Ruby master - Feature #14277

Improve strings vs symbols ambiguity

01/03/2018 04:06 AM - dsferreira (Daniel Ferreira)

Status:	Rejected	
Priority:	Normal	
Assignee:		
Target version:		

Description

This is the ambiguity:

```
alias_method :foo, :bar
alias_method "foo", "bar"
```

Ruby developers are using strings and symbols interchangeably as if they were the same thing.
This is happening in ruby core, in ruby gems and in ruby applications.

This discussion as started 5 years ago in two separate feature requests (both rejected):

- 5964
- 7792

I believe ruby will be much better once the ambiguity between strings and symbols is resolved for good
and ruby 3.0 is a very good opportunity to do so.

From further discussions I got a light of hope that a solution may be accepted if certain conditions are met.
Specifically, a clear transition path that could lead the community towards the break of backwards compatibility.

In the issue Make symbols and strings the same thing

ko1 (Koichi Sasada) wrote:

> Please consider transition path for users who are using symbol and string difference like:
>
> key = ...
> ...
> when key
> case String
> ...
> case Symbol
> ...
> end
> How to find out such programs?

he also wrote:

> If you (or someone) find out any good transition path, we think we can consider again.

Can we discuss here what are the rules that would allow the transition path solution to be accepted?

Also what solutions for the problem are we envisioning?

1. Use current symbols syntax as yet another strings syntax and stop using Symbols?
2. Use current symbols syntax as yet another strings syntax and start use Symbols with a new syntax?
3. Use current symbols syntax as yet another strings syntax and use Symbols purely as a class?

From the challenge presented by Koichi I understand that the transition path to be accepted must allow the current code to raise a warning for the situation where the Symbol is not anymore a Symbol but a String.

Is this assumption correct?

If this is the case then all we need is to make String::===(foo) and Symbol::===(foo) to raise warnings every time foo is a string and it was created using former symbol syntax.

This means the foo object needs to contain encapsulated the information of the syntax used to define it.

Any drawbacks?

NOTE: (I'm only considering solutions 2. and 3. for the purpose of this analysis. Meaning Symbol class will still exist.)

History

#1 - 01/03/2018 05:27 AM - duerst (Martin Dürst)

dsferreira (Daniel Ferreira) wrote:

> This is the ambiguity:
>
> ```
> alias_method :foo, :bar
> alias_method "foo", "bar"
> ```
>
> Ruby developers are using strings and symbols interchangeably as if they were the same thing.
> This is happening in ruby core, in ruby gems and in ruby applications.

So are you saying we should disallow

```
alias_method "foo", "bar"
```

I definitely would support that (after some depreciation period).

#2 - 01/03/2018 05:36 AM - jeremyevans0 (Jeremy Evans)

dsferreira (Daniel Ferreira) wrote:

> This is the ambiguity:
>
> ```
> alias_method :foo, :bar
> alias_method "foo", "bar"
> ```

This is not ambiguity. The methods are designed to be used with symbols. The methods also accept strings in order to be a little easier to use in some situations.

Many methods in ruby accept multiple types of arguments and convert one type to another to improve programmer happiness.

To say strings and symbols are ambiguous is to imply that it is not possible to differentiate the two. That is not true in ruby. Symbols have different purposes than Strings, and in all cases it is possible to determine which one is more appropriate, even if you allow both for ease of use.

> Ruby developers are using strings and symbols interchangeably as if they were the same thing.
> This is happening in ruby core, in ruby gems and in ruby applications.

Just because there some cases where you can use either a string or a symbol does not imply that you can use a string in all cases where you can use a symbol, or vice-versa. Many methods that work with strings do not accept symbols, as using a symbol does not make sense. For example, IO.read doesn't accept a symbol, as using a symbol doesn't make sense semantically.

> This discussion as started 5 years ago in two separate feature requests (both rejected):
>
> - 5964
> - 7792
>
> I believe ruby will be much better once the ambiguity between strings and symbols is resolved for good
> and ruby 3.0 is a very good opportunity to do so.

