# Hello everyone. I'm not a very advanced ruby user, and I
# would like to provide an outsider report on certain ruby
# behavior that might surprise newbies.

```ruby
module A
  class X
    def hello; puts 'hello' end
  end
end

module B
  include A
end

B::X.new.hello
=> hello
# As expected.

# But when I tried to add new functionality to X, ...

module B
  class X
    def goodbye; puts 'goodbye' end
  end
end

B::X.new.hello
=> NoMethodError

# I was surprised, that my .hello method disappeared,
# when all I was trying to do, was to improve X in B.
# I actually somehow expected to work on a subclass
# of X, like this:

module C
  include A
  class X < X
    def goodbye; puts 'goodbye' end
  end
end

# My suggestions are:
# 1. I consider 'class X < X' syntax a little bit
#    mysterious. How about making this a default
#    behavior for 'class X' statements?
# 2. If the above is not considered beneficial, I
#    would welcome if 'class X' statement warned
#    when shadowing an existing name. People might
#    often assume that they are opening an existing
#    class, rather than getting a brand new one
#    shadowing the previous one. If people really
#    want a brand new shadowing class without warning
#    they could use explicit 'X = Class.new'.
```
#1 - 03/28/2012 12:23 AM - mame (Yusuke Endoh)
- Status changed from Open to Assigned
- Assignee set to matz (Yukihiro Matsumoto)

#2 - 04/09/2012 08:12 AM - matz (Yukihiro Matsumoto)
- Assignee changed from matz (Yukihiro Matsumoto) to mame (Yusuke Endoh)
- Target version set to 3.0

Quite reasonable. But the change would introduce serious incompatibility, so that we can not make this happen in 2.0.

Matz.

#3 - 04/18/2012 12:23 AM - mame (Yusuke Endoh)
Hello, Matz

Do you think how should it be fixed?  Suggestion 1?
I think Suggestion 2 (just warning) can be included in 2.0.

--
Yusuke Endoh mame@tsg.ne.jp

#4 - 04/18/2012 10:41 AM - matz (Yukihiro Matsumoto)

Basically it is just alternative appearance of:

```ruby
class A
  A = 22
end

class B < A
end

p B::A
```

```ruby
class B
  A = 42
end

p B::A
```

Current Ruby does not warn you.  If you want to warn class overriding, you have to warn above problem as well.

Matz.

#5 - 05/09/2012 02:47 AM - Anonymous

Well, it feels weird to disagree with the creator of the language you decided to use for your career, but matz, I think that in your example, in

```ruby
class B
  A = 42
end
```

it is very clear, that you are performing an assignment. There is no need for warning.

But my problem was, that I thought I was opening an existing class using

```ruby
class X
  def goodbye, puts 'goodbye' end
end
```

that is, not performing assignment to X, but changing the object to which X refers. But instead, "class X" statement performed a hidden assignment of a newly created class to X. I believe that in this case, warning is needed. The warning could be avoided by eg. explicitly assigning

```ruby
X = Class.new
```

and then working with it:

```ruby
class X
  # do the definitions
```
The warning should activate only when X was pre-assigned and hidden assignment is being performed by 'class' statement.

#6 - 05/09/2012 03:41 AM - Anonymous

In this place, I'd also like to provide user feedback about my (unwarranted, but real) surprise involving a weakly related issue:

```ruby
module A; def initialize; puts "hello"; end
class B; include A end
A = Module.new
B.new
> "hello"
```

And there I was surprised. I did get constant redefine warning, but I ignored it, as I was writing tests and I wanted to blank out A mixin. I clung on the class name A, but to include statement, capital 'A' was just as irrelevant as if it was small 'a' (Another proof that there are actually no constants in Ruby :)

I hope I didn't waste too much of your time by this.

#7 - 05/09/2012 04:23 AM - Eregon (Benoit Daloze)

Hello,

On 8 May 2012 20:41, boris_stitnicky (Boris Stitnicky)
boris@iis.sinica.edu.tw wrote:

In this place, I'd also like to provide user feedback about my (unwarranted, but real) surprise involving a weakly related issue:

```ruby
module A; def initialize; puts "hello"; end; end # there was a missing end
class B; include A end
A = Module.new
B.new
> "hello"
```

And there I was surprised. I did get constant redefine warning, but I ignored it, as I was writing tests and I wanted to blank out A mixin. I clung on the class name A, but to include statement, capital 'A' was just as irrelevant as if it was small 'a' (Another proof that there are actually no constants in Ruby :)

As you say, the constants are not constant in a usual sense, they are global identifiers that should not be reassigned.

You can think of module Name as Name = defined?(Name) ? Name : Module.new and the change of scope (same for class Name). That is, if Name is already defined, just use it, otherwise create the module.

What happens here is you create a module A with a method initialize. You then include it in B, which means adding the module instance in the ancestors:

```ruby
B.ancestors
=> [B, A, Object, Kernel, BasicObject]
```

Then you actually redefine A to an empty module.

Unfortunately, if you do again

```ruby
B.ancestors
=> [B, A, Object, Kernel, BasicObject]
```

You still see A in the ancestors (this might be worth to be reported as a separate issue), although that module you see is still the old one, which name was not updated:

```ruby
B.ancestors[1].private_instance_methods
=> [:initialize]
```
This is maybe not the most intuitive behavior, but it is consistent. An object is not bound to a variable or a constant, and the references are direct. The only "magic" that happens is when you assign for the first time a class/module to a constant, it takes the name of that constant.

