obj::method is introduced at r66667 by #12125 and #13581. It encourages the functional programming style in Ruby.

But this shorthand syntax is just for methods of self without arguments. It causes another feature requests like #16273 (and lambda compositions like #15428).

Such features will introduce a new view of Ruby but no one illustrates the whole picture yet. I worried about such patch work may cause a conflict with future expansion of functional programing style or a just a garbage feature.

`:` syntax is introduced in 2.7.0 preview1, not released in production yet. How about reverting at this time and re-introduce with a big picture.

Related issues:
Related to Ruby master - Feature #12125: Proposal: Shorthand operator for Obj... Open
Related to Ruby master - Feature #13581: Syntax sugar for method reference Closed

I worried about such patch work may cause a conflict with future expansion of functional programing style or a just a garbage feature.

Related to Ruby master - Feature #16275: Proposal: Shorthand operator for Obj... Open
Related to Ruby master - Feature #13581: Syntax sugar for method reference Closed

Associated revisions
Revision fb6a489a - 11/12/2019 08:24 AM - nobu (Nobuyoshi Nakada)
Revert "Method reference operator"
This reverts commit 67c574736912003c377218153f9d3b9c96a17b.
[Feature #16275]

History
#1 - 10/23/2019 04:50 PM - nobu (Nobuyoshi Nakada)
- Description updated
- Subject changed from Revert .: syntax to Revert `.:` syntax

I think that the idea behind .: is ok but syntax-wise I am not sure if it is good to have it in that way. Naruse actually made another point, though, even if the other statement was not entirely correct, in the sense that he referred to proposals that may be in conflict/competition with the syntax - see zverok's suggestion for e. g. &.: or something like that; and I am not quite liking the ideas that spawn off from other prior ideas here, simply syntax-wise alone. Even though that may be subjective due to a personal evaluation (as style preferences may differ between individuals).

It also taps a bit into what Martin Duerst said some time ago about presenting or
using some unified approach, idea and framework for functional programming and ideas originating from here. With many disparate, individual suggestions that may partially "block" one another, it may indeed be difficult to use something more unified and streamlined. So from that point of view, I think naruse's comment may be a good comment too, irrespective of the initial statement which may not be the primary case/cause. (Although I also understand that his comment was not necessarily about preferences in regards to liking or disliking :: as such)

Syntax-wise in general oldschool ruby put up a high barrier as a "standard". :) Syntax may not be the only or a primary area of concern when there are competing suggestions for new features and new behaviour, but I think that syntax is also quite important in general.

#4 - 10/23/2019 06:04 PM - jeremyevans0 (Jeremy Evans)
I am in favor of removing .:. It was introduced before numbered parameters, but I think using numbered parameters is better than using :: for solving the same problems :: is designed to solve:

```ruby
%w'file1  file2'.map(&File.read)
%w'file1  file2'.map(File.read(_1))

%w'file1  file2'.map(&File.read << 'dir/'.++)
%w'file1  file2'.map(File.read('dir/'+_1))
```

nobu (Nobuyoshi Nakada) wrote:

It is not just for "the functional programming style". Originally it was to extract the exact method even if obj.method was overridden. You can use ::Kernel.instance_method(:method).bind_call(obj, :method_name) if you really need this if obj.method is overridden. That's definitely more verbose, but I don't think this need is common enough that we need a shortcut for it.

#5 - 10/23/2019 06:29 PM - Dan0042 (Daniel DeLorme)
I'm also in favor of removing :: until a better "whole picture" is formed. Besides, that syntax encourages the idea that symbols are closely related to functional objects. And it looks like cuneiform.

For example, if we consider that -> is a functional syntax for lambdas, we could adopt the '>' character as a common element of functional syntax in general. Below is a very hypothetical example just to show that a whole-picture view could possibly lead to different results:

```
potential functional operators
  !>
  &>
  {>
  {*}>
  +>
  ,>
  => lambda
  >/ >:word :word.to_proc
  ;>
  <!>
  @>
  ^>
  ^>
  |> pipeline
  >>word obj.method(:word)
  +=word obj.instance_method(:word)
  conflic	/ already used
  ">
  
  '%'>
  string
  %>
  comment
  $>
  stdout
  %>
  string
  )>
  greater-than
  .>
  greater-than
  => hash literal
  >>
  right shift
  ?>
  erb
  ]>
  greater-than
  ```
I think that it was just one of the motivations of that. Anyway, people will mainly (ab)use it like .map(&JSON.parse). I don't like this style. It is ad-hoc, incomplete, not composable, just hacky, and even complicated.

I'm currently thinking that the following design is preferable.

