Ruby master - Feature #14887
Array#delete_if does not use #delete

06/30/2018 09:50 AM - sdaubert (Sylvain Daubert)

Status: Closed
Priority: Normal
Assignee: matz (Yukihiro Matsumoto)
Target version: 

Description
When a class inherits from Array, and its #delete methods is modified, #delete_if and #reject! do not call this method. From sources, its an internal C method which is called instead.

Is there a way to modify #delete_if and #reject! to use #delete? This would ease inheriting of Array class.

History
#1 - 07/02/2018 03:53 AM - shyouhei (Shyouhei Urabe)
- Backport deleted (2.3: UNKNOWN, 2.4: UNKNOWN, 2.5: UNKNOWN)
- ruby -v deleted (2.5.1p57)
- Tracker changed from Bug to Feature

Hmm, let me make it a feature request so that we can consult matz.

#2 - 07/02/2018 07:45 AM - Hanmac (Hans Mackowiak)
I don't think there is a way:

```
a = [1,2,3,2,5]
a.delete_if { |x| x == 2 }
a #=> [1,3,5]
```

this might call delete, but what about this?

```
a = [1,2,3,2,5]
a.delete_if.with_index { |x,i| x == 2 && i == 1 }
a #=> [1,3,2,5]
```

as you can see, you can't make delete_if call delete without breaking other code

#3 - 07/02/2018 10:17 AM - Eregon (Benoit Daloze)
Do you mean delete_if and reject! should call delete_at?

Calling #delete doesn't make sense, as Hanmac (Hans Mackowiak) showed above.

In general, I would not recommend to inherit from Array, there are many pitfalls. It is much simpler and better supported to use composition.

#4 - 07/02/2018 06:03 PM - sdaubert (Sylvain Daubert)
I didn't think to enumerators...

Yes, #delete_at is a better choice than #delete, didn't think to it neither...

But #delete and #delete_if methods seem to not call #delete_at neither. Please let me reword the proposal: may #delete and #delete_if (and #reject!) call #delete in the background to ease inheriting from Array class?

#5 - 07/02/2018 07:41 PM - marcandre (Marc-Andre Lafortune)
- Assignee set to matz (Yukihiro Matsumoto)

You probably meant "... call #delete_at in the background".

I think there could be a performance issue for big arrays if calling delete_at. For example delete_if(true) would be moving O(n^2) elements (for no reason!) while delete_if is strictly O(n).
Overall, I'm very sceptical about the request. Array could have been built using some kind of RandomAccess module with a minimal set of primitives to read, write, add and remove elements, but that's simply not the case. delete_if doesn't even call Array#each for example, and Array#each doesn't call Array[]. I think it's unrealistic at this point to go in that direction.

#6 - 07/05/2018 07:05 AM - shevegen (Robert A. Heiler)

This is actually a (to me) quite interesting discussion. I want to comment on a very few things but of course I'll keep in mind that there is a specific suggestion in the issue which should not be forgotten.

Sylvain suggested, essentially, the ability to be able to:

a) subclass from Array
b) let #delete use #delete_if / #reject!

Marc-Andre made one comment that there may be a performance issue.

To this I would like to add that there is a philosophical consideration to it, in that we can distinguish two cases based on the following:

- The ability for ruby hackers to use something or not in the first place

There may be a performance issue (let's assume that there is one); but we do not currently have the option to do what Sylvain suggested. In other words, the above could mean "because there is a performance issue, you are unable to do as proposed" - which is different to "there is a performance issue, but you can use this or that specific workaround at your own discretion". What I am trying to say here is that I think it is a different case if we say that because of performance issue, you are unable to do this. Which I think is not so close to ruby's philosophy. Ruby allows lots of flexibility and I think this should be taken into consideration as well; not just performance issues alone.

I'll soon come to look for alternatives but to a comment made by Benoit:

In general, I would not recommend to inherit from Array, there are many pitfalls. It is much simpler and better supported to use composition.

I agree with this personally; I myself avoid subclassing from the core classes usually (in particular Array and Hash; I think I have subclassed from class String before, without problems though).