In any case, avoid redefining constants as much as possible, and don't make assumptions on constant resolution when the module is included. If you want to ensure to re-open a class, just use `ClassName.class_exec { ... }`.

Back to the original issue, having

```ruby
module A; class X; def hello; puts 'hello' end end end
module B; include A end
```

then using "class X" statement inside B module does not behave as you say, "X = defined?(X) ? X : Class.new", but as "X = self.constants(false).include?(X) ? X : Class.new", which is undocumented and yet has to be memorized - a surefire recipe for unwanted language exploration session in irb.

Let me say in detail what problem I was solving back then. It was a Petri net with place and transition classes in its namespace:

```ruby
module Petri
  class Place # define Petri net place (marking property etc.)
    end
  class Transition # defines Petri net transition (firing, enabled-disabled etc.)
    end
  class Net # a collection of connected Places and Transitions
    end
end
```

Having defined this, I said to myself: Now I'll make a special kind of a Petri net, that can do some additional tricks:

```ruby
module ChemicalPetri
  include Petri
  NA = 6.022e23 # teach it Avogadro's number
  class Transition
    # teach transitions Arrhenius equation
    end
end
```

In fact, I expected to work on a deep subclass of 'Petri' module. But "class Transition" gave me a brand new class silently, and there I had to forget about chemical equations and hit irb. After 1 day, I figured out I have to write:

```ruby
class Transition < Transition
  # teach transitions Arrhenius equation
  end
```

Yet, "class Transition" was the first thing that jumped to my mind to get what I wanted. From retrospective, there are 3 logically justified behaviors for "class Transition" statement in this situation:

1. Opening the ancestor's class, explicitly:
   ```ruby
class Petri::Transition; # do modifications
   end
   This is most "logical", because Transition constant is already there, but opening ancestor's assets in offspring modules is hardly a good habit.
   
2. Creation of a brand new class, explicitly:
   ```ruby
   Transition = Class.new; class Transition; # do modifications
   end
   Less logical, but justifiable behavior, encouraging bad habits less.
   
3. Operation on a subclass, explicitly:
   ```ruby
   class Transition < Transition; # do modifications
   end
   Perhaps least logical, but I suspect that most frequently needed behavior.
I lean towards concluding, that in this situation "class X" statement is always intuitively ambiguous and perhaps should always warn, explaining what exactly it is doing, no matter which of the 3 behaviors is chosen in Ruby implementation.

User should explicitly either ask for the parent's class (class A::X), or explicitly create a new class (X = Class.new), or explicitly subclass parent's X (class X < X). Since "class X < X" requires good understanding of what's going on behind the scenes, perhaps there should be a new statement(s) controlling this kind of subclassing behavior, something like "subclass_also X", "deep_subclass_also X". (These are really not good suggestions)

In sum, I'm trying to convey my feelings that once Ruby is used as a math language to make slightly more complex object models, deep subclassing might be an everyday need and should be provided for in the language itself, rather than expecting users to write their own gems for this.

#9 - 05/12/2012 04:53 AM - Eregon (Benoit Daloze)

On 9 May 2012 13:49, boris_stitnicky (Boris Stitnicky)
boris@iis.sinica.edu.tw wrote:

Issue #5632 has been updated by boris_stitnicky (Boris Stitnicky).

Back to the original issue, having
module A; class X; def hello; puts 'hello' end end end
module B; include A end

then using "class X" statement inside B module does not behave as you say, "X = defined?(X) ? X : Class.new", but as "X = self.constants(false).include?(:X) ? X : Class.new",

Indeed, that's what I wanted to say, thanks for finding my error.

From retrospective, there are 3 logically justified behaviors for "class Transition" statement in this situation:

1. Opening the ancestor's class, explicitly:
   class Petri::Transition; # do modifications end
   This is most "logical", because Transition constant is already there, but opening ancestor's assets in offspring modules is hardly a good habit.

2. Creation of a brand new class, explicitly:
   Transition = Class.new; class Transition; # do modifications end
   Less logical, but justifiable behavior, encouraging bad habits less.

3. Operation on a subclass, explicitly:
   class Transition < Transition; # do modifications end
   Perhaps least logical, but I suspect that most frequently needed behavior.

I would not want the class keyword without "< ParentClass" to inherit from anything else than Object.

The main class keyword use is to create a class, so it makes sense to me it reopens only if it is in the strictly same context (self.constants(false)).

But I'll agree with you "class X < X" is rather confusing. I would rather write it "class Transition < Petri::Transition" in your case, to be clear on the intention.

I believe a good documentation on constants would be a great addition.

#10 - 05/18/2012 01:29 PM - Anonymous

I would not want the class keyword without "< ParentClass" to inherit from anything else than Object.

Yes, that's correct when creating a new class. But I was not interested.
in creating a class in my code. I just wanted to open the class inherited
from the namespace of the parent module (Petri) and teach it new tricks in
the child module (ChemicalPetri), representing Petri applied to chemistry,
all of this without changing the vanilla Petri in any way.

#11 - 01/21/2019 08:53 AM - mame (Yusuke Endoh)
- Description updated

#12 - 02/07/2019 07:49 AM - matz (Yukihiro Matsumoto)
- Status changed from Assigned to Closed

After 7 years of consideration, I reject this issue. In the example in the original, the first X class is defined as A::X and the second definition (override attempt) was done in B::X. Since they are different, we define different classes. Case closed.

Matz.