Ruby - Feature #18930

Officially deprecate class variables

07/20/2022 10:57 AM - Eregon (Benoit Daloze)

Status:	Rejected	
Priority:	Normal	
Assignee:		
Target version:		

Description

Ruby's class variables are very confusing, and it seem many people agree they should be avoided (#18927).

How about we deprecate them officially?

Concretely:

- Mention in the documentation that class variables are deprecated and should be avoided/should not be used.
- Add a parse-time deprecation warning, now that we only see those with Warning[:deprecation] = true it seems reasonable to add.

History

#1 - 07/28/2022 07:21 AM - byroot (Jean Boussier)

Even though I don't remember adding any new class variable in the last half decade, I'm not sure we can deprecate them, as IMO deprecations should be actionable, and some of these may not be.

I think the main use case for class variables that is still legitimate is for class level configuration without inheritance, e.g.

```
class SomeThingNeverInherited
  class << self
    def some_config=(value)
      @@some_config = value
    end
  end

  def do_thing
    if @@some_config
      do_one_thing
    else
      do_another_thing
    end
  end
end
```

If you deprecate class variables, the only way to do this is to publicize that state, so that the instance can access it with self.class.some_config. That's not a huge deal, but still feels wrong.

I know it's a big ask, but I really feel a better solution would be to introduce true class variables with an usable semantic (like Python).

#2 - 07/28/2022 09:57 AM - Eregon (Benoit Daloze)

For such a case it seems fine to use instance variables on the class and expose e.g. SomeThingNeverInherited#some_config?.
If you don't want that to be public API it could be :nodoc: or made private + using send.

```
class SomeThingNeverInherited
  @some_config = :initial_value

  class << self
    def some_config=(value)
      @some_config = value
    end

    def some_config?
      @some_config
    end
  end

  def do_thing
```

```
    if SomeThingNeverInherited.some_config?
      do_one_thing
    else
      do_another_thing
    end
  end
end

SomeThingNeverInherited.new.do_thing
```

So I think it is fully actionable, even though it might require some thinking especially if inheritance is used, but then the actual semantics were very likely not the intended ones, so such thinking will actually fix bugs.

#3 - 07/28/2022 12:07 PM - byroot (Jean Boussier)

> If you don't want that to be public API it could be :nodoc: or made private + using send.

Yes that's what I say in my comment, and I see how some users may find this unsatisfactory.

#4 - 07/28/2022 11:26 PM - shan (Shannon Skipper)

In the Ruby communities on IRC and Slack we frequently have to convince new Rubyists to just treat class variables as deprecated rather than wasting time learning all about them to find they'll never use them or get them through a code review. It's a very common distration.

It came up again today even on Slack, and the response from an regular user following a discussion of all the gotchas was a typical, "@@vars are considered soft deprecated these days." From my vantage, class variables seem to cause more pain by existing than they would from the relatively rare usage in the wild that would need to be changed to instance variables.

#5 - 07/29/2022 05:54 PM - Eregon (Benoit Daloze)

@shan Agreed.

If we are not yet ready to deprecate them, could we at least discourage them in official docs?

[@byroot (Jean Boussier)]:

> Yes that's what I say in my comment, and I see how some users may find this unsatisfactory.

I think it is a very small inconvenience, which seems to me much less important than many people getting confused and getting bugs due to the near-impossible-to-understand semantics of class variables in Ruby.

> I know it's a big ask, but I really feel a better solution would be to introduce true class variables with an usable semantic (like Python).

I would be fine with that, but the last time I tried something like that it was rejected: [#14541].

#6 - 07/29/2022 05:58 PM - byroot (Jean Boussier)

> the last time I tried something like that it was rejected: [#14541].

[#14541] proposed to change their semantic, which is a hard proposition for backward compatibility. What I'm alluding to would be to introduce another syntax for true class variables, e.g @$var or something like that.

That being said I don't want to derail this thread, I think it's a totally orthogonal proposition.

#7 - 07/30/2022 01:45 AM - Dan0042 (Daniel DeLorme)

I am 100% opposed to deprecation. I use class variables regularly. As long as you initialize them in the class definition, there is no issue with shadowing. They're inherited in subclasses. They work perfectly fine.

#8 - 07/30/2022 11:32 AM - Eregon (Benoit Daloze)

@Dan0042 It seems likely you might not know well their semantics then and how much a trap they are.
Using @byroot's example from https://bugs.ruby-lang.org/issues/14541#note-25:

```
class A
  @@foo = 1
```

```
  def self.foo
    @@foo
  end
end

class B < A
  @@foo = 2 # actually writes in A
end

p A.foo # => 2, but should be 1
```

And if one does not write to them then constants are a much better choice.