I believe the separation of symbols and strings in ruby greatly increases the expressiveness of the language and allows the creation of libraries like Sinatra, Sequel, Cuba, and Roda. All of those libraries, and many other libraries, rely on symbols and strings being separate in order to function, and the different handling for strings and symbols is a core part of the library. I checked Rails and Rack and making symbols the same as strings would change the behavior of them as well (no case Symbol there, but check places where is_a?(Symbol) is used).

> Also what solutions for the problem are we envisioning?

> 1. Use current symbols syntax as yet another strings syntax and stop using Symbols?
> 2. Use current symbols syntax as yet another strings syntax and start use Symbols with a new syntax?
> 3. Use current symbols syntax as yet another strings syntax and use Symbols purely as a class?

If Symbols are kept at all, it doesn't make sense to change the current Symbol literal syntax to result in Strings. Assuming Symbols are kept at all, another possible option would be to add an alternative string literal syntax that saves a character, since that seems to be the primary reason that people want to use symbol literals for strings.

> From the challenge presented by Koichi I understand that the transition path to be accepted must allow the current code to raise a warning for the situation where the Symbol is not anymore a Symbol but a String.
>
> Is this assumption correct?
>
> If this is the case then all we need is to make String::===(foo) and Symbol::===(foo) to raise warnings every time foo is a string and it was created using former symbol syntax.

If you look at ruby code in the wild, is_a?(Symbol) seems even more common than case Symbol, and sometimes in places where symbol behavior differs from string behavior by design.

> Any drawbacks?

- Breaks backwards compatibility in ways that in some cases completely break the ways common libraries are currently used
- Reduces expressiveness
- Further reduces separation of concerns, merging identifiers (Symbol) into a class (String) already overloaded to handle both raw/binary data and text

> NOTE: (I'm only considering solutions 2. and 3. for the purpose of this analysis. Meaning Symbol class will still exist.)

As stated above, I don't see the point of using the current symbol literal syntax for strings if you are keeping symbols. If you need to have symbols anyway, then the only reason to use the symbol syntax for strings is to save a single character. In which case it would be better to introduce a new terser form of string literal.

I strongly urge matz to reject this for the same reasons he has rejected the previous requests for this (mis)feature.

#3 - 01/03/2018 05:43 AM - jeremyevans0 (Jeremy Evans)

duerst (Martin Dürst) wrote:

> So are you saying we should disallow
>
> `alias_method "foo", "bar"`
>
> I definitely would support that (after some depreciation period).

I would also support deprecating passing strings to methods which internally convert the strings to symbols, and raising TypeError in Ruby 3. However, I don't feel strongly about that, as it would break backwards compatibility significantly and I'm not sure the improvement is worth the cost.

#4 - 01/03/2018 05:43 AM - duerst (Martin Dürst)

dsferreira (Daniel Ferreira) wrote:

> Can we discuss here what are the rules that would allow the transition path solution to be accepted?

Koichi has given a challenge, most probably because he thinks it's very difficult if not impossible. Koichi only wrote "we can consider again".

So there are no "rules" that symbols and strings will be collapsed when certain well-defined conditions are met. In the end, the only rule is that Matz will have to decide.

Anyway, to me this looks similar to somebody coming e.g. from JavaScript or JSON and asking when Integers and Floats will be merged. JavaScript and JSON don't distinguish Integers and Floats (although some JavaScript implementations try to make the distinction under the hood for efficiency). JavaScript and JSON also don't distinguish between Symbols and Strings.

Teaching programming, and looking back at my own experience, I know that many people have difficulties at first to distinguish Integers and Floats (or whatever that's called in a different language). However, it's not too difficult to get the distinction after a little bit of time. That's the same for Symbols and Strings.

However, Ruby distinguishes Integers and Floats, and it distinguishes Symbols and Strings. While the former distinction is more popular among programming languages, the later distinction is available in quite a few, too, and came from Lisp, on of the oldest high-level programming languages.

And just in case somebody cares, I have a library where the distinction between Symbol and String is crucial (and easy to understand). So I'm not supporting such a change, even if the general transition issues could be handled.

