Ruby master - Bug #11779

Module#using does not make sense as a method

12/06/2015 09:40 PM - bughit (bug hit)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>matz (Yukihiro Matsumoto)</td>
</tr>
<tr>
<td>Target version:</td>
<td>ruby -v: 2.2.3</td>
</tr>
<tr>
<td>Backport:</td>
<td>2.0.0: UNKNOWN, 2.1: UNKNOWN, 2.2: UNKNOWN</td>
</tr>
</tbody>
</table>

Description

1. it can't be called from another method
2. the receiver must be self
3. since refinements are lexically scoped the self receiver must match the currently open class

#3 is particularly curious

```ruby
module Refinement
  refine String do
    def refined?
      true
    end
  end
end

module Foo
  def self.refined?
    '.refined? rescue false
  end
end

module Bar
  def self.refined?
    '.refined? rescue false
  end
  Foo.module_eval do
    using Refinement
  end
end

p Foo.refined? #false
```

The module_eval #using call does not raise (it's not from a method and the receiver is self), but evidently because currently open class does not match self, it does not do anything. So it should at least raise.

So #using, though a method, does not function as a method, which is misleading.

History

#1 - 12/07/2015 03:52 AM - bughit (bug hit)
- Subject changed from Module#using does not make sense as a method to Module#using does not make sense as a method

#2 - 12/07/2015 06:52 AM - matz (Yukihiro Matsumoto)
- Status changed from Open to Feedback

Can elaborate what do you want if using should not be a method.
Considering a new keyword would break existing code, I don't think it's a good idea.
Can elaborate what do you want if using should not be a method. Considering a new keyword would break existing code, I don't think it's a good idea.

I don't know what possibilities there are, it just struck me that it was not behaving at all as a method, methods affect their dynamically scoped receiver, using affects the lexically scoped currently open module. So a keyword, if it were possible, would be more appropriate. Don't know what else it could be.

If it stays a method, what about #3

```ruby
module Bar
def self.refined?
  '.refined? rescue false
end
Foo.module_eval do
  using Refinement
end
end
```

what should happen here?

#4 - 12/07/2015 05:00 PM - matz (Yukihiro Matsumoto)
Providing a feature by a method does not imply dynamic scoping, for example, Module#private etc. work in lexical scope.

Matz.

Well, since it's an established pattern, one has to just accept it. It does seem unintuitive to me that method calls, which are dynamically bound to self, are actually operating on the currently open class

#5 - 12/07/2015 05:21 PM - bughit (bug hit)
Yukihiro Matsumoto wrote:

> Providing a feature by a method does not imply dynamic scoping, for example, Module#private etc. work in lexical scope.

Matz.

actually private is more dynamic than using, here's an example of private working not on the currently open class but on the dynamically bound default definee:

```ruby
Class1.class_eval do
  private
  def meth1
  end
end
```

Class1.new.meth1 # private method 'meth1' called

whereas using is not affected by self or "default definee"

```ruby
module Refinement
  refine String do
  end
end
```

03/19/2020
Is the above behavior correct? Instead of doing nothing, should this use of using produce an error or perhaps work inside the module_eval block?

#7 - 12/09/2015 01:17 AM - shugo (Shugo Maeda)
Yukihiro Matsumoto wrote:

Providing a feature by a method does not imply dynamic scoping, for example, Module#private etc. work in lexical scope.

However, it might be better to provide Kernel#using instead of main#using and Module#using, because the behavior does not depend on the receiver.

#8 - 12/09/2015 01:31 AM - bughit (bug hit)
Please explain the following behavior of using:

If this is a noop, as it seems to be, then it should not be allowed

#9 - 12/09/2015 01:41 AM - nobu (Nobuyoshi Nakada)
It is not a noop, just you don't use the refined method there.

#10 - 12/09/2015 01:51 AM - bughit (bug hit)
Nobuyoshi Nakada wrote:

It is not a noop, just you don't use the refined method there.

Where "there"? Where should I use the refined method to see the effect of that using call?

#11 - 12/09/2015 01:55 AM - bughit (bug hit)
bug hit wrote:
Nobuyoshi Nakada wrote:

It is not a noop, just you don't use the refined method there.

Where "there"? Where should I use the refined method to see the effect of that using call?

ok got it, thanks

module Refinement
  refine String do
    def refined?
      true
    end
  end
end

module Foo
  def self.refined?
    ".refined? rescue false
  end
end

Foo.module_eval do
  using Refinement
  p("".refined? rescue false)) #=>true
end

#12 - 12/09/2015 02:00 AM - bughit (bug hit)

bug hit wrote:

bug hit wrote:

Nobuyoshi Nakada wrote:

It is not a noop, just you don't use the refined method there.

