Ruby - Feature #13927

Integrate module_function as a core language type

09/21/2017 07:57 PM - rocifier (Ryan O'Connor)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Using ruby commercially we have discovered that modules are no longer serving their original intended purpose of mixins. Another usage of the module has been shoehorned by using module_function. We have found that when developers use module_function they are converting the module into a new concept, and no longer intend their module to be used as a mixin for classes. Instead the intent of this new type of module is to provide publically accessible class methods withoutthe need for instantiation, and without polluting the global namespace. This usage is becoming potentially more common than the mixin usage.

I propose that we integrate the module_function pattern as its own core ruby object along with Module and Class.

History

#1 - 09/22/2017 05:21 AM - matz (Yukihiro Matsumoto)

I admit modules with module_function play different role from ordinary modules, and so do refinements.
I am not sure how much gain we would have by making them their own objects. Adding new syntax, keywords, or concepts may make the language slightly more understandable, but may introduce incompatibility, that everyone hates.

Matz.

#2 - 09/22/2017 10:18 AM - rocifier (Ryan O'Connor)

matz (Yukihiro Matsumoto) wrote:

> I admit modules with module_function play different role from ordinary modules, and so do refinements.
> I am not sure how much gain we would have by making them their own objects. Adding new syntax, keywords, or concepts may make the language slightly more understandable, but may introduce incompatibility, that everyone hates.
>
> Matz.

Hi Matz thanks for your reply. I think we would have a reasonably significant gain in making the language easier to pick up for newcomers, as well as allowing existing developers to opt-in to creating more concise and readable code.

In terms of compatibility, I suppose we would need to keep backwards compatibility in a feature like this. The downside I can think of is if people need to downgrade the version of ruby they are using, which admittedly does happen from time to time. My only answer to this is that it could be an automated rubocop fix (as an example) to easily go from one syntax to the other.

#3 - 09/23/2017 09:20 PM - shevegen (Robert A. Heiler)

I wanted to write a reply but it was too long, so here just a shorter one.

Even new syntax may make a language more complex but also simpler at the same time.
People may have to learn/understand what a concept means in a language. New features
may provide more flexibility but often make a language also more complex. A good example
are **refinements** - they are not so difficult to understand but there is some syntax
(how to activate a refinement) and people may have to understand how refinements
work and whether they are limited. I think there was some presentation a while ago
about "nobody uses refinements" or something like that, which reminds me of "nobody
knows nobu", until the epic storyline was revealed where Nobu arose rom Mt. Fujimoto. :)

Anyway, not going to write too much, but when I was writing my longer reply, I actually
wondered whether this change targeted ruby 2.x or ruby 3.x.

Rocifier, can you say which ruby you have had in mind primarily?

If ruby 3.x, I guess it may be possible to also release "future" ruby builds more frequently
that are pre-ruby 3.x, with more experimental changes so that people can test them extensively
well before ruby 3.x will be finalized. Not just in regards to the module_function situation
(I am neutral on this) but in general. This may also allow people to add new features into
ruby - like developer A is very fast, developer B is slower and the code has to be more
complex, so other ruby hackers could test code from developer A more frequently in these

ruby pre-3.x builds.

If ruby 2.x then this is much harder. Not everyone wants to be required to have to adjust syntax code all of a sudden and not everyone knows how to use rubocop either or wants to be required to have to use it.

#4 - 09/25/2017 07:23 AM - duerst (Martin Dürst)

rocifier (Ryan O'Connor) wrote:

> Using ruby commercially we have discovered that modules are no longer serving their original intended purpose of mixins. Another usage of the module has been shoehorned by using module_function. We have found that when developers use module_function they are converting the module into a new concept, and no longer intend their module to be used as a mixin for classes. Instead the intent of this new type of module is to provide publically accessible class methods without the need for instantiation, and without polluting the global namespace. This usage is becoming potentially more common than the mixin usage.

As far as I understand (and found in many books and blogs,...), the 'classic' explanation of Module/module is that it serves two purposes:

1. Mixin (e.g. Enumerable), and 2) Namespacing (e.g. Net::HTTP). This is also how Matz explained it last week at his keynote at Ruby Kaigi 2017 (see http://rubykaigi.org/2017/presentations/yukihiro_matz.html).

