Ruby - Feature #16276

For consideration: "private do...end" / "protected do...end"

10/23/2019 07:49 PM - adh1003 (Andrew Hodgkinson)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Private or protected declarations in Ruby classes are problematic. The single, standalone public, private or protected statements cause all following methods - *except* "private" class methods, notably - to have that protection level. It is not idiomatic in Ruby to indent method definitions after such declarations, so it becomes at a glance very hard to see what a method's protection level is when just diving into a piece of source code. One must carefully scroll *up* the code searching for a relevant declaration (easily missed, when everything's at the same indentation level) or have an IDE sufficiently advanced to give you that information automatically (and none of the lightweight editors I prefer personally have yet to support this). Forcibly indenting code after declarations helps, but most Ruby developers find this unfamiliar and most auto-formatters/linters will reset it or, at best, complain. Further, the difficulty in defining private *class* methods or constants tells us that perhaps there's more we should do here - but of course, we want to maintain backwards compatibility.

On the face of it, I can't see much in the way of allowing the public, private or protected declarations to - *optionally* - support a block-like syntax.

```
class Foo

  # ...there may be prior old-school public/private/protected declarations...

  def method_at_whatever_traditional_ruby_protection_level_applies
    puts "I'm traditional"
  end

  private do
    def some_private_instance_method
      puts "I'm private"
    end

    def self.some_private_class_method
      puts "I'm also private - principle of least surprise"
    end

    NO_NEED_FOR_PRIVATE_CONSTANT_DECLARATIONS_EITHER = "private"
  end

  def another_method_at_whatever_traditional_ruby_protection_level_applies
    puts "I'm also traditional"
  end

end
```

My suggestion here confines all public do...end, protected do...end or private do...end protections strictly to the confines of the block alone. Outside the block - both before and after - traditional Ruby protection semantics apply, allowing one to add new block-based protection-enclosed method declarations inside any existing code base without fear of accidentally changing the protection level of any methods defined below the new block. As noted in the pseudocode above, we can clean up some of the issues around the special syntax needed for "private constants", too.

I see a lot of wins in here but I'm aware I may be naïve - for example, arising unanswered questions include:

- Is the use of a block-like syntax making unwarranted assumptions about what the Ruby compiler can do during its various parsing phases?
- Does the use of a block-like syntax imply we should support things like Procs too? (I *think* probably not - I see this as just syntax sugar to provide a new feature reusing a familiar idiom but without diving down any other rabbit holes, at least not in the first implementation)

I've no idea how one would go about implementing this inside Ruby Core, as I've never tackled that before. If someone is keen to

pick up the feature, great! Alternatively, if a rough idea of how it *might* be implemented could be sketched out, then I might be able to have a go at implementation myself and submit a PR - assuming anyone is keen on the idea in the first place :-)

Related issues:

Is duplicate of Ruby - Feature #7019: allow `private` and `protected` keyword...	**Rejected**

History

#1 - 10/23/2019 09:17 PM - shevegen (Robert A. Heiler)

In general I agree with the proposal, or at the least with the basic gist of it, e. g.:

```
private {
  do_stuff
}

public {
  do_stuff
}
```

I do not know how old private/public distinction is in ruby but I think matz added this very early on, perhaps even in the first public releases. I remember in the old pickaxe, it was explained that "private" and "public" are sort of typically used as "toggling the state" that is - you write code, and then you may add e. g. "private", and write all the code that is, well, private.

Personally I do not use the private and public distinction. This is mostly in the way how you (as a ruby user) may want to use ruby. Some prefer a more java-centric OOP model; perhaps this was a use case as to why .public_send() was added by matz at a later time (I think .send() was much older than .public_send() but I do not know when the latter was added). In my opinion, using .send and avoiding private, is more "idiomatic", but this depends on the point of view. Personally I like the self/smalltalk view on OOP more than the C++/java view. Ruby has its own view, sort of; while I think the primary focus is on OOP, ruby has always been multi-paradigm. I think matz likes to play with ideas and concepts. :)

But anyway, back to the suggestion. The reason why I am +1 for this proposal, even though I personally do not use that distinction really, is because I actually would find it convenient. I don't know if this may create incompatibilites or not, but purely from a convenience point of view, I think that would be a good idea.