Where "there"? Where should I use the refined method to see the effect of that using call?

ok got it, thanks

module Refinement
  refine String do
    def refined?
      true
    end
  end
end

module Foo
  def self.refined?
    ".refined? rescue false
  end
end

Foo.module_eval do
  using Refinement
  p("".refined? rescue false)) #=>true
end

so if using inside module_eval is intended to work, then invoking such a module_eval from a method should also work (it raises Module#using is not permitted in methods):

module Refinement
  refine String do
    def refined?
      true
    end
  end
end
module Foo

    def self.foo
        module_eval do
            using Refinement # Module#using is not permitted in methods
            p('"'.refined? rescue false)
        end
    end

end

#13 - 12/09/2015 05:02 AM - shugo (Shugo Maeda)
bug hit wrote:

    so if using inside module_eval is intended to work, then invoking such a module_eval from a method should also work (it raises Module#using is not permitted in methods):

Module#using is not intented to work as you expect.

Instead, the following extension of *_eval might be considerable:

    module_eval(using: Refinement) {
        ...
    }

#14 - 12/09/2015 05:17 AM - bughit (bug hit)
Shugo Maeda wrote:

    bug hit wrote:

    so if using inside module_eval is intended to work, then invoking such a module_eval from a method should also work (it raises Module#using is not permitted in methods):

Module#using is not intented to work as you expect.

This wasn't really my expectation, it was nobu's explanation (It is not a noop, just you don't use the refined method there), which I confirmed.

"using" called from a module_eval block activates the refinement in the block

Are you saying that's a bug?

#15 - 12/09/2015 05:29 AM - shugo (Shugo Maeda)
bug hit wrote:

    Shugo Maeda wrote:

    bug hit wrote:

    so if using inside module_eval is intended to work, then invoking such a module_eval from a method should also work (it raises Module#using is not permitted in methods):

Module#using is not intented to work as you expect.

This wasn't really my expectation, it was nobu's explanation (It is not a noop, just you don't use the refined method there), which I confirmed.

"using" called from a module_eval block activates the refinement in the block

Are you saying that's a bug?

Nobu didn't explain that Module#using should work in methods.
Module#using is designed not for such dynamic use.

#16 - 12/09/2015 05:40 AM - bughit (bug hit)
Shugo Maeda wrote:

bug hit wrote:

Shugo Maeda wrote:

bug hit wrote:

so if using inside module_eval is intended to work, then invoking such a module_eval from a method should also work (it raises
Module#using is not permitted in methods):

Module#using is not intended to work as you expect.

This wasn't really my expectation, it was nobu's explanation (It is not a noop, just you don't use the refined method there), which I
confirmed.

"using" called from a module_eval block activates the refinement in the block

Are you saying that's a bug?

Nobu didn't explain that Module#using should work in methods.
Module#using is designed not for such dynamic use.

Note that when #using is called in a module_eval block, its effect is confined to the body of the block. So why should it matter if module_eval is called from a method or top level or from another module/class? Regardless of where it's called, #using inside it works the same. The call site does not influence what #using does nor is influenced by it.

#17 - 12/09/2015 06:24 AM - shugo (Shugo Maeda)

bug hit wrote:

Nobu didn't explain that Module#using should work in methods.
Module#using is designed not for such dynamic use.

Note that when #using is called in a module_eval block, its effect is confined to the body of the block. So why should it matter if module_eval is called from a method or top level or from another module/class? Regardless of where it's called, #using inside it works the same. The call site does not influence what #using does nor is influenced by it.

Because refinement activation should be as static as possible.
It might be better to prohibit Module#using in module_eval.

#18 - 12/09/2015 04:37 PM - bughit (bug hit)

Shugo Maeda wrote:

bug hit wrote:

Nobu didn't explain that Module#using should work in methods.
Module#using is designed not for such dynamic use.

Note that when #using is called in a module_eval block, its effect is confined to the body of the block. So why should it matter if module_eval is called from a method or top level or from another module/class? Regardless of where it's called, #using inside it works the same. The call site does not influence what #using does nor is influenced by it.

Because refinement activation should be as static as possible.
It might be better to prohibit Module#using in module_eval.

