Ruby master - Bug #11705

Namespace resolution in nested modules with short syntax

11/17/2015 11:28 PM - mwpastore (Mike Pastore)

Status:	Rejected		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:		**Backport:**	2.0.0: UNKNOWN, 2.1: UNKNOWN, 2.2: UNKNOWN

Description

Given the following definition:

```
module Foo
  class Qux
    def self.hello
      'Hello, world!'
    end
  end
end
```

Namespace resolution at a later time works differently when you have nested modules, e.g.

```
module Foo
  module Bar
    # Can't find Foo::Bar::Qux, so "goes up" to find Foo::Qux.
    p Qux.hello # < "Hello, world!"
  end
end
```

vs. the short syntax, e.g.

```
module Foo::Bar
  # Can't find Foo::Bar::Qux, but doesn't "go up" to find Foo::Qux.
  p Qux.hello # < in `<module:Bar>': uninitialized constant Foo::Bar::Qux (NameError)
end
```

Is this intentional and/or expected?

Related issues:

Has duplicate Ruby master - Feature #16430: Resultion of constants in enclosi...	**Rejected**

History

#1 - 12/18/2015 08:30 AM - shugo (Shugo Maeda)

- Status changed from Open to Rejected

Mike Pastore wrote:

> Is this intentional and/or expected?

It's intentional and expected.

If class and/or module definitions are explicitly nested, constants of outer classes and/or modules are looked up.
However, if a class or module definition is not nested, only constants in the class or module, its ancestors,
and if the target is a module, Object and Object's ancestors are looked up.

You can see module nesting information by Module.nesting.

```
module Foo
  module Bar
    p Module.nesting #=> [Foo::Bar, Foo]
    p Qux.hello      #=> "Hello, world!"
```

```
  end
end

module Foo::Bar
  p Module.nesting   #=> [Foo::Bar]
  p Qux.hello        #=> NameError, because Foo is not included in Module.nesting
end
```

#2 - 12/18/2019 01:18 AM - mame (Yusuke Endoh)

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