**Scope**

If you've made it this far in the course, you've been unwittingly utilizing the concept of scope in your code. Let's take a minute explore the fundamentals of **scope** and define some rules! Scope describes the location in your program where a variable is accessible.

**Methods and Local Scope**

Let's explore **lexical scope**. Lexical scope describes how a variable name evaluates in nested code. In other words, how variable names resolve if we put them in structures like methods, conditionals, or blocks?

To start, let's say we had some duh-duh obvious code:

def say\_hello

message = "hello"

p message

end

say\_hello

Above we can say that the say\_hello method has its own scope where the variable message is defined. This means we are free to reference and print that variable in that same scope. The say\_hello method is a *local scope*. This will be true for any method.

This means that if we are outside of the say\_hello method/scope, we **cannot** reference the message variable:

def say\_hello

message = "hello"

end

say\_hello

p message # NameError: undefined local variable

The code above will fail with an error because we are referencing a variable, message, but it was not defined in that scope. We can imagine there being two scopes in that example. There is the sayHello method scope and the surrounding scope that is on the outside of sayHello.

Let's try the opposite scenario. What if we had defined a variable in the surrounding scope and tried to reference it within the say\_hello scope:

message = "hi"

def say\_hello

p message # NameError: undefined local variable

end

say\_hello

Above we suffer the same issue. The say\_hello scope does not have access to the surrounding scope. From our previous example it seems that a scope is determined by a method. Each method will have its own local scope. This is going to be our rule of thumb. Technically we consider code out in the open as another local scope, although it's not a method. It's common for new ruby programmers to think that variables defined out in the open will automatically be accessible in the global scope, but this is not true.

**Global Variables**

Okay, so methods are the primary areas for scope, but are there any others? Everywhere area in our code can access the **global scope**. To define variables in the global scope, we must use special syntax. Let's redo our last example utilizing global scope:

$message = "hello globe"

def say\_hello

p $message

end

say\_hello # => "hello globe"

This code will correctly print "hello globe". Creating a global variable is simple, put a $ at the beginning of the name. Here's a similar example, this time creating the global $message inside of say\_hello but referencing it from the outside:

def say\_hello

$message = "hello globe"

end

say\_hello

p $message # => "hello globe"

Ruby automatically defines some global variables for us to reference. For example, $PROGRAM\_NAME will be a string describing the name of our program. Later in the course we'll use other global names like $stdin and $stdout handle user input and output. For now, just understand that we can reference a $variable\_name anywhere in our program because it is global!

**Constants**

Let's take a quick detour to explore an additional way to create a variable, using a constant. A constant is denoted syntactically by beginning the name with a capital letter. By convention we like to make the entire name capital to emphasize it being a special constant.

A constant variable cannot be reassigned:

FOOD = "pho"

p FOOD # => "pho"

FOOD = "ramen" #warning: already initialized constant FOOD

#warning: previous definition of FOOD was here

You'll receive a warning when reassigning a constant. Reassignment means using = on that name again. A common point of confusion is that while you cannot reassign a constant, *you can still mutate that constant name* without warning:

FOOD = "pho"

FOOD[0] = "P"

p FOOD # => "Pho

Note that above we did not reassign to the FOOD name, instead we assign to an index of the FOOD string. The key takeaway is that constants can still mutate or be "changed", they just can't be reassigned.

Why bring up constants in our chatter about scope? Because constants will exist in global scope! We can do VARIABLE or $variable to ensure global scoping:

FOOD = "pho"

$drink = "ice coffee"

def my\_favorite

p FOOD

p $drink

end

my\_favorite

The code above works because both variables are accessible in global scope. This means they can be referenced anywhere in our code! However, we should use the global scope sparingly. We want to minimize our methods all referencing data that is outside of them, as manipulations to the data will be hard to track. Instead we should write methods that accept data as arguments as it is more explicit where the data is coming from.

**What does not have it's own scope?**

So methods and the global scope will be our primary structures that provide scope. Now let's highlight what structures don't have their own scope.

Blocks don't have their own scope, they are really a part of the containing method's scope. Below, the times block can reference message.

def say\_bye

message = "bye"

3.times do

p message

end

end

say\_bye

Other structures like conditionals or while loops also don't have their own scope, they are really part of the containing scope. Although a variable is defined within the if statement below, it is still accessible outside of the if statement, because if's don't have their own scope.

if true

drink = "cortado"

end

p drink