Perhaps, since you can apply the refinement outside the module_eval block which will also affect the block:

```ruby
module Refinement
  refine String do
    def refined?
      true
    end
  end
end
```
module Foo
end

module Bar
  using Refinement
  Foo.module_eval do
    p(("".refined? rescue false))
  end
end

so a #using in module_eval is only useful if you want the refinement confined to the block. There might be uses for that, not sure.

But if this functionality remains, it should work wherever module_eval is invoked.

#19 - 12/10/2015 03:58 AM - shugo (Shugo Maeda)
bug hit wrote:

so a #using in module_eval is only useful if you want the refinement confined to the block. There might be uses for that, not sure.

Yes, it's the current intended behavior.

But if this functionality remains, it should work wherever module_eval is invoked.

Such dynamic extension of refinements should be discussed in a different ticket as a new feature.

#20 - 12/10/2015 07:01 AM - bughit (bug hit)
Shugo Maeda wrote:

... Because refinement activation should be as static as possible.
...

ruby is too dynamic a language to have a clear distinction between what you're calling "static" and "dynamic" You are labeling class bodies as "static" and methods "dynamic" but a class body is executable ruby and can be invoked by methods, so the current restriction on module_eval can be overcome with relative ease:

module Refinement
  refine String do
    def refined?
      true
    end
  end
end

module Foo
end

module Bar
  def self.mod_eval_with_refine(mod)
    singleton_class.instance_variable_set :@mod, mod
class << self
      @mod.module_eval do
        using Refinement
        p(("".refined? rescue false))
      end
    end
  end
end

Bar.mod_eval_with_refine(Foo)

#21 - 12/10/2015 07:07 AM - shugo (Shugo Maeda)
bug hit wrote:

Shugo Maeda wrote:
Because refinement activation should be as static as possible.

ruby is too dynamic a language to have a clear distinction between what you're calling "static" and "dynamic". You are labeling class bodies as "static" and methods "dynamic" but a class body is executable ruby and can be invoked by methods, so the current restriction on module_eval can be overcome with relative ease:

Methods are expected to be invoked more than once, so there's a significant difference from class bodies. That's why static features like constant assignments are prohibited in method definitions.

#22 - 12/10/2015 07:17 AM - bughit (bug hit)
Shugo Maeda wrote:

bug hit wrote:

Shugo Maeda wrote:

... Because refinement activation should be as static as possible. ...

ruby is too dynamic a language to have a clear distinction between what you're calling "static" and "dynamic". You are labeling class bodies as "static" and methods "dynamic" but a class body is executable ruby and can be invoked by methods, so the current restriction on module_eval can be overcome with relative ease:

Methods are expected to be invoked more than once, so there's a significant difference from class bodies. That's why static features like constant assignments are prohibited in method definitions.

Did you see the example in my previous post? Effectively there is no prohibition against module_eval with #using in methods, because you can open a class in a method and call module_eval from there.

Also you are forgetting a category of methods (class macros) that help initialize/modify classes through meta-programming and are meant to be called once in the class body. Such methods should be able to do whatever the class body can do.

#23 - 12/10/2015 09:28 AM - shugo (Shugo Maeda)
bug hit wrote:

... Because refinement activation should be as static as possible. ...

ruby is too dynamic a language to have a clear distinction between what you're calling "static" and "dynamic". You are labeling class bodies as "static" and methods "dynamic" but a class body is executable ruby and can be invoked by methods, so the current restriction on module_eval can be overcome with relative ease:

Methods are expected to be invoked more than once, so there's a significant difference from class bodies. That's why static features like constant assignments are prohibited in method definitions.

Did you see the example in my previous post? Effectively there is no prohibition against module_eval with #using in methods, because you can open a class in a method and call module_eval from there.

Other features have similar loopholes (e.g., constants can be assigned in methods by eval, private methods can be called by Kernel#send, etc.), but it doesn't mean such restriction is meaningless, because it can express the intention.

Also you are forgetting a category of methods (class macros) that help initialize/modify classes through meta-programming and are meant to be called once in the class body. Such methods should be able to do whatever the class body can do.

Perhaps, perhaps not. I depends on what the phrase "whatever the class body can do" mean. For example, such methods should be able to define constants in a class, but need not to be able to define constants in the same way as in a class body.

#24 - 12/10/2015 04:12 PM - bughit (bug hit)
Shugo Maeda wrote:

bug hit wrote:
ruby is too dynamic a language to have a clear distinction between what you're calling "static" and "dynamic". You are labeling class bodies as "static" and methods "dynamic" but a class body is executable Ruby and can be invoked by methods, so the current restriction on module_eval can be overcome with relative ease:

Methods are expected to be invoked more than once, so there's a significant difference from class bodies. That's why static features like constant assignments are prohibited in method definitions.

