Ruby - Bug #16100

Visibility modifiers don't call super correctly when overridden in alternative ways

08/14/2019 03:05 AM - prebsch (Patrick Rebsch)

Status:	Closed		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:	ruby 2.7.0dev (2019-08-14) [x86_64-darwin18]	**Backport:**	2.5: UNKNOWN, 2.6: UNKNOWN

Description

It seems that the method visibility modifiers don't call super correctly when they are overridden in certain ways. I expected the following examples to all behave the same since they are all being defined on the singleton class, but only the first operates correctly presumably because it is explicitly defined on the singleton class. I've reproduced this behavior with 2.7.0, 2.6.3, and 2.5.5.

```
def test_visibility(description, klass)
  puts "Case: #{ description }"
  puts "  #=> #{ klass.private_instance_methods.include?(:private_method) }"
  puts
end

test_visibility('explicit', Class.new {
  def self.private(*); super; end
  private; def private_method; end
})

test_visibility('opened singleton', Class.new {
  class << self
    def private(*); super; end
  end
  private; def private_method; end
})

test_visibility('include/prepend to singleton', Class.new {
  module M
    def private(*); super; end
  end
  singleton_class.prepend(M)
  private; def private_method; end
})
```

```
Case: explicit
  #=> true

Case: opened singleton
  #=> false

Case: include/prepend to singleton
  #=> false
```

History

#1 - 08/14/2019 03:50 AM - jeremyevans0 (Jeremy Evans)

This issue is related to scope. In all cases, your call to private is calling ruby's default behavior. However, when you call super, it just changes the scope of the super call to private. The scope of your overridden private method in your explicit example just happens to be the same as when you call private on the next line, so you get the same behavior.

Here's an example that shows this in action. Note that C.private is called outside the scope of class C, but the effect takes place inside the scope of class C.

```
t = Thread.new do
  class C
    def self.private(*); super; end
    sleep 0.5
    def private_method; end
  end
end
sleep 0.25
C.private
t.join
C.private_instance_methods(false)
# => [:private_method]
```

I don't think this is a bug, I think this is an implementation detail. Hopefully another committer can confirm that.

This does make it a bad idea to override private/protected/public/module_function, since normal Ruby code cannot replicate their affect on scopes.

#2 - 08/17/2019 01:54 AM - prebsch (Patrick Rebsch)

Thanks for the explanation, Jeremy. I *think* I understand now. I took a look at the source and made another example to help me understand. I see now that the inconsistency only arises with the argument-less call of the visibility modifier.

```
def test_private_visibility(klass, method_name)
  puts
  puts "Case: #{ klass } ##{ method_name } is private?"
  puts "  #=> #{ klass.private_instance_methods.include?(method_name) }"
end

class Parent
  def self.private(*)
    puts "::private called from #{ self.name }"
    super
  end
end

class Child_1 < Parent
  private
  def child_instance_method; end
end

class Child_2 < Parent
  def child_instance_method; end
  private(:child_instance_method)
end

test_private_visibility(Child_1, :child_instance_method)
test_private_visibility(Child_2, :child_instance_method)

::private called from Child_1
::private called from Child_2

Case: Child_1 #child_instance_method is private?
  #=> false

Case: Child_2 #child_instance_method is private?
  #=> true
```

The source code makes it clear that the argument-less version is dependent upon a scope. So if I understand you correctly, the call to super in Parent is what would be calling this, and since that is in the scope of Parent, the originating call from Child_1 doesn't work because the scope doesn't match?

```
static VALUE
set_visibility(int argc, const VALUE *argv, VALUE module, rb_method_visibility_t visi)
{
    if (argc == 0) {
        rb_scope_visibility_set(visi);
    }
    else {
        set_method_visibility(module, argc, argv, visi);
    }
    return module;
}
```

#3 - 10/02/2019 10:33 PM - jeremyevans0 (Jeremy Evans)

- Status changed from Open to Closed

prebsch (Patrick Rebsch) wrote:

> The source code makes it clear that the argument-less version is dependent upon a scope. So if I understand you correctly, the call to super in Parent is what would be calling this, and since that is in the scope of Parent, the originating call from Child_1 doesn't work because the scope doesn't match?

Sorry for taking a long time to respond. Yes, the Child1.private call sets the private scope visibility in the Parent scope, because that is the current scope when super is called in Parent.private.

I'm going to close this now, as I don't think this behavior is a bug.