More Variable Scope

In the previous assignment we talked about local variable scope as it relates to *method definition* and *method invocation with a block*. We briefly mentioned that when you have method invocation with a block, the block is very closely tied to the method invocation. In this assignment we want to delve into those topics at a deeper level. Our goal here is to accomplish two things:

1. Build on the definitions outlined in the previous assignment
2. Provide a slightly more complex but more accurate mental model of methods, blocks, how they inter-relate, and how local variable scope fits into the picture

Two key terms in building this mental model are **method definition** and **method invocation**. You have come across these two terms already in this course, but below is a brief outline just to recap a little.

Method definition is when, within our code, we define a Ruby method using the `def` keyword.

```
def greeting
  puts "Hello"
end
```

Method invocation is when we call a method, whether that happens to be an existing method from the Ruby Core API or core Library, or a custom method that we've defined ourselves using the `def` keyword.

```
greeting # Calling the greeting method outputs "Hello"
```

You should by this point be familiar with code like that shown in the above two examples.

You will also have seen many examples of methods being called with blocks.

```
[1, 2, 3].each { |num| puts num }
```

Technically any method can be called with a block, but the block is only executed if the method is defined in a particular way. The precise mechanics of how methods interact with blocks are covered in detail in a later course, so don't worry too much about that right now.

What we do want you to be clear on at this stage however, is that a block is *part of* the method invocation. In fact, *method invocation* followed by curly braces or `do..end` is the way in which we *define* a block in Ruby.

Essentially the block acts as an argument to the method. In the same way that a local variable can be passed as an argument to a method at invocation, when a method is called with a block it acts as an argument to that method.

The way that an argument is used, whether it is a method parameter or a block, depends on how the method is defined. Let's look at a few examples.

Example 1: method parameter not used

```
def greetings(str)
  puts "Goodbye"
end

word = "Hello"

greetings(word)

# Outputs 'Goodbye'
```

Example 2: method parameter used

```
def greetings(str)
  puts str
  puts "Goodbye"
end

word = "Hello"

greetings(word)

# Outputs 'Hello'
# Outputs 'Goodbye'
```

In both of the above examples, the method definition is such that the method has a parameter `str`. This allows the method to access the string `"Hello"` since it is passed in as an argument at method invocation in the form of the local variable `word`. Example 1 doesn't do anything with this string, Example 2 outputs it.

The kind of code in Examples 1 & 2, with a local variable being passed to a method as an argument, should already be fairly familiar to you by now; the way in which methods interact with blocks at invocation may be less familiar.

Example 3: block not executed

```
def greetings
  puts "Goodbye"
end

word = "Hello"

greetings do
  puts word
end
```

```
# Outputs 'Goodbye'
```

Example 4: block executed

```
def greetings
  yield
  puts "Goodbye"
end

word = "Hello"

greetings do
  puts word
end

# Outputs 'Hello'
# Outputs 'Goodbye'
```

In Example 3 the `greetings` method is invoked with a block, but the method is not defined to use a block in any way and so the block is not executed.

In Example 4 the `yield` keyword is what controls the interaction with the block, in this case it executes the block once. Since the block has access to the local variable `word`, `Hello` is output when the block is executed. Don't focus here on what `yield` is or how it works; writing methods that take blocks is explored at depth in a later course. The important take-away for now is that blocks and methods can interact with each other; the level of that interaction is set by the method definition and then used at method invocation.

When invoking a method with a block, we aren't just limited to executing code within the block; depending on the method definition, the method can use the *return value* of the block to perform some other action. Consider the following example:

```
a = "hello"

[1, 2, 3].map { |num| a } # => ["hello", "hello", "hello"]
```

The `Array#map` method is defined in such a way that it uses the return value of the block to perform transformation on each element in an array. In the above example, `#map` doesn't have direct access to `a` but it can use the value of `a` to perform transformation on the array since the block *can* access `a` and returns it to `#map`.

Let's review what we discussed in the previous assignment regarding how a method definition accesses local variables compared to how a method invocation with a block accesses local variables. Method definitions *cannot* directly access local variables initialized outside of the method definition, nor can local variables initialized outside of the method definition be reassigned from within it. A block *can* access local variables initialized outside of the block and can reassign those variables. We already know that methods can access local variables passed in as arguments, and now we have seen that methods can access local variables through interaction with blocks.

Given this additional context, we can think of **method definition** as setting a certain scope for any local variables in terms of the parameters that the method definition has, what it does with those parameters, and also how it interacts (if at all) with a block. We can then think of **method invocation** as using the scope set by the method definition. If the method is defined to use a block, then the scope of the block can provide additional flexibility in terms of how the method invocation interacts with its surroundings.

Summary

The purpose of this assignment was to provide some additional insight and a more accurate mental model regarding methods, blocks and local variable scope. The interaction between methods and blocks is a deep topic and is covered in much more detail in a later course; for now you just need to understand a few key points:

- The `def..end` construction in Ruby is method definition
- Referencing a method name, either of an existing method or subsequent to definition, is method invocation
- Method invocation followed by `{..}` or `do..end` defines a block; the block is *part of* the method invocation
- Method definition *sets* a scope for local variables in terms of parameters and interaction with blocks
- Method invocation *uses* the scope set by the method definition

You marked this topic or exercise as completed.
Variable Scope
Pass by Reference vs Pass by Value

Formatting Help

Normal Markdown & GFM

Github Flavored Markdown

****bold****

~~ ~~strike through~~ ~~

```
`single line of code`
```

Supports Emoji

Code Blocks

You can write code blocks using three backticks (```) followed by the language you want to show. For instance, if you wanted to write a Ruby code block you would write:

~~~
```ruby
class Foo
end
```
~~~

```
class Foo
end
```

You can also leave out the language identifier, doing this will make it so your code isn't highlighted at all, and all text will be white with no coloring.

~~~
```
class Bar
end
```
~~~

```
class Bar
end
```

Highlighting

You may highlight lines within your code blocks. This is a different type of highlighting, and isn't related to how the code within your code block may be colored depending on the language identifier used.
To highlight one or more lines of code, add `# highlight` on a separate line just before the first line you wish to highlight. To end highlighting, add `# endhighlight` on a separate line immediately after the last line you wish to highlight.

~~~
```ruby
def say_hello_world
# highlight
  puts 'Hello World!!!'
# endhighlight
end
```
~~~

```ruby
def say_hello_world
  puts 'Hello World!!!'
end
```

×