Could someone point to a gem using class variables where it is not fairly easy to replace them with instance variables?
Or even better, a gem where the semantics shown above are actually desirable?

I believe nobody wants these semantics (it's buggy behavior, if one wants that they might as well use an instance variable on a given class+accessors or a global variable), hence class variables should be discouraged or deprecated as they are just likely to cause bugs and are not necessary (plus, they look like a Perl mess of sigils).

Because fixing their semantics is deemed too incompatible (#14541), then deprecation seems the only way to avoid people running into these buggy class variables.

#9 - 07/30/2022 06:59 PM - austin (Austin Ziegler)

If we're going to deprecate them, we need something that effectively replaces them—and class instance variables aren't it. I was experimenting with something along the lines of:

```
class Base
  def self.dependents
    @dependents ||= {}
  end

  def self.inherited(klass)
     return unless klass.name.match(/V(\d+)\z/)
     name = $1.to_i
     dependents[name] = klass
  end
end

class Rand < Base
end

class Timestamp < Base
end

class V1 < Rand
end

class V2 < Timestamp
end

class V3 < Timestamp
end

class V4 < Timestamp
end
```

This results in Base.dependents producing {}, but Rand.dependents producing {1=>V1} and Timestamp.dependents producing {2=>V2,3=>V3,4=>V4}.

Switching to a class variable produces the result expected.

I've managed to do this a different way, but automatic collection of all dependents in a hierarchy only at the top level requires a bit of a hacky workaround (Base.dependents[name] = klass).

Yes, in this case, I could *probably* switch to a constant (even a private constant), but that also feels wrong.

#10 - 07/30/2022 10:55 PM - jeremyevans0 (Jeremy Evans)

austin (Austin Ziegler) wrote in #note-9:

> If we're going to deprecate them, we need something that effectively replaces them—and class instance variables aren't it.

The particular approach given may not work (I don't see how it tries to replace class variables), but that doesn't mean you cannot replace class variables with class instance variables. There are a couple approaches, depending on the behavior you want.

The first approach is copying the instance variables from the superclass to the subclass during subclassing in inherited:

```
def self.inherited(klass)
  super
  @class_variable = klass.instance_variable_get(:@class_variable)
end
```

This approach results in being able to access the class instance variable directly in the subclass. It provides the fastest access, with the tradeoff of slower subclassing and making future changes to the superclass not affecting subclasses that do not override the variable.

An alternative approach replaces the class variable with a class instance variable and a singleton method:

```
def self.class_variable
  defined?(@class_variable) ? @class_variable : superclass.class_variable
end
```

This results in slower access, but offers faster subclassing and makes it so future changes to the superclass affect subclasses that do not override the variable.

I should be clear that either way does not attempt to implement the weird semantics of class variables, where setting the variable in a subclass sets it for the entire class hierarchy (up to the superclass that defined the variable). I don't think almost anyone actually wants those semantics.

> Yes, in this case, I could *probably* switch to a constant (even a private constant), but that also feels wrong.

For a constant value and not a variable value, I recommend using a constant.

#11 - 07/30/2022 11:41 PM - zverok (Victor Shepelev)

@austin

About your example

I'd say that referencing a class method in another class method in inheritance hierarchy always makes one ask whether you meant the method/data of the base class or of the current one.

Because, like

> but Rand.dependents producing {1=>V1} and Timestamp.dependents producing {2=>V2,3=>V3,4=>V4}.

I can easily imagine it as a *desired* outcome (e.g., "every non-Vx class is root for its own list of versions, and stores them independently").

So, without using class variables, I'd rewrote that with Base.dependents to make the intention unambiguous: "yes, please be aware, whenever in class hierarchy this will be reused, we are putting it into the **Base's** list of dependents."

You mention it as a "hacky workaround," but I honestly see it as a most explicit (and not that verbose) way of stating the intention; I use it instead of class vars even today.

(It wasn't that way always for me: I adored class vars for this kind of trick... But at some point, I came to an understanding that I needed to challenge "well, I learned the class vars quirk long ago, it is natural to me," and now see them as a way of too cryptic coding of the same, not that frequently needed meaning, "yes, that's data shared throughout the hierarchy")

#12 - 07/31/2022 12:24 AM - austin (Austin Ziegler)

jeremyevans0 (Jeremy Evans) wrote in #note-10:

> austin (Austin Ziegler) wrote in #note-9:
>
> > If we're going to deprecate them, we need something that effectively replaces them—and class instance variables aren't it.
> > The particular approach given may not work (I don't see how it tries to replace class variables), but that doesn't mean you cannot replace class variables with class instance variables. There are a couple approaches, depending on the behavior you want.