#5 - 01/03/2018 05:49 AM - duerst (Martin Dürst)

jeremyevans0 (Jeremy Evans) wrote:

> If you need to have symbols anyway, then the only reason to use the symbol syntax for strings is to save a single character. In which case it would be better to introduce a new terser form of string literal.

On top of that: On many if not most keyboards, "'" can be typed without the shift key, but ":" needs a shift key. Thus while symbol: is shorter, 'string' shouldn't take more effort to type.

#6 - 01/03/2018 09:05 AM - dsferreira (Daniel Ferreira)

jeremyevans0 (Jeremy Evans) wrote:

> If you look at ruby code in the wild, is_a?(Symbol) seems even more common than case Symbol, and sometimes in places where symbol behavior differs from string behavior by design.

If the transition path to be accepted must allow the current code to raise a warning for the situation where the Symbol is not anymore a Symbol but a String and we allow foo to contain encapsulated the information of the syntax used to define it then we can easily update String#is_a? method to raise the warning for such cases.

#7 - 01/03/2018 09:16 AM - dsferreira (Daniel Ferreira)

jeremyevans0 (Jeremy Evans) wrote:

> To say strings and symbols are ambiguous is to imply that it is not possible to differentiate the two.

Ambiguity definition: "doubtfulness or uncertainty of meaning or intention: to speak with ambiguity; an ambiguity of manner."

I use the term "Ambiguity" to emphasise the grey area where ruby developers are using strings and symbols interchangeably without any concerns about the differences between the two types of objects.

#8 - 01/03/2018 09:30 AM - dsferreira (Daniel Ferreira)

duerst (Martin Dürst) wrote:

> So are you saying we should disallow
> ruby
> alias_method "foo", "bar"

As a ruby developer when I use an API I tend to expect that both strings and symbols are accepted since that has become the standard way of doing things in ruby land like the example highlights.
When an API doesn't met that expectation things get confusing. I believe I'm not the only one to have this mind map inferring from previous discussions.

Also what this brings is that when I design interfaces I'm impelled to force the new API to met the same expectations although I clearly dislike that approach.

> I definitely would support that (after some depreciation period).

To disallow strings over symbols or the opposite in so many interfaces it would be a very hard task not to mention an almost impossible goal to be accomplished.
Plus I believe that would kill ruby as a language. People would go away for good in my opinion because the chaos would be even worst.

Now if we accept both syntaxes in the same way and treat them always as strings we would remove all the extra work we are having currently to handle the existing differences.

#9 - 01/03/2018 09:32 AM - dsferreira (Daniel Ferreira)

duerst (Martin Dürst) wrote:

> Koichi has given a challenge, most probably because he thinks it's very difficult if not impossible. Koichi only wrote "we can consider again".

Koichi is a very busy man like most of us.
I hope he can clarify that in person.

#10 - 01/03/2018 09:37 AM - dsferreira (Daniel Ferreira)

duerst (Martin Dürst) wrote:

> And just in case somebody cares, I have a library where the distinction between Symbol and String is crucial (and easy to understand). So I'm not supporting such a change, even if the general transition issues could be handled.

I believe that it is for cases like your library that we will need the transition path. Wouldn't you be able to use Symbols and the same logic with solutions 2. or 3. in place?

Same remark and question applies for the likes presented by Jeremy.

#11 - 01/03/2018 09:47 AM - Hanmac (Hans Mackowiak)

the difference from Symbol vs String was from a time long ago before frozen String literals where added

in 99% of the cases, use Symbol if you care about the name, like keys in a hash, key parameters, or method names, or 'enum' values
use String for other cases where you care about the content of the string itself

Symbols are way more smaller in the memory, and ruby can handle them faster (from MRI point of view)
if you use the same thing multiple times, symbols only create once

Now with Frozen String literals, some of that points are not needed anymore

that so many functions that works with method names or instance variables, accept strings too, is just a nice feature for you developers
or do you want to be forced to write v.to_sym or :"#{v}"

#12 - 01/03/2018 09:58 AM - shevegen (Robert A. Heiler)

> I use the term "Ambiguity" to emphasise the grey area where ruby developers
> are using strings and symbols interchangeably without any concerns about
> the differences between the two types of objects.

