Class Variables

Class variables have a class wide scope, they can be declared anywhere in the class. A variable will be considered a class variable when prefixed with `@@`

```
class BMW
    @@classification = "BMW Automobiles"

    def self.classification
        @@classification
    end

    def classification
        @@classification
    end
end

dino = BMW.new
dino.classification
# => "BMW Automobiles"

BMW.classification
# => "BMW Automobiles"
```

Class variables are shared between related classes and can be overwritten from a child class

```
class BMWModified < BMW
    @@classification = "BMWModified Automobiles"
end

BMWModified.classification
# => "BMWModified Automobiles"

BMW.classification
# => "BMWModified Automobiles"
```

This behaviour is unwanted most of the time and can be circumvented by using class-level instance variables.

Class variables defined inside a module will not overwrite their including classes class variables:

```
module BMWModules
    @@classification = "Module"
end

class BMWCLass < BMW
    include BMWModules
End

BMWCLass.class_variables
# => [:@@classification]
BMWModules.class_variables
# => [:@@classification]

BMWCLass.classification
 => "BMWModified Automobiles"
```

Global Variables

Global variables have a global scope and hence, can be used everywhere. Their scope is not dependent on where they are defined. A variable will be considered global, when prefixed with a `$` sign.

```
$global = "global"

class Test
    def instance_method
       p "#{$global} vars can be used #{$another_global_var}"
    end

    def self.class_method
       $another_global_var = "everywhere"
       p "#{$global} vars can be used #{$another_global_var}."
```

```
    end
end

Test.class_method
"global vars can be used everywhere."
 => "global vars can be used everywhere."


test = Test.new
test.instance_method
"global vars can be used everywhere"
 => "global vars can be used everywhere"
```

Instance Variables

Instance variables have an object wide scope, they can be declared anywhere in the object, however an instance variable declared on class level, will only be visible in the class object. A variable will be considered an instance variable when prefixed with @. Instance variables are used to set and get an object's attributes and will return nil if not defined.

```
class Test
    @base_object = "raw"#shared among various object of classes as
this is available to class object

    def initialize(instance_object = nil)
        @instance_object = instance_object || self.class.base_object
    end

    def get_instance_object
        @instance_object
    end

    def get_class_object
        @base_object
    end
```

```
    def self.base_object
        @base_object
    end
end

test = Test.new
test.get_instance_object
 => "raw"
Test.base_object
 => "raw"
test.get_class_object # available to class object
 => nil

test1 = Test.new "gaw"
test1.get_instance_object
 => "gaw"
Test.base_object
 => "raw"
test1.get_class_object
=> nil
```

Local Variables

Local variables (unlike the other variable classes) do not have any prefix.

Its scope is dependent on where it has been declared, it can not be used outside the "declaration containers" scope. For example, if a local variable is declared in a method, it can only be used inside that method.

```
def method

    method_scope_var = "hi are you there"

    p method_scope_var

end
```

```
method
```

```
"hi are you there"
```

```
 => "hi are you there"
```

```
method_scope_var
```

```
NameError: undefined local variable or method `method_scope_var' for
main:Object
```

```
      from (irb):68
```

As soon as you declare a variable inside a `do ... end` block or wrapped in curly braces `{}` it will be local and scoped to the block it has been declared in.

```
2.times do |n|
    local_var = n + 1
    p local_var
end
# 1
# 2
# => 2
```

```
local_var
# NameError: undefined local variable or method `local_var'
```

However, local variables declared in if or case blocks can be used in the parent-scope:

```
if true
    usable = "yay"
end

p usable
# yay
# => "yay"
```

The variables used for block arguments are local to the block, but will overshadow previously defined variables, without overwriting them.

```
overshadowed = "sunlight"

["darkness"].each do |overshadowed|
    p overshadowed
end
# darkness
# => ["darkness"]

p overshadowed
# "sunlight"
# => "sunlight"

my_variable = "foo"
```

```ruby
my_variable.split("").each_with_index do |char, i|
    puts "The character in string '#{my_variable}' at index #{i} is #{char}"
end
# The character in string 'foo' at index 0 is f
# The character in string 'foo' at index 1 is o
# The character in string 'foo' at index 2 is o
# => ["f", "o", "o"]


def some_method
    puts "you can't use the local variable in here, see? #{my_variable}"
end


some_method
# NameError: undefined local variable or method `my_variable'
```