It doesn't—as such. This was something I coded up and was *surprised* that it didn't work the way that I expected, and exploring it resulted in the understanding that I have. The *intent* that I had was very much one where I got the behaviour from a class variable instead of a class instance variable. Both suggestions you made would absolutely work, although if this were something that I exploring further, I'd be doing much more work on it (probably building out a bit of a DSL; for what I'm writing, it's overkill).

zverok (Victor Shepelev) wrote in #note-11:

> but Rand.dependents producing {1=>V1} and Timestamp.dependents producing {2=>V2,3=>V3,4=>V4}.

I can easily imagine it as a *desired* outcome (e.g., "every non-Vx class is root for its own list of versions, and stores them independently").

Possible, but IMO unlikely.

Note that I'm in favour of officially deprecating class variables, but this is *one* scenario where the use of a class variable instead of a class instance variable provides a desirable effect with minimal code compared to replicating the behaviour with class instance variables. I think that all three of the workarounds (Base.dependents or either of Jeremy's) are *less clear* in their intent and that the default behaviour is *unexpected*. Not wrong, but surprising.

One of the reasons I still like writing Ruby is there's very little *ceremony* / boilerplate. For this advanced use, it would be nice to have that remain the case.

#13 - 07/31/2022 11:42 AM - zverok (Victor Shepelev)

austin (Austin Ziegler) wrote in #note-12:

> > but Rand.dependents producing {1=>V1} and Timestamp.dependents producing {2=>V2,3=>V3,4=>V4}.
>
> I can easily imagine it as a *desired* outcome (e.g., "every non-Vx class is root for its own list of versions, and stores them independently").
>
> Possible, but IMO unlikely.

Well, it depends on the task. I have met several times with architecture like this:

```
AbstractBaseWidget - DashboardBaseWidget - {DashboardWidget1, DashboardWidget2, DashboardWidget3}
                   \
                     BillingBaseWidget - {BillingWidget1, BillingWidget2}
```

(with AbstractBaseWidget being "library base" without any necessity to know the system, but its immediate children are "roots" of sub-hierarchies and store their children.)

TBH, even by the first glance at your example, my first guess was "Vx's are stored in their own sub-hierarchies." So, "topmost class rules everything" is not a given—and maybe a good thing to pronounce explicitly.

Which leads us to...

> I think that [...] the workarounds [with] Base.dependents are *less clear* in their intent and that the default behaviour is *unexpected*. Not wrong, but surprising.

Would it be possible to expand on that, why it looks less clear for you? The question is not idle for me, I reflect a lot about Ruby's "intuitions", its documentation and "naturality" of the features; and I believe some insights might come out of trying to explain "how I read this" (instead of, how it frequently happens, end with "I see it clearer, period.")

From my side, choosing of three alternatives (tricks with copying variables on the fly aside):

1. Base.descendants[name] = klass
2. descendants[name] = klass (with descendants implemented however... say, with class variables!)
3. @@descendants[name] = klass (explicit use of class variables)

...for me, the reasoning on why prefer (1) to (2) goes like this:

- (2) reads (as it in all other places) as self.descendants[name] = klass
- with self being always *the current object*
- and what looks like "current object's attribute getter" probably encapsulates current object's data
- so in any of the intermediate descendants, the reader's *expectation* would be "it has its own descendants copy"

So, it leaves us with Base.descendants vs @@descendants... With the latter being a "special sigil for helping in this exact case, and no else," and the former somehow perceived as "ceremony" and "boilerplate"?..

#14 - 07/31/2022 12:03 PM - Dan0042 (Daniel DeLorme)

Eregon (Benoit Daloze) wrote in #note-8:

```
class A
  @@foo = 1

  def self.foo
```

```
    @@foo
  end
end

class B < A
  @@foo = 2 # actually writes in A
end

p A.foo # => 2, but should be 1
```

This is exactly the behavior that I want and expect from class variables. If I wanted one 'foo' per subclass I would use class instance variables.

It's not like I use class variables all *that* often, but I do use them more than global variables. So does that mean we should deprecate global variables? Class variables have their uses, and deprecating them is just removing a tool from the programmer's toolbox.