Nonetheless, I think subclassing should work "in general", meaning even for core classes. This is more from a philosophical consideration - I think it would be better if we can all safely subclass from every class. But you also mentioned composition and I think this is actually interesting.

If we subclass from an existing class, ruby sort of "copies" that class and we can then modify the subclass - add or change methods, instance variable and so on.

You can do just about the same through modules too, but at a more fine tuned control level. For example, add a few methods into a module and then include that module or otherwise call it from the class (or subclass). So subclassing and inheritance/composition is very similar; just that presently ruby allows us to do only 100% inheritance (via subclassing) whereas modules allow for a different percentage here.

What about "partial subclassing"? For example, where the ruby hacker can say which parts he wants to subclass - aka below 100%? Like to specify which methods should be subclassed from, rather than the whole class altogether.

Some ruby gems do this in some ways already, although not in the sense of a "partial subclass", but in regards to modules that are almost on a per-method basis ... I think the facets gem and active* gems, at the least a few of the latter, allow you to include specific methods from a module or something like that.

Ultimately this is, in my opinion, mostly something matz has to decide
whether this is "good ruby practice" or not - but I personally can understand the use case why people do this. They have more control over their way to build up composition. If you follow this line of thought - and let's assume that matz thinks it fits to the ruby philosophy - we could also look at Sylvain's suggestion here to mean something like this:

"Please consider giving us ruby hackers a possibility to (a) subclass from Array freely and (b) change internal behaviour of that subclass" (e. g. to tell other methods to use #delete, in that subclass of Array).

This is not for me to say that I am pro or con to the suggestion itself at all, mind you. I am neutral. I just think that the suggestion and possible use case is quite interesting. It's more the question how much control ruby hackers should/may have when it comes to inheritance and subclassing. To me, this also fits towards the suggestion in the proposal, although his suggestion is about class Array behaviour; whereas I partially describe behaviour on the subclassed array. (If code breaks then I don't think the proposal has a real chance to go through, but he specifically also mentioned his use case of subclassing from Array; so in his subclass, why should he not be able to decide that his subclass should behave differently from class Array if he wants to? I don't know of an easy way to do this, but in theory it could be possible. Also note that I have no idea how common this pattern may be, but other people can chime in and say whether they have had a use case or a need for this in the past.)

Calling #delete doesn't make sense, as [Hanmac](https://example.com) showed above.

Hanmac gave one use case involving .with_index but the threadstarter did not mention having a use case for .with_index - and while one can assume that the threadstarter indeed meant to change class Array behaviour, which may not be possible due to backwards issues, he also mentioned subclassing, and in this case I think the described use case is understandable. But again, I am completely neutral here - I only wanted to point out that a few arguments should not be viewed too narrowly in regards to the use case at hand. Last but not least, though, I guess if we want to discuss include/subclassing and fine-tuned control on a "less than 100%" subclass, it may be best to do so in a new issue; I only wanted to point this out here though.

#7 - 07/05/2018 07:28 AM - Hanmac (Hans Mackowiak)

shevegen (Robert A. Heiler) said: the problem is that:

- delete_if can't call delete for the given reason of duplicates. the example with with_index was only an example to show where it differ.
- delete_if can't call delete_at because it would cause massive array resize inbetween. i also don't know if the index would be handled right there

the only thing i could be okay with is that if delete_if would call reject! (and looks for overwrite there)

#8 - 07/05/2018 06:32 PM - sdaubert (Sylvain Daubert)

marcandre (Marc-Andre Lafortune) wrote:

You probably meant "... call #delete_at in the background".

Yes, Marc-André, that's what i want to mean...

Overall, I'm very sceptical about the request. Array could have been built using some kind of RandomAccess module with a minimal set of primitives to read, write, add and remove elements, but that's simply not the case. delete_if doesn't even call Array#each for example, and Array#each doesn't call Array[]. I think it's unrealistic at this point to go in that direction.

I don't known ruby background. If it seems impossible, forget about that.

#9 - 08/09/2018 06:13 AM - matz (Yukihiro Matsumoto)

It is intentional optimization for time & space. Besides that a subclass of Array (or String) is generally not a good idea.

Matz.

#10 - 08/09/2018 06:13 AM - matz (Yukihiro Matsumoto)