Ruby trunk - Feature #6806

Support functional programming: forbid instance/class variables for ModuleName::method_name, allow for ModuleName.method_name

07/28/2012 10:26 PM - alexeymuranov (Alexey Muranov)

Status:	Feedback	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:	Next Major	

Description

What would you say about this proposal? Is there a better alternative?

I suggest to support functional programming in Ruby by making module methods called with ModuleName::method_name syntax raise an exception if the method uses instance or class variables (instance variables of the singleton class, of course).
If i understand correctly, currently ModuleName::method_name and ModuleName.method_name behave identically, so i propose that they be different:

```
module M
  module_function
    def f(x)
      x*x
    end
    def g(x)
      @x ||= x
      @x*@x
    end
end

M.f(2) # => 4
M.g(2) # => 4
M::f(3) # => 9
M::g(3) # => Error: instance variable `@x` used in a functional call `M::g`
```

Current behavior:

```
M.f(2) # => 4
M.g(2) # => 4
M::f(3) # => 9
M::g(3) # => 4
```

History

#1 - 07/28/2012 10:37 PM - alexeymuranov (Alexey Muranov)

I think that if you tell a new person that M::f and M.f are the same in Ruby, they won't believe at first. So why keeping them the same?

I do not know how to be with the use of global variables in M::f. I imagine there are other details to discuss.

This feature wouldn't ensure truly functional programming, but i hope it can introduce a "good practice" of making it clear from the call syntax if a method can change the state of the object. In other words, i propose that when x::f(y) is called, the state of the object x be frozen, and unfrozen after the return from f, so x act simply as a namespace.

I do not know whether it would be better for x::f to allow read-only access to the state of x, or no access at all. Probably read-only access makes better sense, given impossibility to exclude interaction with other data anyway.

Another approach could be to forbid a function called with :: prefix to change the state of any object that is not created by it, and to forbid to read the state of any object that is not accessible through the state of self or the state of one of the arguments.

If the feature is implemented with read-only access to any existing data, this syntax can be used to give a precise meaning to non-exclaiming methods: a::map instead of a.map, and a.map! stays as is.

To determine what objects are "accessible" through the state of a given object, i think it should be forbidden to go "up" with methods like "#class" or "#superclass" and then go "down". In the direction "up", only the object's class and its ancestors should be accessible, i think.

#2 - 07/31/2012 04:32 AM - drbrain (Eric Hodel)

- Assignee set to matz (Yukihiro Matsumoto)

- Target version set to Next Major

This would break compatibility with much ruby code.

#3 - 06/30/2014 01:31 PM - alexeymuranov (Alexey Muranov)

Besides functional programming, IMO this would support [command–query separation](https://en.wikipedia.org/wiki/Command–query_separation).

#4 - 06/30/2014 01:37 PM - nobu (Nobuyoshi Nakada)

- Description updated

- Status changed from Open to Feedback

It seems unrelated to "functional programming" at all.

#5 - 06/30/2014 01:45 PM - alexeymuranov (Alexey Muranov)

I meant that a function called like Math::sin would be required to return same values (for same arguments) every time. Maybe i did not explain this well. Foo.bar, on the other hand, would not have this restriction.

Same could be generalized to mutable (non-constant) objects, but there the distinction could be harder to make. Maybe x::to_s but x.update!?

P.S. Thanks for reformatting the proposal.

#6 - 06/30/2014 02:02 PM - nobu (Nobuyoshi Nakada)

It's just your style.
I use Math.sin and so on.