This is a general problem.

I personally think that ruby would be simpler without ... Symbols. :)

But people have been using symbols and symbols were available so I guess
matz felt that symbols have a valid use case (speed, performance, lightweight
and so forth).

I personally like symbols. I use them very happily, most definitely in ways
others may not use. Like some_method(:ignore_the_cats) and handling these
cases within the method.

I also agree with Martin - while it is just a tiny difference, it is easier
to write :foo than 'foo'. It may not be of a primary concern, but I like
typing less.

And I agree as well with as to what Martin wrote that ultimately matz has
to decide whether to make a change or not concerning the usage
pattern/frequency of symbols. That also dates back to the pickaxe IMO -
the old pickaxe wrote that using symbols as keys in a hash is much better
aka faster or less memory use. But that was before frozen strings existed
by default, so I am not sure if this still applies or not ... lots of
knowledge becomes outdates.

By the way, I have no problem if

```
alias_method "foo", "bar"
```

would become deprecated. I am not even using alias_method at all in the
first place, simply because I don't like it. Just like tenderlove is a
puts debugger, and I am a pp debugger, I am an "alias foo bar" person,
even though alias and alias_method are not completely the same. I love
typing less, usually. :D

I think Koichi Sasada referred to the amount of changes required, compared
to the amount of real gain, in the symbol-situation. Matz also gave a
presentation some time ago about good and bad changes. I have no problem
with change if the change makes something better or improves on parts of
ruby - speed increases (but I prefer pretty code over min-max speed),

did-you-mean gem, omitting require 'pp' and things like that. Some changes are harder ones; encoding is not a real issue to me anymore, but it simply is different compared to the ruby 1.8.x era. Requires me to think more about string1 versus string2 that may have different, thus incompatible, encodings. (The way I work around this in my own projects is by usually specifying the main encoding to be used, in a constant; and then enforcing that in my project. This may be optional if I would use Unicode instead but I rarely do so, for reasons that take too long to explain. My main point here is just the added amount of code that one may have to deal with. Unicode at the least has emojis which I can't resist using; it is also why I keep most of my larger projects flexible. If I want to switch to unicode one day, I should be able to do so easily, by just changing a single constant.)

To some of your suggestions:

> Use current symbols syntax as yet another strings syntax and stop using Symbols?

As I wrote about above, I would not mind if symbols would not exist in the first place, largely as it may make ruby simpler. So it is not as if I would be completely opposed to your idea. BUT, since I also really like Symbols, or primarily, the syntax it has, I also don't really want to stop using them. If they are treated internally like string objects or string-like objects, I possibly would not mind really, simply because I could keep on using the syntax. But I am also wary of change just for sake of change alone.

> Use current symbols syntax as yet another strings syntax and start use Symbols with a new syntax?

This sounds really awful - I hope you don't take it personal but this would probably be the worst of all suggestions. What would be the net gain, just using a new syntax here? Would old code break or require changes?

You also did not suggest an alternative syntax to that specifically.

What syntax would you propose to then use Symbols? Would :foo still be available or would you remove it?

Syntax is also important. When I realized that @@foo is not really needed because @foo works just fine on a "module-level instance" (module Foo; @foo = 42; def self.foo?; @foo etc...) I stopped using @@foo, largely because the syntax was not appealing to me.

Unless one wants to use unicode for defining symbols, we do not have that many characters to choose from.

> Use current symbols syntax as yet another strings syntax and use Symbols purely as a class?

Similar problem to the above.

Anyway, I don't want to discourage you since I understand that this is probably harder to change compared to, say, 10 years ago or so (change takes time too and we did not have frozen strings back then). But to be honest, I do not really see any real net advantage in the proposal alone here.

I think this is an area where matz has to decide either way since it is somewhat more difficult, IMO, and only he knows how ruby should be in the end.