There are a few parts I disagree with your proposal though. For example, you wrote:

> to have that protection level

I am not putting too emphasis here really, so excuse my nit-picking, but IMO ruby does not have a strong "protection" level because I think it would not be completely well aligned with ruby's philosophy of flexibility, more-than-one-way and in general letting people decide what they want to use it, and how. (Compare this to python in many ways, which pursues a different model.) For example, we can use .send() to "get access" to literally anything; we can obtain instance variable and change them if we want to. I love this dynamic nature. Others may dislike it, if they think that ruby should be less dynamic. When you have "two sides", one saying that a particular use case is bad, the other saying that it is good, it is difficult to align them with the same thought, since these thoughts are orthogonal and conflicting. I'd always reason in favour of .send() for example and never use .public_send() myself. :)

But I am a bit nit-picking here, so don't mind this comment too much.

> It is not idiomatic in Ruby to indent method definitions after such declarations

I would not use the word "idiomatic", but I actually agree with you for another reason. Indenting code can be a bit annoying. Typically most people may tend to use two spaces per indent level. I actually ignore that when I define classes in modules, e. g.:

```
module Foo
module Bar
  class Cat
```

^^^ module Bar "should" be two spaces to the right, but I much prefer it to the very left, and then using just one indent for class. Now - I don't expect many other people to use this, but I liked it; and while this is not completely related to the same use case and the description of the feature here, I concur with you here. I actually never indent when private/public is used. Actually, I do sometimes use private, but I then use something crazy:

```
def foo
end; private :foo
```

This is more work but ... I like that I don't have to indent to the right. :P

So for this and similar reason, I actually agree with your premise mostly, even though I do not use the same argument(s).

> so it becomes at a glance very hard to see what a method's protection level is when just diving into a piece of source code.

Yup - I sort of agree with you here. Even though I guess we both may write ruby code differently. :)

Makes sense to me what you write in this context.

> As noted in the pseudocode above, we can clean up some of the issues around the special syntax needed for "private constants", too.

I have no problem either way but I think "constants" is a bit of a misnomer in general. The ruby philosophy here is more that ruby allows you to change it if you want to. Which I think makes sense, oddly enough; at the same time, I remember apeiros wondered about the name constant, and I sort of agree because people may assume that "a constant may never change". Which is not wrong, either. Just more-than-one-way to look at something (perhaps it should have another name than constant, so people don't get confused).

This is also a bit strange when it comes to "private" constants in ruby. Can these be changed? Should these be changed? Are there even really "private" constants in ruby?

I actually really don't know. It has been rare that I used constants "dynamically" and changed them. These days I tend to use @foo toplevel instance variable more, like:

```
module Foo
  @bar = {}
```

and work from here IF it has to be dynamic (aka data that is to be changed for some reason or another).

> Does the use of a block-like syntax imply we should support things like Procs too?

Aren't blocks in ruby in general procs too? But anyway, to answer the question - I don't think every method supports blocks uniformly strongly, meaning that some methods make use of blocks more, and other methods don't. To me blocks are more like an additional argument that is more flexible (usually). Sort of you can use it if you want to - but you don't have to. So from here, I don't really see a problem if private/public were to have a {} block variant.

There may perhaps be other issues, though, such as backwards incompatibility. I guess this all has to be discussed and of course you have to ask matz about the design consideration for private/public. Perhaps there was a reason why the block variant was not considered or used or there were some other problems.

#2 - 10/23/2019 09:44 PM - shevegen (Robert A. Heiler)

Actually I should clarify some of my statements a bit more; I'll do it in a terse add-on.

- I believe that many ruby users may actually NOT indent when using private/public.

They may write code like this:

```
class Foo
  def bla
```

```
  end
  private
  def hop
  end
end
```

I believe in these cases, the suggestion here may possibly be of less value to this style, because they may prefer the current style more than any alternative.

An alternative may be:

```
class Foo
  def bla
  end
private
  def hop
  end
end
```

This is a bit "off" because the indent to private is one level to the left, but I see this style sometimes as well. IMO I think an additional requirement for this suggestion, although I think it is an ok-suggestion, should be how many ruby users may want to use e. g. private {} in the first place. If there are only a very few then perhaps this may not be worth to add/change it (again, not assuming anything either way, but I think the usability and practical use cases should be considered too).

