# Variable Scope 1

A variable’s scope determines where in a program a variable is available for use. A variable’s scope is defined by where the variable is initialized or created. In Ruby, variable scope is defined by a block. A block is a piece of code following a method invocation, usually delimited by either curly braces `{}` or `do/end`. Be aware that not all `do/end` pairs imply a block*.

Remember this rule:

**Inner scope can access variables initialized in an outer scope, but not vice versa.**

Looking at some code will make this clearer. Let’s say we have a file called `scope.rb`

# `scope.rb`

```ruby
a = 5                   # variable is initialized in the outer scope

3.times do |n|          # method invocation with a block
a = 3                   # is a accessible here, in an inner scope?
end
```

```

  puts a
```

What is the value of `a` when it is printed to the screen? Try it out.

The value of `a` is 3. This is because `a` is available to the inner scope created by `3.times do … end`, which allowed the code to re-assign the value of `a`. In fact, it re-assigned it three times to 3. Let's try something else. We'll modify the same piece of code.

# `scope.rb`

```ruby
a = 5
3.times do |n| # method invocation with a block
    a = 3
    b = 5        # b is initialized in the inner scope
end
puts a
puts b       # is b accesible here, in the outer scope?
```

What result did you get when running the program? You should have gotten an error to the tune of:

```
scope.rb:11:in `<main>': undefined local variable or metho
d `b' for main:Object
(NameError)
```

This is because the varibale `b` is not available outside of the method invocation with a block where it is initialized. When we call `puts b` it is not available within that outer scope.

*Note: the key distinguishing factor for deciding whether code delimited by `{}` or `do/end` is considered a block (and thereby creates a new scope for variables), is seeing if the `{}` or `do/end` immediatley follows a method invocation. For example:

```
arr = [1, 2, 3]
for i in arr do
    a = 5                # a is initialized here
end
puts a                   # is it accessible here?
```

The answer is yes. The reason is because the `for...do/end` code did **not** create a new inner scope, since `for` is part of Ruby language and not a method invocation. When we use `each`, `times` and other method invocatios, followed by `{}` or `do/end`, that's when a new block is created.

## Types of Variables

Local variables are the most common variables you will come across and obey all scope boundaries. These variables are declared by starting the variable name with neither `$` nor `@`, as well as not capitalizing the entire variable name.

Example of a local variable declaration:

```ruby
var = 'I must be passed around to cross scope boundaries.'
```

## Exercise

Look at the following programs:

```ruby
x = 0
3.times do
    x += 1
end
puts x
```

and...

```ruby
y = 0
3.times do
    y += 1
    x = y
end
puts x
```

What does x print to the screen in each case? Do they both give erros? Are the errors different? Why?

## Solution

The first solution prints `3` to the screen. The second throws an error `undefined local variable or method` because `x` is not available as it is created within the scope of the `do/end` block.

## Video Walk through Explanation

Looking at the first piece of code, on line one a local variable `x` is initialized and it references the integer `0`. We then invoke the `times` method on the integer `3` and we've passed a block to this `times` method. Ruby is going to iterate through this block 3 times. The code on line 3 is shorthand for re-assignment. We are re-assigning `x` to the value of x + 1. So on our first iteration thorugh this block, x will be reassigned to 1, and then 2 and then 3. Then Ruby stops iterating through this block and our program exectuion reaches line 5. On line 5 we invoce the puts method to output x. Because we know we have iterated through this block 3 times, incrementing the integer referenced by `x`, we expect that `x` now references the integer `3`.

So lets take a look at the second piece of code we were given. We have two variables, `y` and `x`. `y` is initialized on line one and it references `0`. Again we invoke the `times` method on the integer `3` and pass it a block. Within the block we are going to re-assign `y` to `y + 1` each time we iterate through. Then on line 4 we initialize a local varibale x and we point it at the same value referenced by y. So as we iterate through this block three times, `y` will be reassigned to `1`, `2`, and `3`. When the code on line 4 runs, `x` will also be assinged to those integers. When program execution reaches line 6 and we output `x` you may expect to see `3`. However there is a problem here because unlike `y`, `x` was initialized within the block. Blocks create an inner scope. Variables initialized within an inner scope are not available within an outer scope. On line 6 we expect that ruby will raise an error. `undefined local varibale or method`.