I also agree that it could only be for ruby 3.x to change. ko1 only gave one use case but I think there are more - in my code I sometimes do "if object.is_a? Symbol" as opposed to whether it is a String. Admittedly the code may be simpler if symbols would not exist in the first place - but since they do, and in your examples symbols would still remain, I'd be hard pressed to really want to have to lose symbols altogether.

There also have been some other issue requests where methods were added to Symbols, so I really think that this is a large topic altogether.

> As a ruby developer when I use an API I tend to expect that both strings and symbols are accepted since that has become the standard way of doing things in ruby land like the example highlights.

That depends on the API. To me it would be weird if an API rejects a symbol input just because it is a symbol, if both strings and symbols exist.

Would an API make sense if it distinguishes between 'cat' and :cat alone? It sounds a lot more as an opinionated API aimed for purism rather than practical use. I agree that people may spend time asking these questions (when to use symbols and when to use strings), so I am not against saying to remove symbols altogether IF this would lead to a simpler language (if I could still use :foo) - but if that is not done, then I'd also rather retain the way how ruby uses both. It may not be as elegant or pure as compared to the situation where we'd have no symbols, but I just do not see the net gain really.

> When an API doesn't met that expectation things get confusing.

It's simple for APIs to handle both symbols and strings.

I know that because I am using it in my code.

I do, however had, also agree that it would be simpler if ruby hackers would not have to decide between either one. Less brain-power to process. But it's hard to change at this moment in time IMO. So I am somewhat against the proposal, even though I agree with it somewhat. :)

> I believe I'm not the only one to have this mind map inferring from previous discussions.

This may well be but you can find opinions on either side of string-versus-symbol discussion. :)

> Plus I believe that would kill ruby as a language. People would go away for good in my opinion because the chaos would be even worst.

This is not a good design goal.

People will always tend to use this or that language, for this or that reason. With more choice comes more fragmentation in general. And I very much doubt that the string-versus-symbol debate is why people pick any other language. I also don't think that people often act in a very logical manner - often they make a decision and then just reason to support that decision no matter what.

It's how many people "work". ;) (Though that is not to say that they may not have valid reasons for using other languages. Speed issue is one I may tend to agree with IF it is important to someone).

There is simply more competition these days IMO. You can also see it in perl; but even PHP has lost grounds in the last ~6 years, largely due to javascript alone IMO.

> Now if we accept both syntaxes in the same way and treat them always as strings we would remove all the extra work we are having currently to handle the existing differences.

This may be one approach but you also wrote above in two of your three examples that the syntax may change or a new syntax may

be added; or that symbols may still be available, just with another
syntax. So I am confused now. Which variant do we prefer or talk
about now?

Also note that new syntax for symbols creates new/more complexity
as well. Better to not have symbols at all in this case ;) - but
as long as they exist, I like the way as it is largely because
it has been that way for a really long time.

> Wouldn't you be able to use Symbols and the same logic with
> solutions 2. or 3. in place?

But your solution requires changes there still, right?

I am not entirely sure where the real net gain is, unless you
refer to the case where there would not be any new syntax for
symbols.

#13 - 01/03/2018 10:06 AM - dsferreira (Daniel Ferreira)

jeremyevans0 (Jeremy Evans) wrote:

> Just because there some cases where you can use either a string or a symbol does not imply that you can use a string in all cases where you can use a symbol, or vice-versa.

True. In some cases strings and symbols are accepted. In same cases only strings. In other cases only symbols.

I love ruby mostly because it follows very well the principle of least surprise.
In ruby we get what we expect without to much burden.

Where is the principle of least surprise in the way we are handling strings and symbols?
(Here I'm not referring to the interfaces where we are making the distinction in a proper way, I'm referring to all the remaining interfaces coupled with the good ones. The so called ecosystem.)

All it takes is one example of handling it in different ways to set the doubt and to require extra reasoning to keep track of the interface type we are handling.
We should remove any chance to have this kind of ambiguity in ruby ecosystem.

IMHO this implies that symbols should not be allowed to be used as string replacements.
The language should make it very explicit by removing from Symbol class all the extra string goods that currently exist in it.
Interfaces should then be designed considering that strings and symbols are different objects such as strings and integers are.