- A plain old style should be the base: .map { |x| x.to_s(16) }.map { |x| JSON.parse(x) }
- A numbered parameter makes simple cases simpler: .map { _1.to_s(16) }.map { JSON.parse(_1) }
- #16120 is worth reconsidering which makes typical case rather simpler: .map { .to_s(16) } (and this style is more flexible than the current .map(&:to_s))
- .map(&:to_s) should be deprecated

I agree with all the above. And if you can get Matz to reconsider #16120 nothing could make me happier :-) But since I prefer map(&:to_s) to map[_1.to_s], deprecating the former without #16120 would be rather painful.

I was looking forward to seeing this because it is very useful when checking the source location of a method during debugging (especially with in irb). It is sad that .: syntax is reverted. However, I agree to use Numbered parameters rather than .: for block arguments.

Disclaimer: This matters are somewhat personal for me, I was between the people pushing for "method reference operator", and one of the tickets naruse (Yui NARUSE) references from the phrase "It causes another feature requests ..." is written by me (and another one is pretty close to one of my recent tickets: #16264). Despite this, I am trying to the best of my judging to focus on Ruby's consistency, evolution, and prosperity, not "I want them to merge my tickets!" stance.
1. The idea that "the new feature causes people to think of new feature requests" is a bad thing, seems a bit unusual. It seemed to me that this is how the good design happens: we investigate where current language’s syntax and semantic leads us, what new ideas emerge, and what upcoming ideas those new ideas produce. So, "this feature leads to thinking about new ways of language usage" seemed a sign of "something right".

2. Ruby’s design process is known to be opaque (unevenly split between public discussions, closed meetings, and some particular person-in-charge decisions), but till this moment, I had an unfortunate belief at least part of it happens in this tracker. Probably, this is a false belief.

3. For example, following this belief, I am creating probably an awful lot of proposals, expecting to not only receive "accepted"/"rejected", but clarify understanding of what is in line with language's goals and intuitions. For more particular example, here: #16264 I am not only proposing some new syntax/class, which could be good or bad, but also trying to state some general design ideas, which seem to me generally applicable, and have a (slight already) hope to discuss them with core team—because I don't know any other place to do so.

4. That's why the claim of "not having the whole picture" was a big surprise: previously, I hoped that discussions in this tracker is the way of formulating the "whole picture", and, even more bold, that what's forming with new method reference operator is, in fact, a part of clearly visible picture of "gradually yet consistently moving towards functional Ruby". Now, it is a mystery to me whose right and responsibility is to own "the whole picture": is it a closed group of dev. meeting attendants? Matz's very own? Some dedicated "design manager of upcoming functional features"?

5. Another big surprise here is how unanimous the aversion for the new operator seems to be. On the previous iteration (when it was about to be introduced) it seemed to me that everybody agrees that it is an obvious and useful thing (method references as a first-class construct), and the only stumbling block is which characters to use for it, with :: being a reasonable compromise. But now a lot of people are suddenly against it (and, the funniest thing is, all for different reasons: "ad-hoc", "don't like how it looks", "numbered parameters is better"), with "method references" believed to, I don't know, esoteric and useless concept?

6. About numbered parameters: I know it sounds bold, but do believe it is the most useless new feature syntax-wise for several latest releases. My justification is exactly symmetrical with naruse /Yui NARUSE’s: this is an "orphan" feature, which is "just nice shortcut", but it doesn't lead to any new ideas, any new possibilities, and increases language's syntax surface without increasing language's power. I believe the feature increases the language's power when it encourages new thinking. This way, method reference encourages clearer APIs suitable for passing arguments that belong to API, e.g. redesigning map { |x| MyService.method(x, lot, of, options) } into map(&configured_service.:method), thinking when and how we pass lot, of, options; while numbered parameters allow preserving exactly the same thinking as before, just spare some keystrokes with map { MyService.method(1, lot, of, options) }

7. I understand there are generally two camps when we talk about language design: one that feels current design is more or less "final", just implementation can be improved (but maybe some small "shortcuts" are allowed); and one that feels that language is a living thing, and constantly picking existing concepts to understand what new ideas they can lead to is necessary. Obviously, numbered parameters belong to "first camp thinking", and method references to the second. It is sad that the core team currently tend to stay in the first camp, yet completely understandable.

8. With this being said, I am grateful to this situation: at least, it allowed me to seriously reconsider my involvement with the language and its future.

#13 - 10/25/2019 06:37 PM - Dan0042 (Daniel DeLorme)

zverok (Victor Shepelev) I really feel your pain, I had the same reaction when some of my ideas were ignored or rejected; that never feels good, especially so if you strongly believe to be right. Just take a bit of distance, don't take it personally, remember that it's human nature to resist change and that "if your ideas are any good, you'll have to ram them down people's throats": This is going to be true anywhere.

