# Ruby master - Feature #7240

## Inheritable #included/#extended Hooks For Modules

10/30/2012 07:08 AM - apotonick (Nick Sutterer)

| | | |
|---|---|---|
| **Status:** | Feedback | |
| **Priority:** | Normal | |
| **Assignee:** | | |
| **Target version:** | | |

**Description**

An inheritable hook mechanism for modules would be a great way for module/gem authors to simplify their implementations.

## The Problem

Let's say I have the following module.

module A
def self.included(base)
# add class methods to base
end
end

So, A is overriding the #included hook to add class methods to base.

module B
include A

# class methods from A are here.
end

Since B is including A, A's #included method is invoked and A's class methods will be copied to B.

module C
include B

# class methods from B are lost.
end

When including B into C, B's #included is invoked and A's #included is lost. In our example, this means no class methods from A are in C.

## Proposal

It would be cool if #included/#extended in a module could be inherited to including descendants. I wrote a small gem "uber" that does this kind of stuff with a simple recursion. Roughly, it works like this.

module A
extend InheritableIncluded # "execute my #included everytime me or my descendents are included."

```
def self.included(base)
  # add class methods to base
end
```

end

Now, A's #included is invoked every time it or a descending module is included. In our example, class methods from A would be around in C.

When discussing this with Matz we agreed that this might be really useful in Ruby itself. I'm just not sure how to mark inheritable hooks.

**History**

#### #1 - 10/31/2012 05:42 AM - drbrain (Eric Hodel)

=begin
What is wrong with using super?

$ cat t.rb
module A
def self.included mod
puts "#{mod} included A"
end
end

module B
def self.included mod
puts "#{mod} included B"

```
  super
end

include A
```

end

module C
include B
end

$ ruby20 t.rb
B included A
C included B

=end

#### #2 - 10/31/2012 05:44 AM - drbrain (Eric Hodel)

Sorry, I didn't read my own output, please disregard.

#### #3 - 10/31/2012 07:34 AM - alexeymuranov (Alexey Muranov)

In my opinion, the most common use case for the included hook in modules is adding class methods. However, to me this looks hackish -- hard to follow and understand. I would have preferred if there was a way to define both instance-level and class-level behavior in a module and include both at once:

module M
def foo
# ...
end

```
def base.bar  # here a fictional method Module#base is called
  # ...
end
```

end

class C
meta_include M
end

a = C.new
a.foo
C.bar

After all, ordinary objects have no method table, classes have one method table, why not to allow modules to have 2 method tables?

Talking about method tables, maybe if objects were allowed to have one, there would be no need for metaclasses? (Then classes would have 2 method tables, modules would have 3, and in fact methods and method tables could be regarded as different kinds of attributes.)

By the way, i think that from the point of view of English, the method name included is not consistent with the method name extended (the *module* is included, but the *base* is extended).

#### #4 - 10/31/2012 12:54 PM - trans (Thomas Sawyer)

=begin
I have always thought module class methods should be included. Matz has stated his opposition to it b/c he sees it as a hindrance for some use cases, for example, internal methods spilling in when including the Math module. But I've always felt that those occurrences would always be easier to counteract (by just making a separate inner-module) if it were really necessary, than hacking around the limitations of the current behavior.

Short of changing the behavior of (({#include})), there are two other options. The first is to add another method like #include that does the deed. A good name for which is, as usual, hard to come up with. (Maybe (({#include!})) would be a viable option?) The other possibility is to allow class methods to be classified as inclusive or exclusive. By default module class methods would be exclusive, but using a keyword/method this could be changed in much the same way that methods are classed into public/private/protected.

For example:

```
module M
  def self.foo; "foo"; end

  inclusive
  def self.bar; "bar"; end
end

module N
  include M
end

N.bar  => "bar"
N.foo  => Error
```

I'm not sure I like this idea --I still think it would be better just to make it the usual behavior of (({#include})), but it is an option.
=end

#### #5 - 10/31/2012 06:02 PM - alexeymuranov (Alexey Muranov)

=begin
trans (Thomas Sawyer) wrote:

> I'm not sure I like this idea --I still think it would be better just to make it the usual behavior of (({#include})), but it is an option.

I think that there would be issues with "including" methods defined on the module itself. For example:

module M
# ...
end

class C
include! M
end

Now, which methods defined of M can be called on C? Only those from the meta-class of M? Those from the meta-class of M and from the ancestors of the meta-class? All public? All? I cannot come up from the top of my mind with a (({Module})) public method which is not defined in (({Class})), but there are at least private ones, for example: should (({C.send(:module_function, :f)})) work after the inclusion in the above example?

To me, a possible solution seems to be to have a second method table in modules (i suggested above to define such methods as singleton methods on some object returned by some private method named, for example, (({Module#base})). The inclusion method could be still called (({include})) as it would not conflict with the current usage.

To state completely my current point of view, meta-class looks to me like a hack (to make up for the fact that objects are not allowed to keep their own methods), and inheriting (({Class})) from (({Module})) does not look as a very good idea (this is witnessed for example by the fact that a class cannot be ((*included*)), despite that (({Class < Module}))) :).
=end

#### #6 - 10/31/2012 06:32 PM - alexeymuranov (Alexey Muranov)

I'll open a feature request do discuss module inclusion separately.

Update: #7250

#### #7 - 10/31/2012 07:01 PM - apotonick (Nick Sutterer)

=begin
Interesting points, mates. I don't know about the method tables so all I can do is summarizing what we came up with so far.

