

Ruby trunk - Feature #12861

super in a block can be either lexically or dynamically scoped depending on how the block is invoked

10/21/2016 06:59 PM - bughit (bug hit)

Status:	Feedback
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	
Description	
<pre>class Class1 def self.foo 'foo' end def self.method1 'method1' end end class Class2 < Class1 def self.foo bar do super() end end def self.bar(&block) a = block.() define_singleton_method :method1, &block b = send(:method1) c = block.() [a, b, c] end end p Class2.foo # ["foo", "method1", "foo"]</pre>	
<p>It doesn't seem like a good idea for a given language construct to be either lexically or dynamically scoped, depending on how its surrounding block is invoked (which is not visible at the point of definition). I think it would be better if super were always lexically scoped, and a different keyword (dynamic_super) were always dynamically scoped</p>	

History

#1 - 10/24/2016 03:59 AM - shyouhei (Shyouhei Urabe)

bug hit wrote:

I think it would be better if super were always lexically scoped

Agreed but... What should then happen for inter-class block passing situation, like this?

```
class Foo
  def self.foo
    'foo@foo'
  end
end

class Bar < Foo
  def self.bar(&block)
    define_singleton_method :foo, &block
  end
end

class Baz
  def self.foo
```

```
Bar.bar do
  super
end
end
end
```

```
Baz.foo
```

#2 - 10/24/2016 04:20 AM - jeremyevans0 (Jeremy Evans)

I don't think this is a bug, the ability to change the scope of a block is just part of ruby, and arguably one of the things that makes ruby flexible and a joy to program in:

```
foo = proc do
  bar
end
```

```
foo.call # main.bar method call
```

```
Object.define_method(:foo, &foo)
Object.foo # Object#bar method call
```

```
Object.define_singleton_method(:foo, &foo)
Object.foo # Object.bar method call
```

```
a = 'a'
a.instance_eval(&foo) # a.bar method call
```

```
Object.class_eval(&foo) # Object.bar method call
```

The fact that super is sometimes lexical and sometimes dynamic follows naturally from how method calls in blocks are sometimes lexical and sometimes dynamic, depending on how the block is invoked.

In any case, changing super behavior to be purely lexical by default would likely break a ton of existing ruby code.

#3 - 10/24/2016 06:01 AM - bughit (bug hit)

Shyouhei Urabe wrote:

bug hit wrote:

I think it would be better if super were always lexically scoped

Agreed but... What should then happen for inter-class block passing situation, like this?

```
class Foo
  def self.foo
    'foo@foo'
  end
end

class Bar < Foo
  def self.bar(&block)
    define_singleton_method :foo, &block
  end
end

class Baz
  def self.foo
    Bar.bar do
      super
    end
  end
end

Baz.foo
```

you probably meant to call the defined :foo?

```
class Foo
  def self.foo
    'foo@foo'
  end
end
```

```

end

class Bar < Foo
  def self.bar(&block)
    define_singleton_method :foo, &block
    send(:foo)
  end
end

class Baz
  def self.foo
    Bar.bar do
      super()
    end
  end
end

Baz.foo

```

In your example, if `super` were always lexically/statically bound to the method, I suppose it would have to fail the same way `instance_eval` of the block (with `super`) fails when `self` is wrong (`self` has wrong type to call `super` in this context). But that would be an understandable error.

The solution would be to use a `dynamic_super`, which calls the `super` of the current method on the stack. Maybe it could even be a kernel method rather than a keyword. This way `super` the keyword would always be lexically/statically bound to the method, and a `call_super()` method for meta programming, would call the `super` of the dynamically determined current method on the stack

#4 - 10/24/2016 03:57 PM - bughit (bug hit)

bug hit wrote:

it would have to fail the same way `instance_eval` of the block (with `super`) fails when `self` is wrong (`self` has wrong type to call `super` in this context).

to expand on that, `super` normally binds to the method (class + method) lexically, so when it gets a `self` that's not compatible with its method binding it raises.

```

class Class1
  def self.bar(&block)
    instance_eval(&block)
  end
end

class Class2
  def self.foo
    Class1.bar do
      super
    end
  end
end

```

```
Class2.foo #self has wrong type to call super in this context: Class (expected #<Class:Class2>)
```

This makes sense. But in a `define_method` scenario (are there others?), the same `super` keyword suddenly starts binding to the method dynamically. Such overloading of core characteristics of a given construct seems wrong.

#5 - 10/24/2016 04:04 PM - bughit (bug hit)

Jeremy Evans wrote:

method calls in blocks are sometimes lexical and sometimes dynamic, depending on how the block is invoked.

Method resolution is never lexical, it is always relative to the current, dynamic `self`.

#6 - 10/24/2016 04:42 PM - jeremyevans0 (Jeremy Evans)

bug hit wrote:

Jeremy Evans wrote:

method calls in blocks are sometimes lexical and sometimes dynamic, depending on how the block is invoked.

Method resolution is never lexical, it is always relative to the current, dynamic `self`.

If method resolution is always dynamic, and super is directly related to method resolution, it would certainly be odd for super to always be lexical, right?