This is my personal opinion but it seems (perhaps?) shared by many here: The idea behind :: is fundamentally good but the syntax is going in the wrong direction. The map(&:to_s) syntax was a hack to begin with. Matz himself doesn't even like it. obj.method and .method go in the same direction of associating symbols with functional/callable objects and that doesn't feel right to me, even though the basic idea/purpose seems good. What originally seemed a reasonable compromise is looking less and less so.

You are entirely correct that "discussions in this tracker is the way of formulating the whole picture", and precisely in this case the whole picture that is starting to emerge feels... off. It's not a problem that a "new feature causes people to think of new feature requests", but the new features feel imperfect and those aditional feature requests feel like compounding the imperfectness, and that's more of a problem. And the worry is that once :: goes out in 2.7 it will be pretty much impossible to go back. Rather than adding this feature and see where things go from here (a.k.a. "ad-hoc"), I feel it's better to backtrack for now and try a slightly different path. And maybe come back to this syntax later, who knows.

TL;DR I believe the opposition here is mostly to the specific syntax and its ensuing repercussions, not the idea behind it.

#14 - 10/25/2019 11:28 PM - timriley (Tim Riley)

I'm opposed to this reversion. I was looking forward to using :: in 2.7. I felt I should say this as a representative of perhaps a "silent majority" of Ruby users.

I think that some of the additional ideas this feature was spawning (i.e. some of the other tickets referenced above) were potentially very helpful evolutions of the language.

Regarding the point that a feature like :: should wait until it comes out as part of a "whole picture," how is that meant to happen when contributors like zverok do not have access to various key aspects of the development and decision-making process, as he has questioned in his comment? Would the Ruby core team really be accepting of a whole family of multiple related features coming into a release all at once? That feels unlikely to me, especially given zverok and the other key drivers of these ideas do not have access to some of the avenues for development discussion outside of this bug tracker. So their best bet is to introduce the smaller features one by one - the initial parts of the "whole family" a which is what they've done with ::.

So where does this leave this area for improvement to Ruby? If :: is rejected now, it feels like it closes off this evolutionary branch of language features for good. Which, imo, is anything but good.
matz (Yukihiro Matsumoto) wrote:

I am for adding syntax sugar for method reference. But I don't like proposed syntax (e.g. ->).
Any other idea?

Out of ruby-core:85038 candidates, :: looks best to me (followed by ::::).
Let me consider it for a while.

So clearly the syntax sugar for method reference is something that we will have. Eventually if not now.
After the "let me consider", nobu implemented it with :: but was there ever a final word from Matz?

#16 - 10/26/2019 07:02 AM - jeremyevans0 (Jeremy Evans)

timriley (Tim Riley) wrote:

I'm opposed to this reversion. I was looking forward to using :: in 2.7. I felt I should say this as a representative of perhaps a "silent majority" of Ruby users.

If you think your opinion represents the "silent majority" of Ruby users, I think you are operating in an echo chamber. I realize the this feature may fit nicely into the dry-rb projects, but you should realize that users of dry-rb are probably not representative of the average Ruby user. I don't want to pick on you personally, but please do not pretend to speak for the majority of Ruby users.

I think that some of the additional ideas this feature was spawning (i.e. some of the other tickets referenced above) were potentially very helpful evolutions of the language.

Most Ruby users I have spoken too (maybe even a "silent majority" of Ruby users :) ) feel that the introductions of similar features such as Proc#<< and Proc#>> actually make the resulting code harder to read. What is considered a helpful evolution by some users can be cryptic and harder to understand for others.

All additional syntax has costs in terms of both internal/implementation and external/cognitive complexity. All proposals to add syntax need to consider the costs of adding the syntax, and the expected benefits of the syntax.

Regarding the tickets referenced above:

#16264: This complicates things further, and I am guessing that most Ruby users would find this approach to structuring more difficult to understand than traditional Ruby. Do you think most Ruby users would prefer:

```ruby
paragraph_hashes.map(&:merge.with(author: current_author))
filenames.map(&File::read.with(mode: 'rb'))
```

to:

```ruby
paragraph_hashes.map{|h| h.merge(author: current_author)}
filenames.map{|f| File.read(f, mode: 'rb')}
```

I guess if you really hate naming block variables, and you really hate numbered parameters, maybe the former is preferable. :)

#14145: While self..method_name instead of method(:method_name) is a bit shorter, that doesn't seem like a major reason to add special syntax, considering inspect is mainly used for debugging. Also, this was already committed, so the loss of :: will not affect it.

