Pass by Reference vs Pass by Value

Rubyists seem to disagree on whether Ruby is "pass by reference" or "pass by value". This assignment will provide an understanding of what those terms mean, and explain Ruby's behavior. In the end, it doesn't matter what you call it, as long as you understand how to invoke methods with the behavior you expect.

The discussion stems from trying to determine what happens to objects when passed into methods. In most programming languages, there are two ways of dealing with objects passed into methods. You can either treat these arguments as "references" to the original object or as "values", which are copies of the original.

What does pass by "value" mean?

In order to talk about what "pass by value" traditionally means, we'll talk about it using a traditional language -- C. In C, when you "pass by value", the method only has a *copy* of the original object. Operations performed on the object within the method have no effect on the original object outside of the method.

Some Rubyists say Ruby is "pass by value" because re-assigning the object within the method doesn't affect the object outside the method. Example:

```
def change_name(name)
  name = 'bob'      # does this reassignment change the object outside the method?
end

name = 'jim'
change_name(name)
puts name           # => jim
```

Note that the code example above has two different local variables named `name`. There is one scoped within the method, and there is one in the main scope. This is not variable shadowing, because the main scope variable is not accessible to the method. Within the method, we could have named the variable something other than `name`.

The question is: when the main scope `name` is passed into the method, via `change_name(name)`, did we pass in a reference or did we pass in the value?

It looks like it's by value, since re-assigning the variable only affected the method-level variable, and not the main scope variable.

What does pass by "reference" mean?

However, it's not quite that simple. If Ruby was pure "pass by value", that means there should be no way for operations within a method to cause changes to the original object. But you can plainly do this in Ruby. For example:

```
def cap(str)
  str.capitalize!   # does this affect the object outside the method?
end

name = "jim"
cap(name)
puts name           # => Jim
```

This implies that Ruby is "pass by reference", because operations within the method affected the original object. However, as we saw with the re-assignment example, not all operations affect the original object. For example, we'll slightly modify the previous code:

```
def cap(str)
  str.capitalize
end

name = "jim"
cap(name)
puts name           # => jim
```

If you can't see the difference, look carefully. It appears we're back in "pass by value" world again, where operations within the method don't affect the original. What's going on?

What Ruby does

As you can see from the previous examples, Ruby exhibits a combination of behaviors from both "pass by reference" as well as "pass by value". Some people call this *pass by reference of the value* or *call by sharing*. Whatever you call it, the most important concept you should remember is:

> when an operation within the method mutates the caller, it will affect the original object

This topic is covered in our Introduction to Programming book here. Re-read that section to refresh your memory.

The natural question is then, *which operations mutate the caller*? Unfortunately, this is something you'll have to slowly discover through usage and reading documentation. In the Ruby core library, a lot of destructive (another term for mutating the caller) methods end with a `!`. But that's just a naming convention, and it's not a guarantee.

For example, the `Array#<<` method is destructive, but doesn't end with a `!`.

```
def add_name(arr, name)
  arr << name
end

names = ['bob', 'kim']
add_name(names, 'jim')
puts names.inspect          # => ["bob", "kim", "jim"]
```

You can see that by sending the `names` array through the `add_name` method, it permanently changed the original `names` array.

Also, study the below code and remember that re-assignment is not considered a destructive operation.

```
def add_name(arr, name)
  arr = arr + [name]
end

names = ['bob', 'kim']
add_name(names, 'jim')
puts names.inspect          # => ["bob", "kim"]
```

Notice how the above code doesn't change the `names` array. Note that if we made a very tiny change to the above code, the result is dramatically different:

```
def add_name(arr, name)
  arr = arr << name
end

names = ['bob', 'kim']
add_name(names, 'jim')
puts names.inspect          # => ["bob", "kim", "jim"]
```

Can you spot the change? We changed the + to a << and moved `name` out of an array. This implies that when we use + to concatenate two arrays together, it is *returning a new array* and not mutating the original. However, when we use << to append a new value into an array, it is *mutating the original array* and not returning a new array.

It's a guarantee that you will run into this issue over and over in your programming career. Learn to train your brain to recognize this as a potential problem, and you'll be able to debug future problems much faster.

Variables as Pointers

Now is a good time to re-read the section Variables as Pointers. Understanding this will help explain why Ruby exhibits the behavior we have talked about in this assignment. Along with that, take some time to read the following articles, where these concepts are explored further:

- Variable References and Mutability of Ruby Objects
- Mutating and Non-Mutating Methods in Ruby
- Object Passing in Ruby - Pass by Reference or Pass by Value

Don't skip over this, as these are some of the most important concepts to understand in your journey to learning to program.

You marked this topic or exercise as completed.
More Variable Scope
Walk-through: Rock Paper Scissors