Did you see the example in my previous post? Effectively there is no prohibition against module_eval with #using in methods, because you can open a class in a method and call module_eval from there.

Other features have similar loopholes (e.g., constants can be assigned in methods by eval, private methods can be called by Kernel#send, etc.), but it doesn't mean such restriction is meaningless, because it can express the intention.

Also you are forgetting a category of methods (class macros) that help initialize/modify classes through meta-programming and are meant to be called once in the class body. Such methods should be able to do whatever the class body can do.

Perhaps, perhaps not. It depends on what the phrase "whatever the class body can do" mean. For example, such methods should be able to define constants in a class, but need not to be able to define constants in the same way as in a class body.

Of course I don't mean in the same exact way, meta-programming is different than native syntax. However the question of "how" does not apply here, because you are arguing that it should be forbidden (not different), and I'm saying that if a class body can call a module_eval with using, then a class macro method should be able to, as well.

#25 - 12/12/2015 04:35 AM - shugo (Shugo Maeda)
- Assignee set to matz (Yukihiro Matsumoto)

bug hit wrote:

Perhaps, perhaps not. It depends on what the phrase "whatever the class body can do" mean. For example, such methods should be able to define constants in a class, but need not to be able to define constants in the same way as in a class body.

Of course I don't mean in the same exact way, meta-programming is different than native syntax. However the question of "how" does not apply here, because you are arguing that it should be forbidden (not different), and I'm saying that if a class body can call a module_eval with using, then a class macro method should be able to, as well.

As I stated before, it might be better to introduce Kernel#using, which ignores module_eval blocks as constant lookup do.

```ruby
module Foo
  refine String do
    def foo
      puts "foo"
    end
  end
end

module Bar
end

module Baz
  Bar.module_eval do
    using Foo
    ".foo += foo
  end

  ".foo += foo
end

".foo += error
```

I'd like to hear Matz's opinion.

#26 - 12/12/2015 06:06 AM - shugo (Shugo Maeda)

Shugo Maeda wrote:
As I stated before, it might be better to introduce Kernel#using, which ignores module_eval blocks as constant lookup do.

Or it might be better to prohibit using in blocks.

#27 - 07/07/2017 05:07 PM - bughit (bug hit)

matz (Yukihiro Matsumoto) wrote:

Providing a feature by a method does not imply dynamic scoping, for example, Module#private etc. work in lexical scope.

I didn't think about Module#private too deeply at the time, but recently was prompted by something, and Module#private is not lexical

```ruby
module Mod1
  class Class1
  end
  def self.lookup_class
    Class1
  end
  lookup_class.class_eval do
    def foo1
      self
    end
    private
    def foo2
      self
    end
  end
  lookup_class.instance_eval do
    define_method :bar1 do
      self
    end
    def bar1
      self
    end
    private
    def bar2
      self
    end
    define_method :bar2 do
      self
    end
  end
  c1 = Class1.new
  c1.foo1.foo2 rescue puts $!.inspect
  Class1.bar1.bar2 rescue puts $!.inspect
  c1.bar1.bar2 rescue puts $!.inspect
end
```

it affects the dynamically scoped default definee, which though dynamic does not necessarily match the receiver.

#28 - 07/08/2017 02:10 AM - nobu (Nobuyoshi Nakada)

Seems that your "dynamic" and "lexical" words differ from ours.

#29 - 07/08/2017 03:28 AM - bughit (bug hit)

nobu (Nobuyoshi Nakada) wrote:
Seems that your "dynamic" and "lexical" words differ from ours.

I was using "dynamic" to mean that its dynamically "bound" (applies) to the default definee at the point of invocation, not the lexically determined currently open class. It's true that the receiver does not matter at all (I initially thought it had to self):

```ruby
module Mod1
  class Class1
  end

  def self.lookup_class
    Class1
  end

  lookup_class.class_eval do
    def foo1
      self
    end
    Module.new.send(:private)
    def foo2
      self
    end
  end

  c1 = Class1.new
  c1.foo1.foo2 rescue puts $!.inspect
end
```

Please clarify how you're using dynamic vs lexical?

---

**#30 - 04/26/2018 11:05 AM - AlexWayfer (Alexander Popov)**

From RSpec:

```ruby
let(:test_class) do
  Class.new(described_class) do
    private
      using SomeRefiningModule # => RuntimeError: Module#using is not permitted in methods
      def foo
        end
      end
  end
end
```