Variable Scope in Ruby

Scope in programming refers to the areas of your code in which a variable can be accessed. Can your variable be changed within a method and have those changes visible outside of the method? What are local and global variables? How does pass-by-reference or -value play into scope? These questions will be addressed here.

Overwriting, or Duplicating?

Let's first address the question, "What happens if I declare a variable in one place in my code, and then declare it *again* with a *different value* elsewhere? Does the new declaration overwrite the original, or do I get two variables of the same name?"

Remember that in Ruby, all values are actually Objects. So when we say `x = 5`, we are *actually* saying that `x` is a reference to an Object with the value of 5. It's difficult to tell what's happening if we simply put two declarations side by side, and print the outcome:

```
x = 5
x = 6
puts x
```

This code prints "6", but it's difficult to tell if it prints that because the second `x` overwrites the first or if it duplicates it and puts `x` is printing the last handled `x` variable. This is often why it's a good idea to give all your variables unique names!

Thankfully, we can use built in method `object_id` to see if the two Objects are the same, or not!

```
x = 5
puts x.object_id
x = 6
puts x.object_id
```

The output is

```
11
13
```

Those values are obviously different, so we can infer that `x` is referring first to one Object, and then to a different one. We're neither duplicating the variable, nor overwriting the value, but we're changing what the variable references. We can *really* see this with the following code:

```
x = 5
y = x
puts x.object_id
```

```
x = 6
puts x.object_id
puts y.object_id
```

We've added variable `y`, which references the Object `x` originally referred to before we changed it to refer to 6. If we print `y`'s id at the end of the code, we get:

```
11
13
11
```

`y` is pointing to the same object that `x` originally did! This further demonstrates that variable reassignment in Ruby does not affect the value or duplicate a variable: it changes where the variable points to, because variables are actually references.

Scope: Local vs Global

With this in mind, we can discuss the scope of local and global variables in Ruby. In general, a local variable only exists within the immediate area of code: for instance, within a class or method. A global variable however exists throughout the entire program. In Ruby, a variable with no special marker is automatically a local variable, while a variable preceded by a dollar sign is a global variable:

```
x = 0 #Local variable
$x = 0 #$Global variable
```

You can have a global variable and a local variable with the same name, and they will be entirely independent of each other. That dollar sign in front makes a big difference!

So far, we've been using local variables only, and in a local context. A local context includes anything you'd put inside of a "main" function in Java; code that isn't in a method or class. For example, this for loop is all local:

```
x = 3

for a in 1..1
  x = 5
end

puts x
```

In this for loop, we have a local variable `x` being set equal to 3 outside of a for loop, and then equal to 5 inside of a for loop. At the end, we print `x`--and it gives us an output of "5". We can easily change what variable `x` refers to because we are exclusively using `x` in a local context.

Global variables behave the same way in a local context:

```
$x = 3

for a in 1..1
  $x = 5
end

puts $x
```

In the above code, we get an output of “5”.

We can use similar code to demonstrate the fact that a global variable can have the same name as a local one, provided it has that dollar sign out front:

```
$y = 3

for a in 1..1
  y = 5
end

puts $y
puts y
```

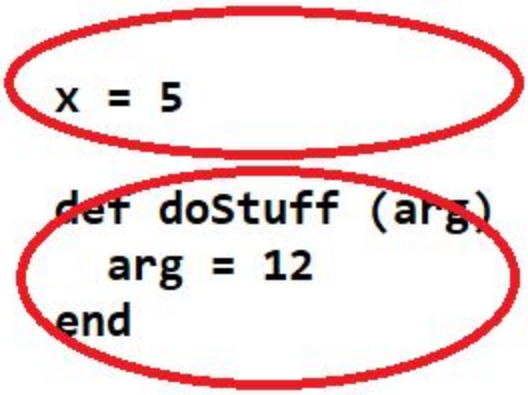
Our output is:

```
3
5
```

In essence: Ruby allows us to declare global variables and local variables easily. They are independent of each other. Global variables can be accessed *anywhere* in a program (they have a wide scope) while local variables are limited to the immediate “area” they’re in (they have a narrow scope).

What about methods?

We’ve talked about local and global variables, and reference reassignment on a local scale, but how about when the “area” differs? The code inside of a method has a different “local” than the code *outside* of it, which means our variables have different reaches.