Once again, IMHO Symbol class API should be totally independent from String class API.
Symbol#capitalise? Symbol#downcase? Why?

#14 - 01/03/2018 12:31 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

For those saying that typing two "" is about the same effort as a ":", you didn't actually understand the real issue. Most savings happens with the hash syntax. I take much less time to type "{a: 1}" than "{'a' => 1}".

#15 - 01/03/2018 12:49 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Since not everyone here knows Sequel, let me give an example on how strings and symbols are treated differently in that library.

```
DB[:users].select(:name).sql == %q{SELECT "name" FROM "users"}
DB[:users].select('name').sql == %q{SELECT 'name' FROM "users"}
```

Even though in the query above only the symbol version makes sense, there are many other queries where selecting a string makes sense. For example:

```
DB[:user_groups].import [:user_id, :group_name], DB[:users].select(:id, 'children').where{ age <= 10 }
```

I mean, there are legit cases where the distinction between strings and symbols are useful. If symbols didn't exist in the first place, I guess the last import statement would be written like:

```
DB['user_groups'].import ['user_id', 'group_name'], DB['users'].select('id', Sequel.lit('children')).where{
age <= 10 }
```

But if we introduce such an incompatible change now, lots of applications and libraries would require significant changes in order to be compatible with the new Ruby release introducing such changes.

I believe 90% of the pain regarding strings and symbols come from them being compared differently, specially when using as hash keys. For example:

```
my_hash = { a: 1 }
[my_hash[:a], my_hash['a']] == [1, nil]

deserialized_hash = JSON.parse(JSON.unparse my_hash)
# the above would be equivalent to the pseudo code below:
# store_in_redis 'cache_key', my_hash
# cached_hash = restore_from_redis 'cache_key'

[deserialized_hash[:a], deserialized_hash['a']] == [nil, 1]
```

While there are legit and very common usage for obj.is_a?(Symbol) and Symbol === obj (like in case statements, for example), it's much rarer to find cases where hash[:a] != hash['a'] and both are not nil. And in most of those cases it's actually a software bug, rather than the author's intention.

Maybe if we could change Hash's behavior to compare strings and symbols the same way as keys, most of the pain would have gone without breaking too much code. Also, I'm curious how much code would be broken if Ruby allowed this comparison to be true: ":my_id == 'my_id'". Code checking for obj.class, Symbol === obj (case included) and obj.is_a?(Symbol) wouldn't break. So, I guess the incompatibility would only break a few libraries and applications, but I may be wrong.

#16 - 01/03/2018 07:03 PM - dsferreira (Daniel Ferreira)

Hi Robert and remaining active participants of the conversation,

It is hard to reply to all your considerations.
The subject seems to be very complex to be discussed in a linear way.
I created this new issue in order to understand what Koichi requested in terms of transition path.
Please go read Koichi comments in the former issue to understand the full context which I tried to resume in the description.

I believe we can also address solution 1. transition path using the same logic I presented to address solution 2. and 3. in which case Symbol would act as a deprecated class.

I don't mind with what solution can be accepted.
My main focus is on the first part of the solution not the latter.

#17 - 01/08/2018 05:25 PM - mame (Yusuke Endoh)

I show one example that the difference between Symbols and Strings is actually useful for me: a trie.

```
class Trie
  def initialize
    @root = {}
  end

  def insert(key, val)
    n = @root
    key.each_char do |ch|
      n = (n[ch] ||= {})
    end
    n[:leaf] = val
  end

  def lookup(key)
    n = @root
    key.each_char do |ch|
      n = n[ch]
      return unless n
    end
    n[:leaf]
  end
end

trie = Trie.new
trie.insert("foobar", 1)
trie.insert("fooqux", 2)
trie.insert("foo", 3)
p trie.lookup("foobar") #=> 1
p trie.lookup("fooqux") #=> 2
p trie.lookup("foo")    #=> 3
p trie.lookup("foobaz") #=> nil
```

This trie implementation uses Hash objects for each node of trie. The key type of hashes is a one-letter String, or a Symbol :leaf. The one-letter String key represents a character of an edge that starts from the node, and its value of the hash is the child node. The Symbol :leaf means that the node has an associated value for the string key of a trie, and its value of the hash is the associated value of trie.
Instead of :leaf, I can use another object (for example, nil or Leaf = Object.new), but I like it because it is simple and easy to understand.

