Add a method to alias class methods

There doesn't seem to be an intuitive way to alias class methods. Perhaps we can add a method such as

```ruby
alias_class_method :new_name, :old_name
```

Something like this would work:

```ruby
module Kernel
  def alias_singleton_method new_name, old_name
    singleton_class.class_exec { alias_method new_name, old_name }
  end
end

class Foo
  def self.bar :bar
    end
    alias_singleton_method :baz, :bar
  end
end
```

Having to nest the logic within class << self makes it difficult to understand what is going on, and the code less readable.
JustJosh (Joshua Stowers) wrote:

Thanks Matthew, that result is exactly what I had in mind.

Given that it's very easy (as shown by Matthew) to create such a method, do you think it's necessary that this be implemented by Ruby itself? You haven't yet answered the question about frequency of use or use cases.

shevegen (Robert A. Heiler) wrote:

Given that it's very easy (as shown by Matthew) to create such a method, do you think it's necessary that this be implemented by Ruby itself? You haven't yet answered the question about frequency of use or use cases.

Well I do not have a statistical dataset myself, but I use it all the time.

The thing is that for me it is very natural to use "alias".

def foo; puts 'hi from foo(); end
alias bar foo

I love it. I love aliases. They make me happy. :)

I use aliases mostly because I have a bad memory. And also because I want to remain flexible. Some time ago I started to adopt a more "logical" scheme with my classes, when it may make sense. For example, most of my classes, if necessary, have a "reset" method. Often I also have a "menu" method, which I consider as the interface that can parse the commandline (or optionally any other input that is sent to it).

And so on and so forth.

So in this context, I totally agree with Joshua Stowers.

I think that ruby itself should not care too much if a user wants to use an alias on the class/module level instance or within the context of self with regular instance methods of the class.

So here, I agree with Joshua.

The awkward thing, though, is that I actually dislike the syntax proposal:

```
alias_class_method :new_name, :old_name
```

The reason is, and this may be trivial, is that I really really hate the `:` character there. The symbols are ok although it's better to avoid them.

I also understand that alias_method is not the same as alias, but alias is so cute and short, it is just built to be loved!

Now you may wonder, how do I alias class methods then?

I use a VERY clumsy way. I do not recommend anyone to use it and I am sure there are better ways but here goes:

```
self.instance_eval { alias stop_codons stop_codons? }
```

I dislike the self.instance_eval part because it is so long and verbose - but I love the alias part within the {} because to my eyes, it is neat and cuddly.

I can't say whether I would use alias_class_method - the name is not soooo bad (though, what if we have a module Foo; end method? Do we call it alias_module_method then?), but I am not sure about the syntax.
BUT I also dislike typing the:

```ruby
self.instance_eval { }
```

part altogether, so in this context, I agree with Joshua Stowers.

(I also understand that one should not use too many aliases but I love aliases. The main method is usually the one I use the most, and then I may use some aliases, some for backwards compatibility and sometimes I remove them too at a later point so it just provides some more flexibility "as you go").

Anything that would be as neat or almost as short as:

```
"alias foo bar"
```

but for class-methods / module-methods would be great!

#7 - 05/21/2017 05:33 PM - shevegen (Robert A. Heiler)

Martin showed this example:

```ruby
class Array
  class << self
    alias :my_new :new
  end
end
```

Ruby allows this flexibility, this is true. This is also great, we love it.

But when you use this code and want to distribute it, it is more cumbersome. And not everyone likes to have modifications that are non-standard ruby in their code.

I understand that the ruby team wants to be conservative and not proliferate with lots of extra methods that have only a special case (say active* gems), but to me I also completely understand Joshua Stowers here - if possible it is MUCH, much better when it would be in main ruby. So that everyone could use it.

- I may use it if it is in ruby core/stdlib.
- I may use duck patching (sometimes people use the monkey rather than the duck) for my local code.
- But it is VERY unlikely that I would modify a core class of ruby like this AND distribute my code to other people too. I just consider it too cumbersome (and refinements, while the idea is ok, I never really liked the syntax... and it felt awkward... but this is for another discussion, lots of time towards ruby 3.x I hope).

#8 - 06/14/2017 03:08 AM - JustJosh (Joshua Stowers)

I think Robert is exactly right. I've hoped for such a method on several occasions myself, but highly dislike cluttering up the codebase with logic that is difficult to understand. One of the best things about Ruby is that when written well it can read almost like English. Such a method would help apply that further.

#9 - 09/25/2017 08:13 AM - matz (Yukihiro Matsumoto)

- Status changed from Feedback to Rejected

As Martin-sensei pointed out, use singleton class notation.

```ruby
class Foo
  class << Foo
    def foo; end
    alias bar foo
  end
end
```

I don't think it's worth adding a new method to avoid this simple thing.

Matz.
shevegen (Robert A. Heiler) wrote:

Martin showed this example:

```ruby
class Array
  class << self
    alias :my_new :new
  end

Ruby allows this flexibility, this is true. This is also great, we love it.

But when you use this code and want to distribute it, it is more cumbersome. And not everyone likes to have modifications that are non-standard ruby in their code.

Sorry, I should have used another example than Array. This issue is about aliasing class methods, and aliasing a method in a built-in class would be a problem whether it's done with a new feature or with the way I showed. So the fact that I used Array is confusing, but orthogonal to the issue at hand.