Based on this understanding, are you saying that there's a third purpose? Or are you thinking about module_function as part of, or a variant of, the second purpose? In both cases, why do you think that your purpose should be split out syntactically, while the 'classic' two purposes stay with the same syntactic device? Or if you have another understanding, then please explain.

Also, the most well known module_function example is probably Math.sin and friends. However, as Ruby is an object-oriented language, my personal preference would be to simply write 0.3.sin rather than Math.sin 0.3. I'd expect that commercial applications also would prefer object-oriented architecture and syntax to (namespaced) global functions. What do you think is the reason that you observe extended use of module_function?

> I propose that we integrate the module_function pattern as its own core ruby object along with Module and Class.

Any kind of proposal of how you imagine this to look?

#5 - 09/25/2017 09:40 AM - rocifier (Ryan O'Connor)

shevegen (Robert A. Heiler) wrote:

> Rocifier, can you say which ruby you have had in mind primarily?

I hadn't thought about it, just wanted to see what people thought of the idea first initially. But now that you explain this, version 3 makes more sense to me.

duerst (Martin Dürst) wrote:

> Based on this understanding, are you saying that there's a third purpose? Or are you thinking about module_function as part of, or a variant of, the second purpose? In both cases, why do you think that your purpose should be split out syntactically, while the 'classic' two purposes stay with the same syntactic device? Or if you have another understanding, then please explain.

Yes, there is a third purpose in this case.

> Also, the most well known module_function example is probably Math.sin and friends. However, as Ruby is an object-oriented language, my personal preference would be to simply write 0.3.sin rather than Math.sin 0.3. I'd expect that commercial applications also would prefer object-oriented architecture and syntax to (namespaced) global functions. What do you think is the reason that you observe extended use of module_function?

Good question. The reason I suppose is that developers don't really see that a class is the right solution to providing namespaced pure functions, since a class has to be instanced, and since what they want to do is call out to a kind of "library" instead of *importing* a "library" in to each class they want to use a single method or two in (I suppose the Math module is a little "all-or-nothing" like that). What devs I know seem to desire is to be able to implement a very simple solution when they require simple architecture. Perhaps an equivalent in a typed language like Java would be a static class. Whether that is good practice or not I guess is debatable, but the point is rather moot since Ruby provides a way to do this already with module_function. What I am proposing is to split the module + module function combination and name it as something new and separate to module. It wouldn't be a required syntax, at least not for a long time.

> > I propose that we integrate the module_function pattern as its own core ruby object along with Module and Class.
>
> Any kind of proposal of how you imagine this to look?

I haven't yet thought up a new name for this duo. If I designed the three concepts from scratch today, I would probably want to propose something like

this:

Mixin (for the classic intention, but I would potentially not have namespacing involved here - instead I would support a practice of nesting a Mixin within a Module as defined below)
Module (for the module + module_function replacement with namespacing)
Class (remains as current concept)

However that's too many breaking changes. I will give some more thought into this soon. Welcome other suggestions!

#6 - 09/25/2017 01:11 PM - phluid61 (Matthew Kerwin)

Sounds to me like you want a new type of object which is like Module but can't be included. If that's the case, I'd suggest a name like Library or Namespace.

I'm still not sold on the value of the proposition. Maybe if its methods were all defined on the singleton class? E.g.:

```
library Foo
  def bar; end
end
Foo.bar #=> nil
```

Perhaps that is beyond the scope of the feature.

#7 - 09/25/2017 10:30 PM - rocifier (Ryan O'Connor)

phluid61 (Matthew Kerwin) wrote:

> Sounds to me like you want a new type of object which is like Module but can't be included. If that's the case, I'd suggest a name like Library or Namespace.
>
> I'm still not sold on the value of the proposition. Maybe if its methods were all defined on the singleton class? E.g.:
>
> ```
> library Foo
>   def bar; end
> end
> Foo.bar #=> nil
> ```
>
> Perhaps that is beyond the scope of the feature.

Yes, something like this :) the value is not huge, I admit, since it is mostly bringing the benefit of readability. However, it is also recognising the two ideas as having distinct purposes at the same syntactic level.