#18 - 01/08/2018 05:37 PM - dsferreira (Daniel Ferreira)

Hi Yusuke:

> I show one example that the difference between Symbols and Strings is actually useful for me

Please see my proposal: 14336

I believe proposed String#symbol? will help you on your example.

#19 - 01/08/2018 05:45 PM - mame (Yusuke Endoh)

dsferreira (Daniel Ferreira) wrote:

> I believe proposed String#symbol? will help you on your example.

How? My example needs to distinguish between hsh["leaf"] and hsh[:leaf]. (To be exact, this example is too simple to describe this issue because it uses only one-character Strings, but this distinction is actually needed when the implementation is improved to Patricia trie.) I have no idea how String#symbol? solves this issue. Could you elaborate?

#20 - 01/08/2018 06:10 PM - dsferreira (Daniel Ferreira)

mame (Yusuke Endoh) wrote:

> How? My example needs to distinguish between hsh["leaf"] and hsh[:leaf]. (To be exact, this example is too simple to describe this issue because it uses only one-character Strings, but this distinction is actually needed when the implementation is improved to Patricia trie.) I have no idea how String#symbol? solves this issue. Could you elaborate?

It is in the proposal description for 14336

Basically the idea is to treat the existing symbols as a string and add to the string the knowledge of its syntax.

```
bar = :foo
bar.class # => String
bar.symbol? # => true

baz = "foo"
baz.class # => String
baz.symbol? # => false

bar == baz # => false
```

That way code like yours that relies on the syntax to build the logic will not break.

#21 - 01/08/2018 07:07 PM - zverok (Victor Shepelev)

```
bar = :foo
bar.class # => String
bar.symbol? # => true

baz = "foo"
baz.class # => String
baz.symbol? # => false

bar == baz # => false
```

So, just to understand that we are not missing anything, bar and baz have the same **class** (String) and the same **contents** (foo), but it is two completely, totally different values based on some internal flag? Just to get rid of "unnecessary" Symbol class? Awesome. Just awesome.

#22 - 01/08/2018 07:10 PM - zverok (Victor Shepelev)

I think we should go further! Most of classes are unneccessary, in fact, they just burden the language.

```
i = 1
i.class # => Number
i.float? # => false
i.rational? # => false
i.bigdecimal? # => false

f = 1.0
f.class # => Number
f.float? # => true
f.rational? # => false
f.bigdecimal? # => false
```

```
r = 1r
r.class # => Number
r.float? # => false
r.rational? # => true
r.bigdecimal? # => false
```

WDYT? Should we move to this direction in Ruby 3?

Oh, I got one more!

```
a = []
a.class # => Collection
a.array? # => true
a.hash? # => false

h = {}
h.class # => Collection
h.array? # => false
h.hash? # => true
```

#23 - 01/08/2018 07:46 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Yeah, making :id != 'id' doesn't make sense to me either. If Ruby was going in the direction of deprecating symbols, it doesn't make sense to try to implement full backwards compatibility. I was expecting :id == 'id' to return true. But I still consider 'id'.symbol? useful to avoid breaking too much code, by handling the most common cases such as allowing the implementation of Symbol === :id and :id.is_a?(Symbol) to be backwards compatible. Of course, some code would have to break if we want to get rid of symbols.

#24 - 01/08/2018 08:08 PM - dsferreira (Daniel Ferreira)

If we assume :a == “a” # => true aren’t we saying that they are objects with the same properties?

I would say :a.value == “a”.value # => true But one has the symbol syntax as a property and the other not.

#25 - 01/08/2018 10:51 PM - spatulasnout (B Kelly)

danieldasilvaferreira@gmail.com wrote:

> I would say :a.value == “a”.value # => true But one has the
> symbol syntax as a property and the other not.

