Ruby trunk - Bug #15732

Strict mode

03/27/2019 10:24 PM - localhostdotdev (localhost .dev)

Status:	Open		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:		**Backport:**	2.4: UNKNOWN, 2.5: UNKNOWN, 2.6: UNKNOWN

Description

A lot of issues could be easily prevented with a strict mode, for instance:

- Passing a block to a method that doesn't accept blocks could raise an exception, e.g. methods would be required to explicitly ask for a block (def a(..., &block))
- $1, $2, etc. variables when matching with a regex would not be created
- Could throw error when re-assigning constant

And maybe more controversial:

- Using reversed keywords as argument names
- Using method name as argument name
- Using ambitious class/constant/method names (e.g. class A; end; class B; class A; def me; A; end; end

I'm sure there is more, using things rubocop does for instance: https://github.com/rubocop-hq/rubocop/blob/master/manual/cops.md

I could be implemented as a magic header, e.g. # strict_mode: true.

This is obviously a similar idea from Javascript's strict mode:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

History

#1 - 03/27/2019 11:04 PM - jeremyevans0 (Jeremy Evans)

localhostdotdev (localhost .dev) wrote:

> A lot of issues could be easily prevented with a strict mode, for instance:
>
> - Passing a block to a method that doesn't accept blocks could raise an exception, e.g. methods would be required to explicitly ask for a block (def a(..., &block))

This is being discussed in #15554.

- $1, $2, etc. variables when matching with a regex would not be created

Not sure why these would be an issue (performance?). You can use String#match? if you want to avoid it.

- Could throw error when re-assigning constant

There is already a warning for this, is there a reason it would need to be an error? You can already turn it into an error:

```
def Warning.warn(s)
  raise s if s.match?(/: warning: already initialized constant /)
  super
end
```

> And maybe more controversial:
>
> - Using reversed keywords as argument names

Already a SyntaxError, assuming you meant method argument name:

```
def x(begin)
end
```

Maybe you meant method name and not method argument name? Sometimes the method name that makes the most sense is a reserved keyword. Considering that there are core ruby classes that do this (Range#begin, Range#end), I'm not sure how this would work.

- Using method name as argument name

Such as this?:

```
def foo(foo)
  foo
end
```

I don't see this pattern used much, but it doesn't seem to cause a problem. Any reason why you would want to disallow argument names that are the same, but not local variables that are the same?:

```
def foo(bar)
  foo = bar
  foo
end
```

- Using ambitious class/constant/method names (e.g. class A; end; class B; class A; def me; A; end; end

I'm guessing you meant ambiguous? This actually isn't ambiguous, you just need to know how constant lookup is preformed.

> I'm sure there is more, using things rubocop does for instance: https://github.com/rubocop-hq/rubocop/blob/master/manual/cops.md
>
> I could be implemented as a magic header, e.g. # strict_mode: true.

This proposed collection of behavior changes added by a strict mode seems mostly arbitrary to me. And I don't think we want to add more magic headers if we can avoid it. Some of these things happen at run time and not parse time, and a magic header doesn't work well for that. For example:

a.rb:

```
# strict_mode: true
class B
  class A
    def me
      A
    end
  end
end
```

b.rb:

```
require_relative 'a'
A = B
A::A.me{|x| x}
```

Does this trigger an error for "ambiguous" constants, even though file b.rb doesn't use strict mode and when a.rb was parsed, the constant wasn't "ambiguous"? How about for passing a block to the me method that doesn't accept a block?

#2 - 03/28/2019 07:53 AM - shevegen (Robert A. Heiler)

This suggestion confuses me quite a bit. I am not sure of the use case; or perhaps I don't quite understand it.

I think what you essentially suggest is to be able to run/interprete any given .rb file with additional restrictions in place, yes? So a bit like the old $SAFE settings, if I understood it correctly.

On the other hand ... hmm. Not creating $1 when a regex is used - why would that have to be done? The $1 etc... are already different to other normal $ variables - they are very volatile. I can not think of any situation where they may be a problem?

