Variable Scope 2

The two major areas where we encounter local variable scoping rules are related to *method definition* and *method invocation with a block*. We'll cover each of these two areas below.

Variables and Blocks

At this point, you've already been using Ruby blocks when you type `do..end` or `{..}` following a method invocation. For example, this is pretty typical Ruby code for iterating through an array and printing out each element.

```
1|  [1, 2, 3].each do |num|
2|      puts num
3|  end
```

The part of the code that we call the *block* is the part following the method invocation:

```
1|  do |num|
2|      puts num
3|  end
```

The `do..end` can be replaced by `{..}`.

```
1|  { |num|
```

```
2|      puts num
3|  }
```

The above code works, but Rubyists prefer to use `do..end` for multi-line blocks, and `{..}` for single line blocks. The blocs following the method invocation is actually an argument being passed into the method. For now, we won't dive deeper into how to use blocks or what they mean.

Instead, we'll focus on one particular attribute of Ruby blocks: they create a new scope for local variables. You can think of the scope created by a block following a method invocation as an *inner scope*. Nested blocks will create nested scopes. A variable's scope is determined by where it is initialized.

Varibales initialized in an outer scope can be accessed in an inner scope, but not vice versa.

Example 1: outer scope variables can be accessed by inner scope

```
1|  a = 1                # outer scope variable
2|
3|  loop do              # the block following the invocati
on of the `loop`
4|      puts a           # => 1
5|      a = a + 1        # "a" is re-assigned to a new valu
e
```

```
6|      break           # necessary to prevent infinite lo
op
7|  end
8|
9|  puts a              # => "a" was re-assigned in the in
ner scope
```

This example demonstrates two things. The first is that inner scope can access outer scope variables. The second, and less intuitive, concept is that you can *change* variables from an inner scope and have that change affect the outer scope. For example, whe we re-assign the variable in the inner scope with `a = a + 1`, that reassignment was visible in the outer scope.

This means that when we instantiate variables in an inner scope, we have to be very careful that we're not accidentally re-assigning an existing variable in an outer scope. This is a big reason for avoiding single-letter variable names.

Example 2: Innter scope variables cannot be accessed in outer scope.

```
1|  loop do                  # the block following the invo
cation of the `loop
2|      b = 1
3|      break
4|  end
5|
```

```
6|  puts b                    # => NameError: undefined loca
l variable or method
```

Here, `main` is the outer scope and does not have a `b` variable. Remember that where a variable is initialized determines its scope. In the above example, `b` is initialized in an inner scope.

Example 3: Peer scopes do not conflict

```
1|  2.times do
2|      a = 'hi'
3|      puts a      # 'hi' this will be printed out twice
due to the loop
4|  end
5|
6|  loop do
7|      puts a      # => NameError: undefined local variab
le or method `a` for main: Object
8|      break
9|  end
10|
11| puts a          # => NameError: undefined local variab
le or method `a` for main: Object
```

Executing the code `puts a` on lines 7 and 11 throws an error because the initial `a = 'hi'` is scoped within the block of code that follows the `times` method invocation. Peer blocks cannot reference variables

initialized in other blocks. This means that we could re-use the variable name `a` in the block of code that follows the `loop` method invocation, if we wanted to.

Example 4: Nested blocks

Nested blocks follow the same rules of inner and outer scoped variables. When dealing with nested blocks, our usage of what's "outer" or "inner" is going to be relative. We'll switch vocabulary and day "first level," "second level", etc.

```
1|  a = 1                # first level variable
2|
3|  loop do              # second level
4|    b = 2
5|
6|      loop do          # third level
7|          c = 3
8|          puts a      # => 1
9|          puts b      # => 2
10|         puts c      # => 3
11|         break
12|     end
13|
14|     puts a          # => 1
15|     puts b          # => 2
16|     puts c          # => NameError
17|     break
```

```
18|  end
19|
20| puts a              # => 1
21| puts b              # => NameError
22| puts c              # => Name Error
```

If any of the outputs above surprises you, make sure to study it carefully and understand the rules around inner scope vs outer scope.

