Ruby master - Bug #10845

Subclassing String

02/10/2015 11:41 PM - sawa (Tsuyoshi Sawada)

Status: Closed
Priority: Normal
Assignee: matz (Yukihiro Matsumoto)
Target version: ruby -v: 2.2
Backport: 2.0.0: UNKNOWN, 2.1: UNKNOWN, 2.2: UNKNOWN

Description

If I make a subclass of String, the method * returns an instance of that class.

class MyString < String
end

MyString.new("foo").*(2).class #=> MyString

This is different from other similar operations like + and %, which return a String instance.

MyString.new("foo").+("bar").class #=> String
MyString.new("%{foo}").%(foo: "bar").class #=> String

I don't see clear reason why * is to be different from + and %, and thought that perhaps either the behaviour with * is a bug, or the behaviour with + and % is a bug.

Or, is a reason why they are different?

Related issues:
Related to Ruby master - Bug #6087: How should inherited methods deal with re...
Has duplicate Ruby master - Bug #11209: [PATCH] Fix for String#+ when subclassed

History

#1 - 02/10/2015 11:58 PM - dummey (Ricky Ng)
Hmm, guessing that '+' and '%' are being a bit weird...
Verified the some thing happens in: ruby 2.0.0p481

It does look like '<<' is working (or not working) though.

irb(main):007:0> MyString.new("foo").<<("foo").class
=> MyString

--
Incoherently,
Ricky Ng

#2 - 02/11/2015 12:01 AM - sawa (Tsuyoshi Sawada)

Methods that modify the receiver and return the receiver would have to return the exact same object, which means that the class is the same. That is not the issue here.

#3 - 02/11/2015 12:18 AM - dummey (Ricky Ng)
Ahh yea, me being derpy =(. What I wanted to actually do was sanity check some other methods that would return a new_str:

irb(main):005:0> MyString.new("hello")[0].class
=> MyString
Incoherently,
Ricky Ng

#4 - 06/02/2015 07:38 PM - marcandre (Marc-Andre Lafortune)
- Assignee set to matz (Yukihiro Matsumoto)

It's clear to me that there's no rationale behind the current behavior.

The question is broader than that.

First, it's not only for String:

class MyArray < Array; end
x = MyArray.new([1,2,3])
x.first(2).class == x[0..1].class #=> false
(x+x).class == (x * 2).class #=> false

etc...

Also troubling is the fact that no constructor is called at all... Take this somewhat absurd example where @other is normally guaranteed to be a Hash (and to_s assumes that):

class MyArray < Array
  def initialize(size = 0, default = nil, **other)
    @other = other
    super(size, default)
  end

  def initialize_clone(x)
    raise "This is never called"
  end

  def initialize_dup(x)
    raise "This is never called"
  end

  def initialize_copy(x)
    raise "This is never called"
  end

  def to_s
    super + @other.keys.to_s
  end
end

MyArray.new(2, :foo, bar: 42).*{2}.to_s
# => undefined method 'keys' for nil:NilClass (NoMethodError)

The newly created MyArray has no @other because no constructor is called.

#5 - 06/04/2015 09:12 AM - nobu (Nobuyoshi Nakada)
- Has duplicate Bug #11209: [PATCH] Fix for String#+ when subclassed added
#6 - 12/16/2019 03:35 PM - mame (Yusuke Endoh)
- Related to Bug #6087: How should inherited methods deal with return values of their own subclass? added

#7 - 12/20/2019 01:25 AM - mame (Yusuke Endoh)
- Description updated

#8 - 10/25/2020 10:39 PM - jeremyevans0 (Jeremy Evans)
I've added a pull request for making String methods consistently return String instances when called on String subclasses:
https://github.com/ruby/ruby/pull/3701
For the reasons @marcandre (Marc-Andre Lafortune) mentioned, I think this makes more sense. Instance specific state is not copied over, and you can end up with invalid objects. If we are going to make similar changes for Array in 3.0 (see #6087), I think it's worth considering the same change for String.

#9 - 11/20/2020 06:16 AM - matz (Yukihiro Matsumoto)
I am OK with experimenting. But I've heard Rails relies on the old behavior (SafeStringBuffer?). In that case, it might be hard to apply this.
Matz.

#10 - 11/20/2020 07:33 AM - matsuda (Akira Matsuda)
I've heard Rails relies on the old behavior (SafeStringBuffer?)

Just commented about the impact of this patch on Rails (ActiveSupport::SafeBuffer)
https://github.com/ruby/ruby/pull/3701#issuecomment-730960288

#11 - 11/20/2020 10:17 AM - ioquatix (Samuel Williams)
Is there are logic why we should prefer to return String vs custom subclass instances? I can see argument for both way. Why is one way preferred over the other?

#12 - 11/20/2020 12:47 PM - Eregon (Benoit Daloze)
@ioquatix (Samuel Williams) See #6087. Returning subclass instances leads to all kind of issues, because there is no good way to build the subclass instance.