Regarding the point that a feature like :: should wait until it comes out as part of a "whole picture," how is that meant to happen when contributors like zverok do not have access to various key aspects of the development and decision-making process, as he has questioned in his comment? Would the Ruby core team really be accepting of a whole family of multiple related features coming into a release all at once? That feels unlikely to me, especially given zverok and the other key drivers of these ideas do not have access to some of the avenues for development discussion outside of this bug tracker. So their best bet is to introduce the smaller features one by one - the initial parts of the "whole family" a which is what they've done with ::.

I don't think we should reject this because we don't have the "whole picture". I think we should reject this because the reason it was added is better served by another feature that is more flexible and easier for the average Ruby user to understand. Numbered parameters handle both cases where you would use :: and cases where you would use the feature proposed in #16264.

So where does this leave this area for improvement to Ruby? If :: is rejected now, it feels like it closes off this evolutionary branch of language features for good. Which, imo, is anything but good.

I don't think this closes off anything. This is just a specific feature that is independent of other features. If there were other features in this "evolutionary branch" that improved Ruby, I think they would certainly be considered.

For example, with #16264, if :: was removed, I assume that paragraph_hashes.map(&:merge.with(author: current_author)) could still work, and
filenames.map(&File.read.with(mode: 'rb')) could be changed to filenames.map(&File.method(:read).with(mode: 'rb'))

#17  - 10/26/2019 07:17 AM - nobu (Nobuyoshi Nakada)

zverok (Victor Shepelev) wrote:

OK, with several core team members agreeing on this (and even PR prepared), I believe this is already a lost cause, but I'd still like to put some perspective here, if just for reference and future discussion.

Just to be clear, that I had written the patch doesn't mean that I'm favor in the removal.

#18  - 10/26/2019 12:09 PM - shevegen (Robert A. Heiler)

with several core team members agreeing on this

At the end of the day you only need to find pro/con arguments for matz. :)

Once a syntax is picked, though, it may be mutually exclusive to other syntax and follow-up ideas building up on that syntax. I think that was one part of the reasoning for the suggestion here.

What might perhaps help would be a separate issue/discussion for a larger more comprehensive "functional" approach/overview in ruby, as Martin suggested some time ago. Something like listing what ruby users may want to have, and then looking at syntax that could fit to that approach. And to perhaps aim for it past ruby 3.0, mostly because I think the time may become a bit short otherwise, since ruby 3.0 will be released in a bit over a year from now on - matz and the ruby team probably want to make 3.0 as polished as possible. :)

#19  - 10/29/2019 11:05 AM - timriley (Tim Riley)

Jeremy – I'm sorry for the overreach in that clumsy wording. Let me rephrase: what I saw here was a fair number of comments supporting the removal of an already-implemented feature of 2.7 that I felt had good potential and that I was personally looking forward to using. I wanted to make it clear that there was at least some positive anticipation for the feature from within the community of Ruby users; if I had said nothing, maybe that sentiment would've gone unnoticed.

As for dry-rb, I did not leave my original comment as any kind of representation from that project (which is why I did not mention it). If there's any further misunderstanding I'd be more than happy to clarify things with you personally, outside of this ticket. Cheers :)

#20  - 10/30/2019 11:11 PM - alanwu (Alan Wu)

`:` has special power in that one cannot change its semantics by redefining a method. It other words, it's a fundamental operation in the language like class <<- object and def object.method_name; end. I think a fundamental operation should only be introduced if there is a good supporting ecosystem for it in the language.

I see reverting as a move to take a step back and do more design thinking about functional programming and how it fits in with the rest of the language. With a good ecosystem behind it, `:` could be much more powerful than just a shortcut.

On the other hand, keeping `:` could be a way to crowd-source the design process. zverok (Victor Shepelev) already has some tickets that depend on `:` for expanding the FP part of Ruby. If `:` hits a mainline release, more people could come forward with ideas to gradually make `:` less orphan and more powerful. In the worst case scenario, no new paradigm is explored and we are left with a syntactic sugar for an unpopular operation, I don't know if that is an okay risk.

When it comes to primitive operations, less is more, so I think we should revert for now.

#21  - 11/05/2019 11:38 AM - maciej.mensfeld (Maciej Mensfeld)

- File Zrzut-ekranu-z-2019-11-02-17-04-12.png added

#22  - 11/05/2019 11:38 AM - maciej.mensfeld (Maciej Mensfeld)

- File deleted (Zrzut-ekranu-z-2019-11-02-17-04-12.png)
I'm also in favor of what Tim stated but at the same time, I want to point out, that the way it is now is not optimal from the performance point of view.