- I am mostly neutral on the whole issue actually, only a very slight +1 support. As said I probably may not need this, but this of course does not exclude the possibility that others may use/want this. :)

It may be helpful if other ruby users could comment on the issue too in the coming days/weeks, including people from the core team if they write lots of ruby code in general.

#3 - 10/24/2019 01:52 AM - shyouhei (Shyouhei Urabe)

- Is duplicate of Feature #7019: allow `private` and `protected` keywords to take blocks added

#4 - 10/24/2019 01:58 AM - duerst (Martin Dürst)

shevegen (Robert A. Heiler) wrote:

> Actually I should clarify some of my statements a bit more; I'll do it in a terse add-on.
>
> - I believe that many ruby users may actually NOT indent when using private/public.
>
> They may write code like this:
>
> ```
> class Foo
>   def bla
>   end
>   private
>   def hop
>   end
> end
> ```

This simply is sloppy coding.

> I believe in these cases, the suggestion here may possibly be of less value to this style, because they may prefer the current style more than any alternative.
>
> An alternative may be:
>
> ```
> class Foo
>   def bla
>   end
> private
>   def hop
>   end
> end
> ```

This is what should be used, or an intermediate:

```
    class Foo
      def bla
      end
     private
      def hop
      end
    end
```

I hope rubocop (and similar tools) produces a warning fro the first example above, and allows the style in the second or third example.

#5 - 10/24/2019 02:19 AM - shyouhei (Shyouhei Urabe)

- C++: There are private, but no private {}
- Java: There are private, but no private {}
- Scala: There are private, but no private {}
- Kotlin: There are private, but no private {}
- Rust: Everything are private by default, there is pub instead. But there is no pub {}

Correct me if I'm wrong. But it seems the idea of "private with a block" isn't seen anywhere.

#6 - 10/24/2019 06:29 AM - marcandre (Marc-Andre Lafortune)

FWIW, my personal style has evolved to using private inline:

```
class Foo
  def public
  end

  private def some_private_instance_method
    puts "I'm private"
  end

  private def also_some_private_instance_method
    puts "I'm also private"
  end
end
```

I don't believe I would use private {}, but I'm not against the proposal, in particular since it doesn't add much cognitive load.

#7 - 10/29/2019 08:38 PM - adh1003 (Andrew Hodgkinson)

shevegen (Robert A. Heiler) wrote:

> > As noted in the pseudocode above, we can clean up some of the issues around
> > the special syntax needed for "private constants", too.
>
> I have no problem either way but I think "constants" is a bit of a misnomer
> in general. The ruby philosophy here is more that ruby allows you to change
> it if you want to.

Agree with all you said and appreciate the detailed feedback - but on this, and in many ways on the use of send, those are workarounds for "I really know what I'm doing". Private methods are not normally callable by conventional syntax, and constants pretty much are constants; attempts to redefine them raise warnings, so although it's possible it is, again, inadvisable and private scope constants *are* a thing (and are useful).

The reason for public/protected/private is not just about what "can be done". It's about the contract you're drawing within your class or module, to which clients of that class or module must adhere. Public things are for anyone; protected things for subclasses; private things are implementation. This is vital - no mere decoration - it goes to the very core of OOP and software engineering. Calling send to hack into a private method means the caller is breaking the contract with the target entity and thus risks its own implementation failing at any time, since the target entity is freely at liberty to change anything in its private implementation without any prior warning.

duerst (Martin Dürst) wrote:

> ```
> class Foo
>   def bla
>   end
>   private
>   def hop
>   end
> end
> ```
>
> This simply is sloppy coding [...left-indent "private"...]

Coding style wars wage often. Some of it is functional and objective, but much of it is aesthetic and subjective. The complaint about the coding style does not, I think, really change whether or not we might want to tighten the behaviour of the public/protected/private declarations using block-like syntax in a manner that would be 100% backwards compatible with all existing code (since the no-block syntax would still be there and not be deprecated).

shyouhei (Shyouhei Urabe) wrote:

- C++: There are private, but no private {}
- Java: There are private, but no private {}
- Scala: There are private, but no private {}
- Kotlin: There are private, but no private {}
- Rust: Everything are private by default, there is pub instead. But there is no pub {}

> Correct me if I'm wrong. But it seems the idea of "private with a block" isn't seen anywhere.

