- My Assessments
- Sign Out

- Courses
- RB101 Programming Foundations
- Lesson 2: Small Programs
- Variable Scope

Give us your feedback
×

Suggestions, errors or compliments on this page - let us know!

Select a category
Please don't use this Feedback form to ask questions. If you have a question about the content, use the lesson forums and one of our staff will take a look. If you have a general non-technical question, send us an email at support@launchschool.com.
Your comment Enter comment
Edit Send feedback
Preview Send feedback

Variable Scope

One of the trickiest things to understand for a beginner is the concept of *scope* and how it pertains to accessing variables. In Ruby, there are several different types of variables, each with their own scoping rules. In this assignment, we'll be talking only about **local variables**.

First, refresh your memory a bit by re-reading this section from the Introduction to Programming book: Variable Scope

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

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

```
do |num|
  puts num
end
```

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

```
{ |num|
  puts num
}
```

The above code works, but Rubyists prefer to use `do..end` for multi-line blocks, and `{..}` for single line blocks. The block 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.

> Variables 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

```
a = 1         # outer scope variable

loop do       # the block following the invocation of the `loop` method creates an inner scope
  puts a      # => 1
  a = a + 1   # "a" is re-assigned to a new value
  break       # necessary to prevent infinite loop
end

puts a        # => 2  "a" was re-assigned in the inner 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, when we re-assigned 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: inner scope variables cannot be accessed in outer scope

```
loop do       # the block following the invocation of the `loop` method creates an inner scope
  b = 1
  break
end

puts b        # => NameError: undefined local variable or method `b' for main:Object
```

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

```
2.times do
  a = 'hi'
  puts a      # 'hi' <= this will be printed out twice due to the loop
end

loop do
  puts a      # => NameError: undefined local variable or method `a' for main:Object
  break
end
```

```
puts a        # => NameError: undefined local variable 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 say "first level", "second level", etc.

```
a = 1           # first level variable

loop do         # second level
  b = 2

  loop do       # third level
    c = 3
    puts a      # => 1
    puts b      # => 2
    puts c      # => 3
    break
  end

  puts a        # => 1
  puts b        # => 2
  puts c        # => NameError
  break
end

puts a          # => 1
puts b          # => NameError
puts c          # => NameError
```

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, 2, 3].each do |n|
  puts n
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, it's called *variable shadowing*, and it prevents access to the outer scope local variable. See this example:

```
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`:

```
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

While a block (doesn't matter whether it follows a method invocation) has a scope that "leaks", a method's scope 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

```
a = 'hi'

def some_method
  puts a
end

# invoke the method
some_method     # => NameError: undefined local variable or method `a' for main:Object
```

Example 2: a method definition can access objects passed in

```
def some_method(a)
  puts a
end

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 all you need to know with regards to variable scope and method definitions. We'll talk more about what it means to pass an argument to a method definition in a separate assignment. Just remember: local variables that are not initialized *inside* a method definition must be defined 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.

```
def some_method
  a = 1
  5.times do
```