Syntax seems to have been mentioned a number of times during this
discussion, as though syntax were the distinguishing characteristic
between symbols and strings.

Symbols are conceptually distinct from strings, and the syntax used
to create either has no bearing on this fundamental distinction.

In Smalltalk, string literals are created with syntax 'foo' and
symbols are created with syntax #bar. The syntax isn't what makes
them conceptually different entities.

[https://www.gnu.org/software/smalltalk/manual/html_node/Two-flavors-of-equality.html]

A statement suggesting :a and "a" are equal, apart from one having
"the symbol syntax", reads rather strangely to one accustomed to
thinking about the differences between strings and symbols as being
independent of their syntax.

Regards,

Bill

#26 - 01/08/2018 11:01 PM - dsferreira (Daniel Ferreira)

spatulasnout (B Kelly) wrote:

> A statement suggesting :a and "a" are equal, apart from one having
> "the symbol syntax", reads rather strangely

In my comment I stated the opposite.

:a == "a" # => true would mean that the objects have the same properties

which is not the case.
Since in my proposal (now rejected) [14336]

```
foo = :a
foo.class # => String
foo.symbol? # => true

bar = "a"
bar.class # => String
bar.symbol? # => false

# Since the properties are different then I would say that

foo != bar # => true

# but since they both are String and "a"

foo.value == bar.value # => true
```

Just to explain what would be the outcome.

But since the proposal was rejected we may end the discussion around this matter.

#27 - 01/09/2018 12:56 AM - mame (Yusuke Endoh)

dsferreira (Daniel Ferreira) wrote:

> That way code like yours that relies on the syntax to build the logic will not break.

Thanks, it does not break my example, indeed. However, then, I don't see what problem the proposal solves. I think the class of objects does not matter so much because Ruby allows duck typing.

#28 - 01/09/2018 01:01 AM - dsferreira (Daniel Ferreira)

> However, then, I don't see what problem the proposal solves.

By considering symbols as strings all string methods will work so compatibility is much bigger.

I also think we can find a way of telling the hash that we want all strings keys to be non symbols once set or not.

Many possibilities here.

#29 - 01/09/2018 01:15 AM - dsferreira (Daniel Ferreira)

Also if you think about my API example:

```
alias_method :foo, :bar
alias_method "foo", "bar"
```

We would not need to care about duplicating efforts for these kind of interfaces anymore.
The doubt about if the argument expected by the API is a string, a symbol or both would not exist anymore.

Interfaces with a clear distinction like yours would be an exception and not a daily overhead.

#30 - 01/09/2018 01:30 AM - dsferreira (Daniel Ferreira)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Of course, some code would have to break if we want to get rid of symbols.

Is it a bad thing to not break backwards compatibility?
Wasn't that the aim of the transition path Koichi asked for?

We will have a chance to present warning messages alerting what will happen next.

Also I'm trying to promote a discussion. What I'm presenting is supposed to be a starting point for something that could actually work.

#31 - 01/09/2018 03:16 AM - matz (Yukihiro Matsumoto)

Of course, it is always bad to break compatibility. The point is if the benefit of the change overcomes the penalty of incompatibility. This case, I am not persuaded by the benefit of unifying.

Matz.

#32 - 01/09/2018 12:57 PM - dsferreira (Daniel Ferreira)

matz (Yukihiro Matsumoto) wrote:

> Of course, it is always bad to break compatibility. The point is if the benefit of the change overcomes the penalty of incompatibility. This case, I am not persuaded by the benefit of unifying.

Since Solution 1. was rejected in 14336 and this issue is still open does it mean you may consider a solution where Symbols are still available to the end user but in a different way like solution 2. and 3. propose?

#33 - 01/10/2018 07:06 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

I didn't reject this proposal because I wanted to keep the place you can express why you want to unify them. As I stated https://bugs.ruby-lang.org/issues/14336#note-16 strings and symbols are two totally different things. Some Ruby users may misunderstand the concept of symbols. But there's no good reason to support their misunderstanding, breaking existing code. I still haven't seen big enough reason to break compatibility.

Matz.