In those languages it's impossible (or extremely difficult, via complex reflection programming) to call a private method as a client of the class, but in Ruby you just use send - not that you usually should. In most of those languages it's impossible (or again extremely difficult) to redefine a constant, but in Ruby you can do so easily (const_defined?, remove_const, const_set) - again, not that you should. All of those languages are statically, strongly typed, but Ruby is not. Things like C++ or Java are surely (in general) bad (or at best, difficult) places to look for syntax to copy, since they're generally hopelessly over complicated and require extremely heavy IDE support to make any kind of sense out of a typical code base. Ruby is typically far simpler and clearer; that's part of the reason why it was made in the first place.

Ruby is its own language. Just because other languages don't do it, does not mean Ruby would not benefit. And again, this is an *extension* to the existing syntax, not a replacement.

#8 - 10/29/2019 09:17 PM - Eregon (Benoit Daloze)

I kind of like this idea as it would make it clear which methods are private by indentation.
private alone is indeed hard to notice as soon as there are a few methods in the class.

OTOH, using this to change a method from private to public or vice-versa would cause a lot of indentation changes, so it's a double-edged sword.

private def moves the def, the method name and parameter definitions quite a lot on the right side, which I find not so nice.
Repeating private def often is also quite verbose.
Also worth noting that private def actually defines two methods, one public and then a copy of it as private, overriding the public one in the method table.

#9 - 10/29/2019 09:51 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> Also worth noting that private def actually defines two methods, one public and then a copy of it as private, overriding the public one in the method table.

I don't think this is true. private :method only adds an entry (zsuper method) to the method table if the method is not defined in the current class (i.e. it is defined in a superclass or included module). If the method is defined directly in the class, it just updates the visibility flag on the existing method table entry. Since def always defines in the current class, private def should not be defining two methods.

Regarding the feature, I think if we didn't have the current scope visibility behavior when calling private with no arguments, this would be a reasonable approach for implementing the behavior. However, I don't think it is worth adding as an alternative approach.

#10 - 10/30/2019 01:42 AM - shyouhei (Shyouhei Urabe)

Yes, I agree we don't copy C++ / Java. What I wonder is *any* other language who have such syntax. Swift? no. Python? no. TypeScript? no. AFAIK no one on this planet have such thing so far. It seems to me that IDEs are not the answer to this phenomena.

#11 - 10/30/2019 07:08 PM - adh1003 (Andrew Hodgkinson)

shyouhei (Shyouhei Urabe) wrote:

> Yes, I agree we don't copy C++ / Java. What I wonder is *any* other language who have such syntax. Swift? no. Python? no. TypeScript? no. AFAIK no one on this planet have such thing so far. It seems to me that IDEs are not the answer to this phenomena.

And yet every language has unique syntax features that aren't in any other language, else, what would be the point of any of them.

There was a first language to implement a syntax to say "call this method, unless the target is nil, in which case return nil". Nobody else would've had that first. And then we have Objective C sat out on its own, doing what I think is an admittedly verbose, but exceptionally readable message send (i.e. method call - but *actually* messages, with Mach) using its 'square bracket' notation and named parameters, and with no need for any sort of special syntax for "call this method, allowing target to be nil" since sending a message to a null entity always results in null by default - all of which leads to very easy to read and write code.

Procs/lambdas/closures were uncommon things indeed in many mainstream languages when Ruby came along with them all over the place. Now most languages have something along those lines. Being a pioneer didn't stop Ruby then, and IMHO it shouldn't stop it now. There seems to be a general agreement that this proposed extension would be a good thing, so if the best objection we can come up with is "most other languages don't do it", we honestly might as well pack up and go home; because that would mean Ruby never again ever implements anything other languages don't commonly do.

