array_plus_3(array)
#=> [4, 5, 6, 7]

array
#=> [1, 2, 3, 4]
```

> Our new array is composed of the return value from each invocation of the block.

In Rails, we might use `map` to convert an array of users into an array of those users' names:

```
deadly_viper_assassination_squad =
  [{ id: 1, name: 'Beatrix' },
  { id: 2, name: 'O-Ren' },
  { id: 3, name: 'Vernita' },
  { id: 4, name: 'Elle' }]

deadly_viper_assassination_squad.map { |f| f[:name] }
#=> ["Beatrix", "O-Ren", "Vernita", "Elle"]
```

> The original `deadly_viper_assassination_squad` collection remains untouched; we just iterate through it to return a collection of names.

Select and Reject

`select` is called on a collection and returns a new collection of all the elements for which the supplied block returns true. Let's review the squad again:

```
# Note the addition to the hash representing Budd
deadly_viper_assassination_squad =
  [{ id: 1, name: 'Beatrix' },
  { id: 2, name: 'O-Ren' },
  { id: 3, name: 'Vernita' },
  { id: 4, name: 'Budd', betrays_squad: true }]

deadly_viper_assassination_squad.select { |f| f[:betrays_squad] }
#=>[{ id: 4, name: 'Budd', betrays_squad: true }]

deadly_viper_assassination_squad.reject { |f| f[:name].length > 6 }.map{ |f| f[:name
#=>['O-Ren', 'Budd']
```

> In the example directly above, we "chained" block methods. This is possible (and extremely useful), because the methods both return and are called on collections.

Any and All

`any?` returns `true` if *any* of the elements within the collection result in `true` returned from the block. `all?` works similarly, returning `true` if and only if each element within the collection resulted in a `true` return value from the block. Here are a few examples using our previous data:

```ruby
deadly_viper_assassination_squad.any? { |f| f[:name] == 'Beatrix' }
#=> true
deadly_viper_assassination_squad.any? { |f| f[:name] == 'Bill' }
#=> false
deadly_viper_assassination_squad.all? { |f| f[:name] == 'Quentin' }
#=> false
deadly_viper_assassination_squad.all? { |f| f[:id] > 0 }
#=> true
```

Yield

Yield is a difficult Ruby concept to grasp at first, but it's necessary to fully understand blocks. This example from **Tutorials Point** shows how `yield` works:

```ruby
def test
   puts "You are in the method"
   yield
   puts "You are again back to the method"
   yield
end

test {puts "You are in the block"}
#=> You are in the method
#=> You are in the block
#=> You are again back to the method
#=> You are in the block
```

> Our block does not take a "block argument". Like methods, blocks can be called with and without arguments. We'll discuss yielding with arguments in the exercises.

What's going on here? When our `test` method hits the `yield` command, it literally *yields* control to the "block" called as part of the method. The `yield` method "invokes" the block given, performing whatever actions the given block specifies, before handing control back to the method in which it was called.

> If no block is given when calling our `test` method, what do you think will happen?

```
test
#=> You are in the method
LocalJumpError: no block given (yield)
```


Exercises

- [x] ~~Sorting~~
- [x] ~~Mapping~~
- [x] ~~Any~~

Complete the following brief writing assignment after reading Bloc's **resource on writing to learn**.

Send your mentor a message via the Discussion tab for this checkpoint. In two to four sentences each, your message should answer the following questions:

- How does `each` differ from `map`? In what sorts of cases would you want to use the former? The latter?
- How would you describe the difference between `any?` and `all?` What data type do these methods return? Do they have any effect on the array on which they're called? (Try it using **repl.it**).
- **Bonus**: The `each` method itself uses `yield`. Why does it need `yield` to work as it does? What code is `each` yielding control to, and what does it feed

`yield` as a block argument? (Hint: It's yielding multiple times.)

Solution Walk-Throughs

Do not watch these videos until after you've attempted your own solutions. If you struggle to complete the checkpoint exercises, submit your best effort to your mentor *before watching a solution walk-through*. Submit your answers even if some tests don't pass, as it's important for your mentor to understand your thought process. Discuss difficult exercises with your mentor.

There are many possible solutions to the exercises. A walk-through is meant to provide insight into the thought process of *one possible solution* per exercise. If your solution is different yet passes all the tests, do not change your solution to match what is seen in the video. Instead, discuss your thought process with your mentor.

1. **Sorting**
2. **Mapping**
3. **Any**

assignment completed

?