Ruby master - Feature #11550

Current behaviour of super(...) is dangerous in the presence of more than one included modules.

09/25/2015 06:46 AM - rovf (Ronald Fischer)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Consider a class

```
class C <P
  include M1
  include M2
  def initialize
    ....
  end
end
```

If P, M1 and M2 all provide a initialize method, and C::initialize calls super(...), the first initialize module in the chain, which has a formal parameter list matching the actual parameters in the super(....) call, is executed. The other ones are not executed. The following article demonstrates a clever way, how C::initialize can call all initializers of the included modules and of the parent class:

[[[http://stdout.koraktor.de/blog/2010/10/13/ruby-calling-super-constructors-from-multiple-included-modules/]]]

This solution works, but is complicated, and the reason is that the basic design of Ruby, with respect of initializing the base class, is flawed, for the following reason:

If we define an 'initialize' method, we certainly assume that its execution is necessary for the correct behaviour of our class or module. Therefore, the designer of a class or module should at least have the possibility to REQUIRE that initialize will be called (by a derived class), and the designer of a class which inherits from a parent class or includes a module, should have an EASY way to call all the parent initializers.

Here a first draft of how the language could be changed to meet this criterium; it's perhaps not the best design, but it might help clarifying my point:

(1) A class (or module) may define either a method initialize or a method initialize_strict (but not both). If it has initialize_strict defined, and the class where the module is included, respectively the class which inherits it, does NOT invoke this initializer, an exception is thrown, UNLESS initialize_strict can be called without parameters. In the latter case, it is always executed (even if no 'super' call is present).

(2) A class method super_of is added to class Object, with the prototype super_of(klass,arg*), where klass is a symbol or String or instance of type Class or Module. The affect of invoking super_of(:Foo,x,y) is identical to invoking initialize(x,y) in the ancestor class Foo. If Foo is neither a direct ancestor nor an included module, an exception is thrown.

History

#1 - 09/25/2015 11:08 AM - Eregon (Benoit Daloze)

Why not simply having super in M1 and M2 #initialize?
All constructors should call super, unless they just inherit from Object/BasicObject.

#2 - 10/12/2015 10:47 AM - rovf (Ronald Fischer)

Benoit Daloze wrote:

> Why not simply having super in M1 and M2 #initialize?
> All constructors should call super, unless they just inherit from Object/BasicObject.

First, M1 and M2 don't know where they are going to be mixed in, so they can not invoke suuper - they don't know what parameters are being passed.

Second, the class which mixes in M1 and M2 needs to initialize two modules, which is not possible with the current language.

#3 - 10/12/2015 12:54 PM - Eregon (Benoit Daloze)

Ronald Fischer wrote:

> First, M1 and M2 don't know where they are going to be mixed in, so they can not invoke suuper - they don't know what parameters are being passed.
>
> Second, the class which mixes in M1 and M2 needs to initialize two modules, which is not possible with the current language.

Would

```
module M1
  def initialize(*)
    super
  end
end
```

work for your use case?
Then the whole hierarchy can be initialized by just calling super.

#4 - 01/20/2016 08:44 AM - rovf (Ronald Fischer)

Benoit Daloze wrote:

> Ronald Fischer wrote:
>
> > First, M1 and M2 don't know where they are going to be mixed in, so they can not invoke suuper - they don't know what parameters are being passed.
> >
> > Second, the class which mixes in M1 and M2 needs to initialize two modules, which is not possible with the current language.
>
> Would
>
> ```
> module M1
>   def initialize(*)
>     super
>   end
> end
> ```
>
> work for your use case?
> Then the whole hierarchy can be initialized by just calling super.

This doesn't help either. It is not a problem of the correct number of parameters, but of the semantics. Even in your design, there is no way that class C can tell to the modules M1 and M2, which parameters should be used for each respective module, UNLESS the modules cooperate in the protocol.

For example, if each module allows an arbitrary list of named parameters, but picks from this list exactly the parameter which has a key (name) of the name of this module, the constructor of C could write

```
super(M1: [3,5,8], M2: %w(foo bar))
```

and M1 would use the parameters 3,5,8 and M2 would use the parameters foo and bar.

While this would work (and, IMHO, be even a very useful way), it would require that all modules use the same convention for parameter passing. I can do this in my own application, when I am designing my modules, but it doesn't work well when I publish a module for general use.