Disallowing re-assigning constants ... I think this is largely any personal preference, so I can understand that part a little. Some people may prefer that constants could not be changed in ruby, so I am not at all against the possibility. Note that I am not among those who have a problem with it. I think most people who come from the point of view that changing

constants is bad, approach the issue from the wrong point of view, similar with the older File.exists? versus File.exist situation. One partial application of ruby's philosophy is to let you do what you want to, when you want to (at the least in many situations). So the issue, in my opinion, is not that a "constant" can be changed - I guess the issue is that people have in their mind the notion that a "constant is a constant and does never change"; so when you don't focus on the word constant that much, perhaps these people may suddenly no longer have the same strong opinion. Could give another name to it. :) But as said, I also understand the other argument, even though I have no problem with constant; the only comment I remember having made in the past is that it wuold be nice if we could have more control over warnings issued by ruby in the future (but it may also not be a huge priority; just may be nice if we could have more "fine tuned control" eventually).

Passing a block to a method that doesn't "accept blocks" ... raising an exception .... hmmmmmmm. Every method accepts a block without one having to do something. That's part of what makes ruby great. If you don't need it, then don't make use of the pased block? I usually use something like "if block_given?" ... and then next line ... "yielded = yield", or something funny like that; easier for me to continue to work with a variable. :P

I am not sure why anyone may want to prevent it from happening in ruby.

Using reversed keywords as argument names ... actually, I once had a use case for this, e. g. when I was autogenerating HTML and CSS. I wanted to add a parameter called "class", for a CSS class, but this was not possible. My solution was to simply give it a slightly longer name, aka "css_class" and that was fine (I also added "css_style", so I could differ between a CSS class, and a CSS style attached to a given HTML tag). So from this point of view, I can partially understand the use case - buuuuuuuut I am also not sure if this is so important or good to have.

I also don't think inspiration should be drawn from JavaScript because the "design" of JavaScript makes me quite angry in general. I often compare JavaScript to PHP and I honestly often can not tell the difference as to which one is worse. This is from a design point of view, mind you - of course you can still create useful software in both languages. I myself just prefer better design and ruby has the better design. I am even objective here, despite my bias in favour of ruby. :) I know that because I used PHP a lot in the past and while I was productive in it, comparing the PHP code I wrote with the ruby code is an instant win for ruby. It's just no real comparison, ruby "beats" PHP here easily.

I am not at all in principle against some of the suggestions given here per se, mind you; and I think a general "strict mode" in ruby would be somehwat similar what we may have had in the past with $SAFE - but at the same time, the suggestion also confuses me. It seems to bundle together different things into a larger "meta" suggestion, and while I also understand that this is sometimes possible, I don't quite fully understand the suggestion. In my opinion it would be better to split the suggestions up, if necessary, and then make separate descriptions of use cases. As it is right now, I think it makes it very very hard to want to go about favourably, mostly because different ideas are bundled together, with different quality or (perceived) need. Some of the suggestions can also confuse people, such as recursive definition of names, e. g. the example of class A; end; class B; class A; def me; A; end; end - and, quite honestly, this is an example where it would be better if people would keep code SIMPLE and not confusing, so I am not sure if anyone should WANT to need to add features that could be confusing. Ruby is not JavaScript; neither is the design similar. But I think the format of the suggestion here is problematic - bundling together too many ideas makes it hard(er) to want to reason in favour of any of them really, even if I understand a few of them to a certain extent.

#3 - 03/28/2019 11:54 AM - localhostdotdev (localhost .dev)

So, the main thing I was thinking about is passing blocks that then get silently ignored. I was thinking a good way to solve this would be to have a strict mode, and then I tried to come up with things that could be in a strict mode too. The blocks not being used issues is already being discussed and worked on (but I wasn't aware of that).

- About $1, $2, etc. I'm not a fan of global variables that any random piece of code can change, e.g. you do a =~ then a gem does a =~ and now your $1, $2, ... are randomly filled.

- I'm aware I can use monkey patching to solve most issues
- I meant using a(begin:, end:, for:) but actually the alternative (args as a hash) would then let the caller use any parameter name, so not in favor of that anymore
- And yes using def user(user); user.to_s; end is quite confusing to the reader but this is the kind of things linters are for so that's fine

(also should have re-read my original message, there is a few typos there)

Anyway, the main thing was about silently ignoring ruby blocks, so I'm okay for this issue to be closed. Thinking about this, most of these issues are solved by linters already.