Two different “locals”.

So, let’s test out variable reassignment with a method in play. Check out this code:

```
a = 10

def doesItChange
  a = 100
end

val = doesItChange
puts val
puts a
```

`a` is a local variable and begins with a value of 10. Then, we run the method doesItChange, which *takes no argument* but *reassigns* `a`. At the end, we 1) set new variable `val` equal to the output of the method, and 2) print `val` and `a`. Here’s the result:

```
100
10
```

This is because we have two *different* variables named `a`. The first one, which equals 10, exists within a different scope than the one which equals 100. Local variables within a method are restricted to that method; the `a` that we use within `doesItChange` doesn’t exist within the main body of the program. We can, however, create a *new* reference, `val`, which points to the same Object that the `a` within the method did, which is why `val` outputs 100.

Let’s check out the difference if we make `a` global:

```
$a = 10

def doesItChange
  $a = 100
end
```

```
val = doesItChange
puts val
puts $a
```

Our new output is:

```
100
100
```

This is because `$a` was declared to be a global variable before we ran the method, and the variable `$a` within the method is *also* global--it is the same variable. Global variables can be accessed anywhere throughout a program; we do not even have to declare an `$a` before the method to call `$a` after it:

```
def doesItChange
  $a = 100
end

puts $a
```

The output is:

```
100
```

As before, local variables are *narrow* in scope, and global variables are *wide*.

Pass-by-Reference or Pass-by-Value

In the last guide, we discussed whether Ruby is pass-by-reference or pass-by-value, and settled on the latter--*mostly*. This is because although we send “values” to our methods, we are doing so in a less direct way: we are sending references to Objects which hold values. As such, Ruby is sometimes referred to pass-by-reference-value. With what we just learned about variable scope in Ruby, we can elaborate on why this is.

Consider the following two methods and the attached code:

```
def uppercase(value)
      value.upcase!
end

def uppercase2(value)
      value = "WILLIAM”
end

name = 'William'
uppercase2(name)
```

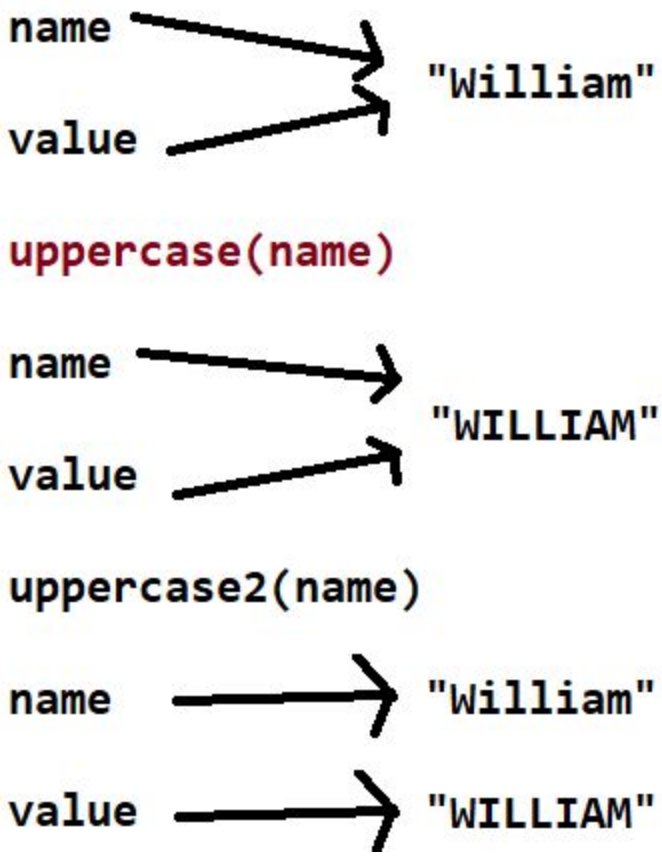
```
puts name
uppercase(name)
puts name
```

The output of this code would be:

```
William
WILLIAM
```

Both methods *look* like they should output “WILLIAM”--one by reassigning `value` to “WILLIAM” and the other by converting `value` to all uppercase characters. But only one method, `uppercase(value)`, gives us “WILLIAM”. The difference is that the first method is *changing the value of the attached Object* and the second is only *reassigning the Object*. This is to say, `value.upcase!` is changing the value of the Object `value` is pointing at to be in uppercase, while `value = “WILLIAM”` is creating a *new* Object for `value` to reference.