#12 - 10/30/2019 08:04 PM - jeremyevans0 (Jeremy Evans)

adh1003 (Andrew Hodgkinson) wrote:

> There seems to be a general agreement that this proposed extension would be a good thing

I'm not seeing general agreement that this would be a good thing to add. In this ticket:

@shevegen : mostly neutral, very slight #+1
@duerst (Martin Dürst) : no recommendation
@shyouhei (Shyouhei Urabe) : no recommendation
@marcandre (Marc-Andre Lafortune) : neither for nor against
@Eregon (Benoit Daloze) : kind of like, but double-edged sword
@jeremyevans0 (Jeremy Evans) : not worth adding

In #7019, it appeared that @shyouhei (Shyouhei Urabe), @drbrain (Eric Hodel), @headius (Charles Nutter), @ko1 (Koichi Sasada), and @mame (Yusuke Endoh) didn't think it was worth adding.

Where are you seeing general agreement that this proposed extension would be a good thing?

#13 - 10/30/2019 09:14 PM - alanwu (Alan Wu)

For the following:

```
class Foo
  private do
    o = Object.new
    def o.hello; end
  end
end
```

Would o.hello be private? Also, I assume you want the visibility change to
be fiber local, to follow the principle of least suprirse. Since this new form
adds no new capability and requires at least a new fiber local to implement,
I don't think it carries its weight.

#14 - 10/31/2019 05:03 AM - mame (Yusuke Endoh)

I have no strong opinion against this issue. In #7019, I just said that it was too late to introduce the feature into Ruby 2.0 because I was the release manager for 2.0.

I'm curious about Rails developers' opinion. I hear that they are using a peculiar indent rule like:

```
class foo
  def foo
  end

  private
    def bar
    end
end
```

Compared to this style, private do ... end is better, IMO. If they will use the new style, it might be worth considering.

Personally I like the private def style or the plain old no-indent style, though.

#15 - 10/31/2019 05:27 AM - duerst (Martin Dürst)

adh1003 (Andrew Hodgkinson) wrote:

> duerst (Martin Dürst) wrote:
>
> ```
> class Foo
>   def bla
>   end
>   private
>   def hop
>   end
> ```

```
end
```

> This simply is sloppy coding [...left-indent "private"...]

> Coding style wars wage often. Some of it is functional and objective, but much of it is aesthetic and subjective. The complaint about the coding style does not, I think, really change whether or not we might want to tighten the behaviour of the public/protected/private declarations using block-like syntax in a manner that would be 100% backwards compatible with all existing code (since the no-block syntax would still be there and not be deprecated).

I'm not advocating any particular coding style. There are many coding styles that make it clear that private applies to a range of methods. There are not many coding styles where that's not clear. The choice between the former and the latter is a functional choice.

The fact that the range of private and friends can be expressed with many reasonable coding styles doesn't mean that we might not want your proposed feature, but it means that you proposed feature is far away from a "need to have", and therefore has low priority.

#16 - 10/31/2019 01:35 PM - Dan0042 (Daniel DeLorme)

I don't see why all the opposition to this. It's a very simple, very intuitive and clear syntax. It fits well with the rest of ruby and the usage of blocks in general.

alanwu (Alan Wu) wrote:

```
class Foo
  private do
    o = Object.new
    def o.hello; end
  end
end
```

> Would o.hello be private? Also, I assume you want the visibility change to be fiber local, to follow the principle of least suprirse.

I think @adh1003 was pretty clear in describing the feature, so that specific piece of code above is strictly equivalent to this:

```
class Foo
  private
    o = Object.new
    def o.hello; end
end
```

Therefore o.hello is public.

Also I'd like to ask how fibers are relevant to this? When the private method toggles the visibility state of the current class/module, does that have anything to do with fibers?

#17 - 10/31/2019 03:10 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> Also I'd like to ask how fibers are relevant to this? When the private method toggles the visibility state of the current class/module, does that have anything to do with fibers?

```
class A
end

Thread.new do
  class A
    def pub; end
  end
end

class A
  private do
    def priv; end
  end
end
```

Depending on thread timing, it is theoretically possible for pub to be private and not public if the visibility stored in the class and not in some sort of scope. This example uses a thread, but the same basic issue applies to fibers.