One could argue that super is currently always relative to the current, dynamic method. In your example:

```
def self.bar(&block)
  a = block.()      # super is called in method foo
  define_singleton_method :method1, &block
  b = send(:method1) # super is called in method method1
  c = block.()      # super is called in method foo
  [a, b, c]
end
```

I think ruby's current behavior makes sense. super should operate on the current method.

If you really want Class2.method1 to call Class1.foo, use super_method in Class2.foo to get the appropriate Method object, and call that:

```
class Class2 < Class1
  def self.foo
    meth = method(__method__).super_method
    bar do
      meth.call
    end
  end
end
```

#7 - 10/24/2016 05:35 PM - bughit (bug hit)

Jeremy Evans wrote:

One could argue that super is currently always relative to the current, dynamic method

Except it's not.

```
class Class1
  def self.foo
    'Class1::foo'
  end
end

class Class2 < Class1

  def self.store_block(&block)
    @block = block
  end

  def self.foo
    store_block do
      super
    end
  end

  def self.call_stored_block
    @block.()
  end

end

Class2.foo
Class2.call_stored_block # "Class1::foo"
```

When the block with super is invoked by Class2.call_stored_block, foo is not the current dynamic method, it's not even on the stack, and yet super calls foo because it is lexically bound to it. That's the current typical behavior of super, i.e. lexical method binding. The one exception is when super is in a block/proc invoked as a method.

#8 - 10/24/2016 09:03 PM - jeremyevans0 (Jeremy Evans)

- Assignee set to matz (Yukihiko Matsumoto)
- Tracker changed from Bug to Feature

bug hit wrote:

Jeremy Evans wrote:

One could argue that super is currently always relative to the current, dynamic method

Except it's not.

```
class Class1
  def self.foo
    'Class1::foo'
  end
end

class Class2 < Class1

  def self.store_block(&block)
    @block = block
  end

  def self.foo
    store_block do
      super
    end
  end

  def self.call_stored_block
    @block.()
  end

end

Class2.foo
Class2.call_stored_block # "Class1::foo"
```

When the block with super is invoked by Class2.call_stored_block, foo is not the current dynamic method, it's not even on the stack, and yet super calls foo because it is lexically bound to it. That's the current typical behavior of super, i.e. lexical method binding. The one exception is when super is in a block/proc invoked as a method.

When the block with super is invoked by Class2.call_stored_block, foo is the current method at that point, according to `__method__`. This is simple to see by changing Class1.foo to raise an exception in your example, using `__method__` from inside the block:

```
class Class1
  def self.foo(v)
    raise v
  end
end

class Class2 < Class1

  def self.store_block(&block)
    @block = block
  end

  def self.foo
    store_block do
      super(__method__.to_s)
    end
  end

  def self.call_stored_block
    @block.()
  end

end

Class2.foo
Class2.call_stored_block
# super_test.rb:3:in `foo': foo (RuntimeError)
#       from super_test.rb:15:in `block in foo'
#       from super_test.rb:20:in `call_stored_block'
#       from super_test.rb:26:in `<main>'
```

Notice how the backtrace states "block in foo" (in ruby 1.8.7, it just shows "foo"), and you can see that `__method__` in the block is :foo, show by the exception message.

It's already possible to get your desired behavior via `super_method`, changing ruby's behavior in regards to `super` will break a large amount of existing code, and the way `super` currently works in ruby makes sense (calling the `super` of the current `__method__`). This behavior dates back to at least ruby 1.8.7.

This is a request for a language behavior change, not a request for a bug fix. I'm changing this from Bug to Feature, and assigning to matz to make the decision.

#9 - 10/24/2016 09:44 PM - bughit (bug hit)

`super` is in sync with `__method__` because they are designed to be in sync, `__method__` called from a block typically returns the enclosing method of the block but not always, when the block is invoked as a method, `__method__` returns the method represented by the block.

This argument does not alter the fact that `super` called from a block is usually lexically bound to the enclosing method but not always. I think that's not ideal, something as significant as lexical vs dynamic binding should be fixed for a given concept/construct.

#10 - 12/21/2016 02:22 PM - shyouhei (Shyouhei Urabe)

We looked at this issue at today's developer meeting.

I think attendees had an assumption that changing behaviour of `super` is too drastic. This feature (if added) should be called something new. So there are rooms for `dynamic_super`. For lexical one, I like the idea. But Matz was wondering if such lexically-scoped `super` would actually get used.

#11 - 12/21/2016 04:24 PM - matz (Yukihiro Matsumoto)

- Status changed from Open to Feedback

I don't see the real-world problem except for a bit of complexity behind. If we distinguished lexical `super` and dynamic `super`, it would confuse more users than the current behavior, I think.

Matz.

#12 - 12/21/2016 05:20 PM - bughit (bug hit)

Yukihiro Matsumoto wrote:

I don't see the real-world problem except for a bit of complexity behind. If we distinguished lexical `super` and dynamic `super`, it would confuse more users than the current behavior, I think.

Matz.

```
def self.foo
  bar do
    super()
  end
end
```

I wouldn't say it's a major practical problem, I just think that it's a flaw that you can't tell what this `super` is going call, as it depends on how `bar` is invoked. If there were distinct dynamic and lexical `supers`, I'd definitely prefer those to avoid ambiguity, making the code clearer and more self documenting