Here is a visual, showing the initial state of the variables `name` and `value` and how they change after each method.



`uppercase(value)` *looks* an awful lot like it's pass-by-reference, but that's because while we are, technically, passing the method a reference, it's the value that it looks at. Thus, we call Ruby pass-by-reference-value.

Finally, let's talk about…

Reassignment in Arrays

We've established that everything in Ruby is an Object. But we've still only used Objects that are either Strings or "primitives" in other languages, like Integers. Let's discuss a slightly more complex Object, Arrays, and how we can change the values within them with our variables-that-are-actually-references.

Consider the following code:

```
a = ['c','a','t']
b = ['d','o','g']
a=b
b[1] = 'u'
puts a
puts b
```

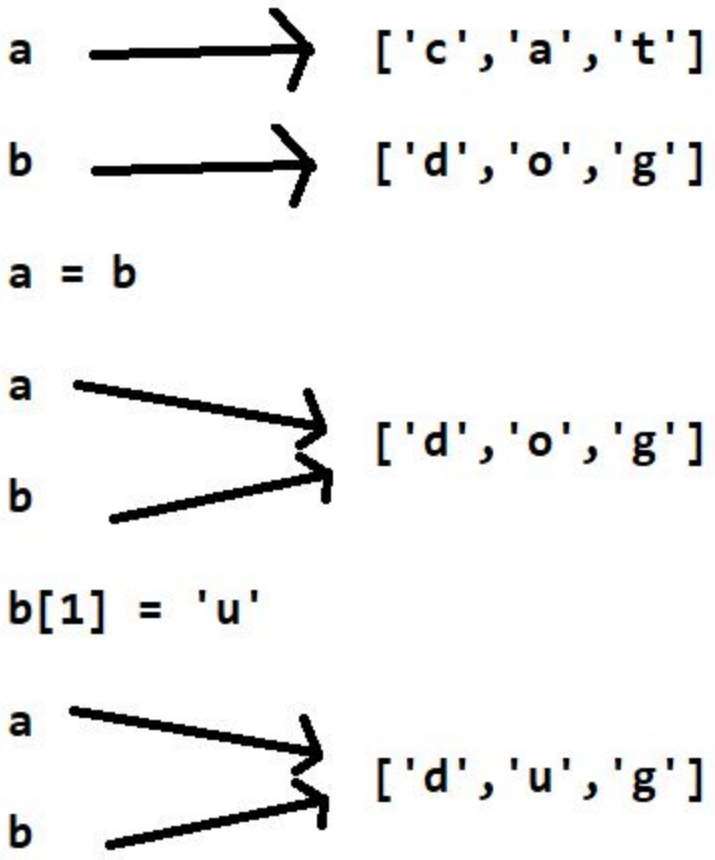
We have two arrays, `a` and `b`, and then we assign `b` to `a` and change the value of an item within b. What do you suppose the output will be?

We get:

```
d
u
g
d
u
g
```

This is because `a` was reassigned to point to the same Object that `b` did. By changing an array element using `b[1] = 'u'` we changed the value of the *Object*; thus, printing both variables/references gives the same result.

Here's another visual of what's happening:



The same exact thing happens if we do `a[1] = ‘u’` instead of `b[1] = ‘u’`; because `a` and `b` are pointing at the same Object.

If you’ve made it this far: thank you! Most of the examples above can be found in the accompanying document, RubyScope.rb. In case you made it this far and didn’t actually read anything, here’s a tl;dr:

1. Local variables have no special signage, but global variables begin with a dollar sign
2. Local variables have a narrow scope, and global variables a wide scope. This means local variables can only be accessed in certain landscapes, while global variables can be accessed anywhere in a program.
3. Ruby variables are actually references. Their values are actually Objects. If we change the value of a variable with an equal sign, we are actually just creating a new Object for the variable to reference.

Sources

launchschool.com https://launchschool.com/blog/object-passing-in-ruby Accessed October 27, 2020.

stackoverflow.com
https://stackoverflow.com/questions/64562581/why-can-you-change-the-value-of-a-local-variable-in-ruby-in-a-function-using-ano Accessed October 27, 2020.