#18 - 10/31/2019 05:43 PM - Dan0042 (Daniel DeLorme)

Ok, but how is this different from the regular syntax? The exact same issue applies to this:

```
class A
end

Thread.new do
  class A
    def pub; end
  end
end

class A
  private
  def priv; end
end
```

If there's no problem with the above, there's no reason why the proposed block form would have any problem either right? In other words the thread/fiber issue is irrelevant to the current proposal.

#19 - 10/31/2019 06:08 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> Ok, but how is this different from the regular syntax? The exact same issue applies to this:
>
> ```
> class A
> end
>
> Thread.new do
>   class A
>     def pub; end
>   end
> end
>
> class A
>   private
>   def priv; end
> end
> ```

Are you sure? You can test this by using queues to synchronize:

```
class A
end

Q1 = Queue.new
Q2 = Queue.new

Thread.new do
  class A
    Q2.pop
    def pub; end
    Q1.push nil
  end
end

class A
  private
  def priv; end
  Q2.push nil
  Q1.pop
end

A.public_instance_methods(false) # => [:pub]
```

> If there's no problem with the above, there's no reason why the proposed block form would have any problem either right? In other words the thread/fiber issue is irrelevant to the current proposal.

There is no problem with fibers/threads currently, because the visibility is stored in the scope (not in the class itself). You stated When the private method toggles the visibility state of the current class/module, implying this would implemented with a visibility flag on the class/module. You cannot have visibility stored in the class/module and handle fibers/threads properly without making the class-level visibility information fiber/thread-aware.

#20 - 10/31/2019 07:11 PM - Dan0042 (Daniel DeLorme)

> There is no problem with fibers/threads currently, because the visibility is stored in the scope (not in the class itself). You stated When the private method toggles the visibility state of the current class/module, implying this would implemented with a visibility flag on the class/module. You cannot have visibility stored in the class/module and handle fibers/threads properly without making the class-level visibility information fiber/thread-aware.

My apologies for the imprecise wording. I wasn't aware of the full technical details of how private/public are implemented. My point was simply that private *with* block is not technically different from private *without* block in terms of how the visibility state is handled. The only difference is restoring the previous state at the end of the block. Therefore there's no issue with threads/fibers.

#21 - 10/31/2019 07:30 PM - alanwu (Alan Wu)

The extra complication comes from this part of the original proposal:

```
private do
  def self.some_private_class_method
    puts "I'm also private - principle of least surprise"
  end
end
```

This is different from the normal blockless private:

```
class A
  private
  def self.hello; end
  p self.singleton_class.public_instance_methods(false) # [:hello]
end
```

So this block form can't be implemented in terms of the old blockless form.
This is why I asked about def o.foo; end. I don't think whether to special
case def self.foo; end inside the visibility block is a trivial decision to make.

Besides the hairy semantics problems, every time you define a method on
a singleton class, you still need to somehow remember that you are within a
private do block in a way that doesn't leak to other fibers.

All this complication for a feature that's mostly for looks doesn't seem worth it.

#22 - 11/01/2019 01:54 AM - Dan0042 (Daniel DeLorme)

Oh! Looks like I missed that part of the proposal! Now the def o.hello; end question makes a lot more sense. Sorry, Alan.

If we break this down a bit, this proposal can potentially set the visibility (within the block) for

1. instance methods: seems like an intuitive, obvious, and low-cost idea
2. constants: changes the semantics of private but seems useful enough to warrant it
3. class methods: changes the semantics of private, seems mostly intuitive, but maybe just too hard to implement to be worth it. It may be possible if the visibility is stored in the scope as Jeremy said. Would have to check feasibility with nobu.
4. arbitrary singleton methods: like above, but it's doubtful this is desirable

#23 - 11/01/2019 02:42 AM - shyouhei (Shyouhei Urabe)

I can think of other cursed usages of private taking a block.

```
class Foo
  Bar = proc do
    Baz = 1
    def foo
      Baz
    end
  end
end

class Bar
  private(&Foo::Bar)
end
```

Should what happen?

- Possibility #1: defines Foo#foo and Foo::Baz. Easiest to implement, however very counter-intuitive because random methods of random classes can be (re)defined at any random calls of private.
- Possibility #2: defines Bar#foo and Foo::Baz. Surprisingly, this is how class_eval works today. Also very counter-intuitive.
- Possibility #3: renders SyntaxError. This requires a massive rewrite of our parser. Theoretically possible but not in practice.