#15 - 07/31/2022 02:23 PM - Eregon (Benoit Daloze)

Regarding @austin's example, there is Class#subclasses now which might be an easier way to do that.
Using class variables for that only works based on a fairly subtle constraint that Base needs to be inherited before any Base subclass can be inherited. Otherwise it would be broken (two subclasses could have separate state).
IMHO using a constant there is what makes most sense, it is clear then where the state lives, and it is convenient to access.

@Dan0042 Right, but most developers do not expect that. And those "class variables" are just prefixed global variables.
With this additional catch (can be a lot less obvious if the cvar is set in a method):

```
class A
end

class B < A
  @@foo = 2
end

class A
  @@foo = 1

  def self.foo
    @@foo
  end
end

p B.foo # => 1, but should be 2
```

> deprecating them is just removing a tool from the programmer's toolbox

Yes, in this case I think this is good because it's a tool which is very difficult to use correctly and can easily be replaced by clearer ways like instance variables, constants, etc.

#16 - 07/31/2022 04:15 PM - zverok (Victor Shepelev)

(Playing for both sides here, hehe)

Actually, I believe that the culprit might be just the name. E.g., we have two camps, basically saying:

1. The behavior of [however it is named] is useful, and it matches my expectations of [this thing], here are usages
2. The behavior of what we call "class variable" doesn't match the expectation of a thing called "class variable," and therefore, it is confusing and hard to use right.

(I still remember the confusion I had when I just studied Ruby and understood that classes are objects too and can have their own *instance* variables, and how those are different from *class variables*, etc.)

That being said, as far as I can Google, "static variables," say, in both C++ and Java, are shared with subclasses, but *defining* the static var with the same name in the sublcass "shadows" the parent's one. In Ruby, the distinction between "whether we are defining it" or "whether we are assigning to the existing thing" is murky for variables, so...

While I still believe what I said above (that class vars is not the best tool for any task), I wonder where the concept of "it should not be shared with subclasses, looks surprising/like a bug" comes from? Maybe just because *the name* implies it, somehow?.. Maybe changing the name to "class shared variables", or "hierarchy variables", or something like that would change the mental model.

#17 - 07/31/2022 05:06 PM - austin (Austin Ziegler)

Eregon (Benoit Daloze) wrote in #note-15:

> Regarding @austin's example, there is Class#subclasses now which might be an easier way to do that.

Not really, because that would require some sort of runtime filter of said subclasses. The concept I was putting together is that there are some defined *ways* of doing things that are versioned. V1 uses a random generator; V2, V3, and V4 use a timestamp generator. There's a payload that would include an integer version number and then it would be something like Verifier[version].verify(payload), where Verifier.[] returns the class, but *only* if it's versioned. Using Class#subclasses would mean that Verifier.[] would need to filter based on Class#name matching V#{version}. This would be runtime looping that doesn't work with the intended performance characteristics.

> deprecating them is just removing a tool from the programmer's toolbox

> Yes, in this case I think this is good because it's a tool which is very difficult to use correctly and can easily be replaced by clearer ways like instance variables, constants, etc.

I agree that @@vars are *very* hard to use properly and are best deprecated with a long-term plan of removal…but I'm not as convinced that the alternative ways are *clearer*. I've been using Ruby for 20 years now and *still* was a little surprised when I tried what I tried and ended up with separate hierarchies. I think that, in order to deprecate them, good documentation needs to be written on how and when to use the alternatives *instead* of class variables. (This brings up a different issue, in that "how the language works" documentation is sparse and poorly organized; where is the discussion of instance variables, class instance variables, and class variables to be found in https://docs.ruby-lang.org/en/3.1/index.html?)

#18 - 08/02/2022 07:00 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

I admit class variables semantics are a bit complex and sometimes misunderstood.
But removing them should cause serious compatibility issues.

Class variables are somewhat similar to global variables. We don't recommend using/abusing them, but not going to remove them from the language.

Matz.

#19 - 12/07/2024 02:26 AM - shan (Shannon Skipper)

Can we formally document that class instance variables are soft deprecated or not recommended for use? In the Ruby community we often see new folk struggle greatly with them and have to recommend against their use over and over.

#20 - 12/07/2024 05:34 AM - Dan0042 (Daniel DeLorme)

I'm still against deprecating, either soft or hard, but I believe the errors could be improved.

```
class A
  def self.foo
    p @@foo ||= rand
  end
end

class B < A
  p @@foo = 2
end

B.foo
```

In this code, the @@foo variable is overtaken but the "overtaken" error isn't triggered because even though the foo method is invoked on B, it was defined in A. If this could be improved to raise an error, I believe it might relieve some of the confusion over class variables.