Ruby - Feature #6810

`module A::B; end` is not equivalent to `module A; module B; end; end` with respect to constant lookup (scope)

07/29/2012 03:55 AM - alexeymuranov (Alexey Muranov)

Status:	Assigned	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		

Description

Is this the expected behavior? To me, it is rather surprising:

```
N = 0

module A
  module B
    def self.f; N; end
  end

  N = 1
end

A::B.f # => 1

N = 0

A::B.f # => 0

A::N = 1

A::B.f # => 1

A::B::N = 2

A::B.f # => 2
```

but

```
N = 0

module A; end

module A::B
  def self.f; N; end
end

module A
  N = 1
end

A::B.f # => 0

N = 0

A::B.f # => 0

A::N = 1

A::B.f # => 0

A::B::N = 2
```

```
A::B.f # => 2
```

Related issues:

Has duplicate Ruby - Feature #16430: Resultion of constants in enclosing clas...	**Rejected**

History

#1 - 07/29/2012 04:10 AM - fxn (Xavier Noria)

Yes, this is expected.

The resolution algorithm first checks the modules in the nesting (in their very constants table, it ignores their ancestors), then the ancestors of the module the constants appears in (the first one in the nesting, or Object if the nesting is empty), and if the former is a module, then Object is checked by hand (and it still tries const_missing if all fails).

The key to understand those examples, in addition to the algorithm, is to be aware of the respective nestings:

```
module A
  module B
    Module.nesting # => [A::B, A]
  end
end
```

whereas

```
module A::B
  Module.nesting # => [A::B]
end
```

In the second case A is not checked, since it is not in the nesting, it is not an ancestor of A::B, and it is not Object.

The nesting is only modified by the class and module keywords, and by opening a singleton class with class << object (and some other obscure cases related to string eval'ing). In particular, the nesting inside and outside class methods defined with def self... is the same.

#2 - 07/29/2012 04:24 AM - alexeymuranov (Alexey Muranov)

Xavier, thanks for the explanation, i see now why it works this way, so this is not a bug. However, can anybody please explain to me why this is a desired behavior, i mean the way the nesting is calculated? Wouldn't it be easier if

```
module A; end
module A::B
  # do something
end
```

was equivalent to

```
module A
  module B
    # do something
  end
end
```

?

P.S. I guess it has to do with lexical scopes... But there has to be a reason to *skip* the module A in the nesting of module A::B, right?

#3 - 07/29/2012 05:09 AM - fxn (Xavier Noria)

I cannot tell you the ultimate rationale behind this, but I can tell you that with the current semantics that is not well-defined.

The problem is that nesting stores module objects, not constant names. Consider for example:

```
module M
end

module A
  B = M
end

module A::B
  Module.nesting # => [M]
end
```

See? No trace of A.

Constants and class and module objects are very decoupled, they are mostly orthogonal concepts in Ruby except for name assignment and some convenient things provided by the class and module keywords.

#4 - 07/29/2012 08:27 PM - alexeymuranov (Alexey Muranov)

Thanks for the example. I do not see however that constants and modules be decoupled: there is Module#name method, so what would be wrong with deriving Module::nesting from the innermost module name? Can an anonymous module ever appear in a nesting?

In other words, what would be wrong if the whole nesting was determined by the name of the last nested module?

#5 - 07/29/2012 09:38 PM - fxn (Xavier Noria)

They are very decoupled, name is the only bit mostly in common, set on constant assignment. May I link to a talk of mine, which covers this (content constrained by the 30min slot): http://www.youtube.com/watch?v=wCyTRdtKm98.

In particular, the name of modules are unrelated to the constants they are stored in. See for example

```
module M
end

N = M

module A
  B = M
end
```

Now, the module object that is *stored* in the constant M, it is also stored in the constant N, and in the constant A::B. It can be stored in a hundred places. And the constant M can disappear:

```
Object.instance_eval { remove_const(:M) }
```

So that module object whose name is "M" is well and alive. You can reopen, mixin the module... everything. The constant M is no longer available, but that is irrelevant as far as Ruby is concerned. Modules are objects, constants just storage, formally they are very superficially related to each other, although in practice we often identify them of course.

#6 - 07/29/2012 10:27 PM - alexeymuranov (Alexey Muranov)

I'll look at the video and think about this, thanks.

#7 - 07/29/2012 11:33 PM - pedz (Perry Smith)

I agree with Alexey... I'm surprised.

His third example shows that the lookup of N is done at run time and starts from the inner scope of B, if no match, search A, if no match search globals. ... At least, that is how it appears to me.

But the fourth example does not mimic this search pattern. It is as if A::B is not nested under A but is a unique constant.

Xavier's first reply https://bugs.ruby-lang.org/issues/6810#note-1 shows this as well.

This explains some "weirdness" (a.k.a. "surprise") I've experienced in the past and I view it as a bug. Indeed, I view both outputs in Xavier's first reply as wrong. They both should be [B,A]. A::B is just a shorthand to gain access to the constant B syntactically nested inside of module A.

At least, that is what I thought it was doing all this time.

Looking at the Dave Thomas' Ruby 1.9 book for why I think this way I see a lot of reasons why. And I wonder if this doesn't relate to some of the class variable "leakage" he details around page 308 in Part III, Chapter 22, section "Scope of Constants and Variables"

Now... it may be that it won't change. So I think Xavier's presentation needs to be part of Ruby 101 rather than an obscure relatively unknown topic.