Redefinition of an existing public method (or a constant) as a private one should break other parts of the program. I now think a private with a block, especially those who generated elsewhere and passed as an & parameter, is dangerous.

#24 - 11/07/2019 12:02 AM - adh1003 (Andrew Hodgkinson)

shyouhei (Shyouhei Urabe) wrote:

> I can think of other cursed usages of private taking a block.

Then it is fortunate, is it not, that this is *not what I am proposing*. What I said was, I thought very clearly:

> ...support a block-like syntax...

...with some very specific limitations which were precisely due to the horrible rat's nest if this were truly a block, a yieldable thing, rather than just syntax sugar.

I'm just asking for a very Ruby-like, clear, simple syntax extension that makes it obvious when a bunch of things are collected inside a specific visibility scope, in passing cleaning up nasty messes like "private_class_method" and solving a couple of (minor) formatting wars in passing.

#25 - 11/07/2019 05:09 AM - mame (Yusuke Endoh)

adh1003 (Andrew Hodgkinson) wrote:

> shyouhei (Shyouhei Urabe) wrote:
>
> > I can think of other cursed usages of private taking a block.
>
> Then it is fortunate, is it not, that this is *not what I am proposing*.

No, not fair enough. It is what you are proposing, even if you didn't intend. We need to care about all the possibilities as far as we can. That's the language design.

@shyouhei (Shyouhei Urabe), good catch. If I need to pick up your possibilities, I like #1 or #2. But now I'm a bit against the proposal. It brings a small advantage and relatively larger
complexity then I expected.

#26 - 11/07/2019 02:43 PM - Dan0042 (Daniel DeLorme)

- Possibility #3: renders SyntaxError. This requires a massive rewrite of our parser. Theoretically possible but not in practice.

I'm a bit curious about this. My understanding is that a Proc object is not created for every block. So it should be possible to know that private{ } is called with a block while private(&block) is called with a Proc (and raise an error in the latter case). In invoke_block_from_c_bh (from vm.c) I can see switch (vm_block_handler_type(block_handler)) which seems to do exactly that: telling apart the types of block. So in the case of private, if vm_block_handler_type returns block_handler_type_proc, it would make sense to me to raise an error. Or quite possibly I'm misunderstanding something about how this all works.

#27 - 11/19/2019 04:27 AM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> - Possibility #3: renders SyntaxError. This requires a massive rewrite of our parser. Theoretically possible but not in practice.
>
> I'm a bit curious about this. My understanding is that a Proc object is not created for every block. So it should be possible to know that private{ } is called with a block while private(&block) is called with a Proc (and raise an error in the latter case).

It may be possible to know at runtime whether a literal block is passed or not (if not, that is probably an easier change to make). However, you can't really know at parse time (SyntaxError is raised at parse time). Example:

```
class A
  class << self
    alias priv private
  end

  priv do
    def
  end
```

```
end
```

> In invoke_block_from_c_bh (from vm.c) I can see switch (vm_block_handler_type(block_handler)) which seems to do exactly that: telling apart the types of block. So in the case of private, if vm_block_handler_type returns block_handler_type_proc, it would make sense to me to raise an error. Or quite possibly I'm misunderstanding something about how this all works.

Since you seem to be in doubt, you should attach a debugger and call with a literal block and call with a block passed via &, and see what the difference is.

adh1003 (Andrew Hodgkinson) wrote:

> I'm just asking for a very Ruby-like, clear, simple syntax extension that makes it obvious when a bunch of things are collected inside a specific visibility scope, in passing cleaning up nasty messes like "private_class_method" and solving a couple of (minor) formatting wars in passing.

What you consider making obvious, others may consider clouding the difference between instance methods of the class and singleton methods on the class.

private_class_method is just a shortcut. Calling it a nasty mess implies define_singleton_method is also a nasty mess. If you always define methods as regular methods, you don't need private_class_method:

```
class A
  class << self
    def public_singleton_method
    end

    private

    def private_singleton_method
    end
  end

  def public_instance_method
  end

  private

  def private_instance_method
  end
end
```