+1 to do it for String since we did for Array.
Maybe we should do it for Hash too?
It seems already the case for most Hash methods but not all (e.g. Hash#merge).

#13 - 11/20/2020 04:45 PM - jeremyevans0 (Jeremy Evans)
Eregon (Benoit Daloze) wrote in #note-12:

Maybe we should do it for Hash too?
It seems already the case for most Hash methods but not all (e.g. Hash#merge).

I looked at Hash#merge, but it doesn't have the same issue as the String and Array methods, since it is implemented as dup.merge!, and dup copies the state into the new object instead of losing it.

#14 - 11/20/2020 07:34 PM - Eregon (Benoit Daloze)
jeremyevans0 (Jeremy Evans) wrote in #note-13:

I looked at Hash#merge, but it doesn't have the same issue as the String and Array methods, since it is implemented as dup.merge!, and dup copies the state into the new object instead of losing it.

Indeed, I think too that case is fine.

I checked all Hash methods listed in the docs and returning a Hash and not just returning self: compact, filter, select, reject, invert, transform_keys, transform_values, and all of them return Hash, except for merge:

```ruby
h = SubHash.new
h[:foo] = 42
# Seems fine, uses #dup
```

08/28/2022
merge({}).class # => SubHash

(keep_if returns self, that confused me)

So Hash is fine already ;)

#15 - 11/21/2020 12:37 AM - jeremyevans0 (Jeremy Evans)
- Status changed from Open to Closed

Thanks to @matsuda's work showing that Rails already does something very similar and only needs a small patch to work with this, I have committed the patch at 58325daae3bee1da13ed100782cd19a89cc68771. If it causes any problems or breakage we are unwilling to accept, we can revert the patch.

#16 - 11/21/2020 05:35 AM - naruse (Yui NARUSE)

In 2.7 we break large compatibilities, and this tickets breaks Rails again. "Rails master already fixes it" doesn't care the problem; we breaks compatibility.

Breaking compatibility itself is not problem, but we can provide very small benefit for Rails application in 2.7 and 3.0. We only provide disadvantages in recent versions. Ractor and types are great milestone for future Rails, but they are not for Rails at this time.

I'm serious about the motivation of upgrading to Ruby 3.0 for Rails users. I worry that so much.

#17 - 11/21/2020 07:31 AM - jeremyevans0 (Jeremy Evans)
naruse (Yui NARUSE) wrote in #note-16:

In 2.7 we break large compatibilities, and this tickets breaks Rails again. "Rails master already fixes it" doesn't care the problem; we breaks compatibility.

In this case, Ruby's behavior resulted in objects that were not internally consistent. Rails has a fair amount of code just to work around the inconsistency, which they will be able to drop once 3.0 is the minimum version (Rails ??). See https://github.com/rails/rails/blob/1165401ee962ae63b00e21a820b108e3e7d8052220fa40/lib/active_support/core_ext/string/output_safety.rb#L48-L253

I appreciate the importance of backwards compatibility. Almost all of my libraries run on Ruby 1.9 - Ruby 3.0. However, I don't think we should skip fixing problems in core Ruby just because it breaks a small part of Rails, especially if fixing the issue in Rails is simple. We should always weigh the costs and benefits when deciding whether compatibility should be broken. That being said, I respect your opinion, and if this experiment causes too many issues, we can revert.

#18 - 11/21/2020 07:45 AM - marcandre (Marc-Andre Lafortune)

I'll ask the same question I've asked before... Why not deprecate this first?

This gives time to Rails to issue a simple update to the versions that are supported, and in a year or two the behavior can change without breaking anything.

Fixing things is good, but there is no urgency to fix this.

#19 - 11/21/2020 12:25 PM - Eregon (Benoit Daloze)

There are far more (intentional and some of them warned before) breaking changes in Ruby 3 than this, so I don't see the point. I guess Rails backported all the keyword arguments changes to previous major versions? Then maybe we can do the same for this.

An old release of e.g. Rails 5 already doesn't work on Ruby 3, so there is no point to be compatible with that. I understand being compatible with the latest Rails 5 release makes sense, though.

naruse (Yui NARUSE) wrote in #note-16:

I'm serious about the motivation of upgrading to Ruby 3.0 for Rails users. I worry that so much.

I wouldn't worry.
If people upgraded to Ruby 2.7, they'll be happy to update to Ruby 3 which has understandable keyword arguments semantics and many other things.
I think a lot of people are excited about Ruby 3.
If there was a release to worry whether people would upgrade to it it was 2.7, but it seems many people upgraded to it.
Rails Active Support CI against Ruby master branch failed. Performed git bisect and found it has been triggered since https://github.com/ruby/ruby/commit/58325daaa3beefda13ed100782cd19a89cc68771.

https://buildkite.com/rails/rails/builds/72947#6a6a6523-71a1-4e64-9de2-597a161f0aaa/994-1003

PR to fix it by @matsuda (Akira Matsuda): https://github.com/rails/rails/pull/40663