Ruby gives you a new instance of a #Method class. Even when you’re fetching the method of the same instance of an object. That’s not all. If used with the & operator, each of the fetched method references is later on converted into a Proc object using the #to_proc method. That makes the code almost 10 times slower than the "standard" one and heavily stresses out the GC. I’ve pointed this here already: [https://bugs.ruby-lang.org/issues/16103](https://bugs.ruby-lang.org/issues/16103) and here are my recent benchmarks: [https://mensfeld.pl/2019/11/the-hidden-cost-of-the-ruby-2-7-dot-colon-method-reference-usage/](https://mensfeld.pl/2019/11/the-hidden-cost-of-the-ruby-2-7-dot-colon-method-reference-usage/)

I would not expect the majority of Ruby programmers to know subtle details like this one (especially as those may or may not be certain Ruby implementation detail).

sidenote: sorry for the several comments: wanted to upload the benchmark file with a proper name.

---

#23 - 11/05/2019 11:39 AM - maciej.mensfeld (Maciej Mensfeld)
- File dot-colon-vs-std-benchmark.png added

#24 - 11/12/2019 08:26 AM - nobu (Nobuyoshi Nakada)
- Status changed from Open to Closed


---

Revert "Method reference operator"

This reverts commit 67c574736912003c377218153f9d3b9c0c96a17b.  
[Feature #16275](https://bugs.ruby-lang.org/issues/16275)

#25 - 11/13/2019 05:23 AM - zverok (Victor Shepelev)


Revert "Method reference operator"

---

"Ruby is no longer my project. It is the Ruby community's project" —Matz (as quoted by Bozhidar Batsov)

For my 15 years of writing in Ruby, teaching Ruby, documenting Ruby, contributing to Ruby, it is the sole worst thing that happened. Just to summarize:

- The feature which is:
  - short, concise and elegant
  - doesn't conflict with any other feature
  - consistent with the rest of the syntax, and with the recent years of evolution towards functional;
  - discussed by dozens of people for multiple years (with a general agreement "it is necessary, just let's invent the best syntax for it")
  - was already merged and lived through preview-1 and preview-2
- ...was reverted, because:
- some of the core team doesn't like "people invent new things basing on this feature" and it is now a bad thing
- some of the core team just don't like it at all or don't see any value or don't like the particular syntax
- ...and it was done:
  - in a "stealth" deal-with-it manner (be aware that automatic comments "Closed via changeset" aren't sent to ruby-core mailing list)
  - without an announcement of any consensus of the core team, even post-factum one (I can assume it was decided on yesterday's "online meeting to confirm spec of 2.7" #16333 - but apparently "general population" doesn't deserve to know what was agreed there)
  - without explanations of the reason of reverting and future of the idea (nobody needs atomic native method references? they would be in Ruby 3.0, just the syntax/consequences would be rethought? Ruby turns off from "functional" ideas altogether with awesome one-size-fits-all numbered block parameters?)
- without comment from Matz, which such a "minor" thing probably doesn't deserve anyways, right?..

This could probably the last drop to lose interest in Ruby's evolution and governance. So be it.

Thanks for everything, anyways.

---

#26 - 11/13/2019 11:15 AM - matz (Yukihiro Matsumoto)

*zverok (Victor Shepelev)* Sorry if we hurt your feeling. In recent years, we have added/discussed a lot of features inspired by functional programming. But those features are not designed with a grand design. Those are rather ad hoc additions. We felt the future, more functional Ruby should be designed with the big picture of functional Ruby programming, which we lack currently.

Besides that, I don't like the specific operator (,:) which looks like braille.
So you don't have to consider this revert as a rejection of this whole idea, but an invitation to the next try.

Matz.

#27 - 11/13/2019 11:18 AM - matz (Yukihiro Matsumoto)
zverok (Victor Shepelev) I should have explained the reason right after the developer meeting. Actually I was asked to do so from the other members of the meeting. Sorry for my laziness.

Matz.

#28 - 11/20/2019 10:21 AM - znz (Kazuhiro NISHIYAMA)
- Related to Feature #12125: Proposal: Shorthand operator for Object#method added

#29 - 11/20/2019 10:21 AM - znz (Kazuhiro NISHIYAMA)
- Related to Feature #13581: Syntax sugar for method reference added

Files

<table>
<thead>
<tr>
<th>File Name</th>
<th>Size</th>
<th>Date</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot-colon-vs-std-benchmark.png</td>
<td>8.69 KB</td>
<td>11/05/2019</td>
<td>maciej.mensfeld (Maciej Mensfeld)</td>
</tr>
</tbody>
</table>