#8 - 07/29/2012 11:49 PM - fxn (Xavier Noria)

Perry, totally agreed. The way this works and how constant name resolution works should be a well-covered topic. But traditionally it has not been very well documented. In particular the separation between constants and modules/classes is unknown to most people, which I think it is a symptom. I have had to research this topic myself to figure out what is going on.

If I get some time I'd like to self publish a monograph about constants in Ruby, with a second part on constant autoloading in Ruby on Rails.

#9 - 07/30/2012 12:40 AM - alexeymuranov (Alexey Muranov)

If i express the rule for constant lookup as i have understood it, it seems that the lookup follows the code indentation from right to left, and in particular in

```
module A
  module B; N=1; end
```

```
  module B::C
    puts N # => NameError: uninitialized constant A::B::C::N
  end
end
```

the constant N will not be found because it is neither in A::B::C, nor in A, nor in Object, and A::B is not searched, because (hehe) there are only 2 levels of indentation at puts N.

Updated and rewritten 2012-10-31

I was not very attentive and not immediately noticed that lexical nesting is checked before ancestors. So class constants are only "half-inherited". I think i am going to specify the full path, starting with ::, to every constant that does not belong to the current module.

What would you say about the following request: when looking up a constant,

1. first, follow the inheritance hierarchy.
2. then, split the module name on '::' and follow the name hierarchy (this does not happen now),
3. last, follow the code nesting lexical hierarchy (called Module::nesting).

It seems to be a common practice anyway to specify the complete path to a constant unless it is in the current module. Non-lexical lookup rules can at least in some cases make it possible to use a constant without specifying a complete path and without worrying that moving a part of code to a separate file would make the constant disappear from the scope.

I am not sure actually what would be the most natural behavior.

#10 - 11/03/2012 12:17 PM - mame (Yusuke Endoh)

- Tracker changed from Bug to Feature

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

- Priority changed from Normal to 3

As Xavier Noria said, this is actually an intended behavior. So I'm moving this ticket to the feature tracker.
And assigning to matz, though I think it is hopeless to change this basic behavior...
At least, matz can explain the rationale.

--
Yusuke Endoh mame@tsg.ne.jp

#11 - 11/03/2012 12:17 PM - mame (Yusuke Endoh)

- Target version set to 3.0

#12 - 05/18/2014 01:17 PM - alexeymuranov (Alexey Muranov)

After not doing any Ruby for a while, i find it hard again to recall the constant scope and inheritance rules.

I know this would be a major change, but **maybe somehow the rules for the constants can be made identical to the rules for methods?** That is, a constant would be essentially a method that always returns the same thing.

I guess, what it means is that

```
FOO = 1
```

would be somewhat equivalent to

```
def FOO; 1 end
```

or

```
module_function def FOO; 1 end
```

or something else along these lines.

(I have not thought enough to figure out how many incompatibilities this would cause, but this would be easy to remember and understand IMO.)

By the way, this would be consistent with my other idea #6806 of using foo::bar instead of foo.bar for methods without side effects.

Edited 2014-05-22

#13 - 01/27/2017 11:58 AM - mijoharas (Michael Hauser-Raspe)

There is another duplicate of this (#11705). I understand this is expected behaviour and it makes sense that this is the way it is with the current

architecture.

I don't however think that this is the way it *should* be. Does anyone else have any opinions on this topic? I think it inhibits the simple code organisation of large projects.

#14 - 12/21/2019 03:50 AM - mame (Yusuke Endoh)

- Has duplicate Feature #16430: Resultion of constants in enclosing class/module affected by how nested classes/modules are declared added

#15 - 12/21/2019 04:52 PM - MikeVastola (Mike Vastola)

I just inadvertently made a dup of this issue (#16430) and would like to throw my hat in the ring as supporting this change.

Honestly, this literally the first time since I started ruby that I encountered something in the language that was inherently non-intuitive. I'm also not sure I understand the use case for not including enclosing modules in the constant resolution hierarchy.

One idea that I did have though is -- if this is seen as a breaking change (or otherwise difficult to implement with the current codebase) -- creating an alternate scope resolution operator (I was thinking :::) when defining modules where you want the scope resolution order to follow the hierarchy.

#16 - 12/22/2019 04:28 AM - sawa (Tsuyoshi Sawada)

- Description updated

#17 - 12/22/2019 04:38 AM - sawa (Tsuyoshi Sawada)

- Description updated

#18 - 12/22/2019 04:55 AM - sawa (Tsuyoshi Sawada)

At least as of Ruby 2.6.5, I get a different output from above for at least one of the examples:

```
N = 0

module A
  module B
    def self.f; N; end
  end

  N = 1
end

N = 0

A::B.f # => 1
```

#19 - 12/28/2019 04:21 AM - kernigh (George Koehler)

My oldest Ruby 1.8.2 (2004-12-25) also says A::B.f # => 1 for that example. That Ruby is 15 years old. The example showing A::B.f # => 0 is 7 years old. I suspect that A::B.f # => 0 was a copy mistake, and Ruby always had A::B.f # => 1.

#20 - 12/10/2020 09:22 AM - mame (Yusuke Endoh)

- Target version deleted (3.0)