== 1. Explicitely define inheritable class methods.

module M
inclusive/inheritable
def self.i_will_be_inherited

When then included, this method will be inherited. This would solve most problems as

inclusive/inheritable

def self.included(..)

would mean that the #included method itself can be made inheritable!

== Explicitely inherit when including

If we had a new #include! method we inherit all (?) class methods.

module M
def self.m
end

module U
include! M

```
# class method #m is now around.
```

end

Problem here is: The *user* has to know about whether or not to import class methods which is not what I want as a gem developer (I don't want my gem users to know whether they should use #include or #include!). Also, no fine-tuning since all class methods are imported.

=end

#### #8 - 10/31/2012 07:24 PM - alexeymuranov (Alexey Muranov)

apotonick (Nick Sutterer) wrote:

> Problem here is: The *user* has to know about whether or not to import class methods which is not what I want as a gem developer (I don't want my gem users to know whether they should use #include or #include!). Also, no fine-tuning since all class methods are imported.

If i understood correctly, Thomas suggested the include! method name simply to not change the current behavior of include, but it seems that he also prefers a single method.

I do not think that in your example the method m should be callable on U because i think it is really a *personal* method of M not intended for inclusion. This is why i suggested to have a separate space (object/table/module/etc) where to keep methods intended for inclusion as class methods. But i think this is a huge feature request, which may bring into question even the relationship Class < Module. I wonder what the core developers would say.

However it seems to me that if it is possible to implement this "single instance/class method inclusion", the end user of a gem would do exactly as before: simply call include (or meta_include, but it can be called include as it wouldn't conflict with current include). The fine tuning by the developer should be possible by choosing which methods to declare as "includable instance methods" and "includable class methods".

#### #9 - 10/31/2012 07:45 PM - apotonick (Nick Sutterer)

=begin
Ah ok, now I understand you, Alexey. Instead of having a keyword (inclusive/inheritable) you want to have a special "namespace" as in

module M
def to_be_inherited.m; end

which would include the class method #m when including M? Cool. However, how would we make self.included "inheritable" then?

> I wonder what the core developers would say.

I talked to Matz at the Sapporo RubyKaigi this year and he made me file this issue!
=end

#### #10 - 10/31/2012 08:18 PM - alexeymuranov (Alexey Muranov)

apotonick (Nick Sutterer) wrote:

> =begin
> Ah ok, now I understand you, Alexey. Instead of having a keyword (inclusive/inheritable) you want to have a special "namespace" as in
>
> module M
> def to_be_inherited.m; end
>
> which would include the class method #m when including M? Cool. However, how would we make self.included "inheritable" then?
>
> > I wonder what the core developers would say.

I talked to Matz at the Sapporo RubyKaigi this year and he made me file this issue!
=end

I am sorry, i used your thread to propose my own feature request in a sense, i have already corrected this with #7250. I hoped that maybe there would be no need for inheritable self.included if "class-level includable" methods could be defined in a module, and if including one module in another would treat those methods appropriately.

"Inheritable self.included" seems a confusing concept for me, because modules do not inherit from one another. I even think that including a module in a class and including a module in a module are different operations which maybe should have been called differently.

I propose roughly the following diagram and behavior:

module
personal_singleton_methods
includable_instance_methods
includable_class_methods

class
personal_singleton_methods
instance_methods

object
personal_singleton_methods

Writing

module M; end
module N
include M # not the same as including a module into a class with respect to includable_class_methods
end
class C < AnAncestor
include N # not the same as including a module into a module with respect to includable_class_methods
end

would result in having pointers

N.includable_instance_methods -> M.includable_instance_methods
N.includable_class_methods -> M.includable_class_methods

and hidden inheritance

C < N_proxy_for_C < M_proxy_for_C < AnAncestor

where N_proxy_for_C has N.includable_instance_methods for its own instance methods and has N.includable_class_methods for its own class methods.

These are rough ideas, i am not familiar with Ruby implementation details.

#### #11 - 11/05/2012 01:30 PM - matz (Yukihiro Matsumoto)

Oops, I made mistakes.

I am positive about the change, but I have concern about compatibility.
Maybe we should provide a new method, but I cannot think of any good name right now.

Matz.

#### #12 - 11/24/2012 10:12 AM - mame (Yusuke Endoh)

*- Status changed from Open to Feedback*

*- Target version set to 2.6*

#### #13 - 04/17/2014 11:46 PM - bkatzung (Brian Katzung)

I'm a relatively new Ruby programmer so I may have missed some of the nuances, but I have written http://rubygems.org/gems/extended_include that I *believe* satisfies the original posting requirements.

In a nutshell:

```
require 'extended_include'

module A
  # ...
  include_class_methods # (module=ClassMethods, ...) -OR-
```

```
  include_class_methods do
    def cm; "Well I'll be a class method!!!"; end
  end
  # (A.extend Extended_Include to get ::included is automatic)
end

module B
  extended_include A # include + arrange to chain A's ::included (if present and base needs it) in ours
  def self.included (base) # Only if we need to do something beyond Extended_Import::included
    super base rescue nil # Let Extended_Include extend and chain
    # Do something more...
  end
end
```

If somebody more experienced would like to look it over, I would love some feedback.

#### #14 - 12/25/2017 06:15 PM - naruse (Yui NARUSE)

*- Target version deleted (2.6)*