Example 5: variable shadowing
We've been using `loop do..end`, which doesn't take a parameter, but some blocks do take a parameter. Take for example:

```
1|  [1, 2, 3].each do |n|
2|      puts n
3|  end
```

The block is the `do..end`, and the block parameter is captured between the `|` symbols. In the above example, the block parameter is `n`, which represents each element as the `each` method iterates through the array.

But what if we had a variable named `n` in the outer scope? We know that the inner scope has access to the outer scope, so we'd essentially have two local variables in the inner scope with the same name. When that happens, its called *variablle shadowing*, and it prevents access to the outer scope local variable. See this example:

```ruby
n = 10

[1, 2, 3].each do |n|
    puts n
end
```

The `puts n` will use the block parameter `n` and disregard the outer scoped local variable. Variable shadowing also prevents us from making changes to the outer scoped `n`:

```ruby
n = 10

1.times do |n|
    n = 11
end

puts n              # => 10
```

You want to avoid variable shadowing, as it's almost never what you intended to do. Choosing long and descriptive variable names is one simple way to ensure that you don't run into any of these weird scoping issues. And if you do run into these issues, you'll have a much better chance of debugging it if you have clear variable names.

Variables and Method Definitions

If the block following a method invocation has a scope that "leaks", then

a method definition has a scope that is entirely self contained. The only variables a method definition has access to must be passed into the method definition. (Note: we're only talking about local variables for now). A method definition has no notion of "outer" or "inner" scope – you must explicitly pass in any parameters to a method definition.

Example 1: a method definition can't access local variables in another scope

```
1|  a = 'hi'
2|
3|  def some_method
4|      puts a
5|  end
6|
7|  # invoke the method
8|  some_method              # NameError: undefined local v
ariable or method
```

** Example 2:** a method definition can access objects passed in

```
1|  def some_method(a)
2|      puts a
3|  end
4|
5|  some_method(5)           # => 5
```

In the example above, the integer `5` is passed into `some_method` as an argument, assigned to the method parameter, `a` , and thus made available to the method body as a local variable. That's why we can call `puts a` from within the method definition

This is ll you need to know with regards to variable scope and method definitions. We'll talk more about what it means to pass in a parameter to a method definition in a separate assignment. Just remember: local variables that are not initialzed inside a method definition must be passed in as parameters.

Blocks within Method Definitions

Unsurprisingly, the rules of scope for a method invocation with a block remain in full effect even if we're working inside a method definition.

```
1|  def some_method
2|      a = 1
3|      5.times do
4|          puts a
5|          b = 2
6|      end
7|
8|      puts a
9|      puts b
10| end
11|
12| some_method          # => NameError: undefined local va
```

```
riable or method `b`
```

Constants

We're focusing only on local variables for now because you should not be using any other type of variable yet. If you find yourself using global, calss or instance variables, stop. But you may be using constants, which is perfectly ok. Therefore, we should briefly talk about the scoping rules for constants.

Surprisingly, the scoping rules for constants is not the same as local variables. In procedural style programming, constants behave like globals. Here is an example:

```
1|  USERNAME = 'Batman'
2|
3|  def authenticate
4|      puts "Logging in #{USERNAME}"
5|  end
6|
7|  authenticate         # => Logging in Batman
```

That's surprising, considering local variables can't "Leak" into method definitions like that. What about constants and method invocation with a block?

```
1|  FAVORITE_COLOR = 'taupe'
2|
```

```
3|  1.times do
4|      puts "I love #{FAVORITE_COLOR}!"        # => I lov
e taupe!
5|  end
```

Not surprising, since the local variables behave the same. What about initializing a constant in an inner scope?

```
1|  loop do
2|      MY_TEAM = "Phoenix Suns"
3|      break
4|  end
5|
6|  puts MY_TEAM            # => Phoenix Suns
```

That is very different from local variables.

Constants are said to have *lexical scope,* which will have more meaningful consequences when we get to object oriented programming. For now, just rememner that constants have different scoping rules from local vaiables.

Summary

Understanding variable sope is one of the most challenging and important aspects to learning to program. Make sure you know how local variable scope works with regards to a method definition and a method invocation with a block. Play around with various scenarios until

you feel comfortable. It’s likely you’ll forget these rules, but the most important thing is to be able to quickly jump in irb or open up your editor and refresh your memory.

Remember that differen variable types will have different scoping rules, and the only other variable type you should run into right now besides local variables are constants.