Ruby master - Feature #7604

Make === comparison operator ability to delegate comparison to an argument

12/22/2012 09:04 PM - prijutme4ty (Ilya Vorontsov)

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

Description

=begin
I propose to expand default behaviour of === operator in the following way:
Objects have additional instance method Object#reverse_comparison?(other) which is false by default in all basic classes.
Each class that overrides Object#===(other) should check whether reverse_comparison? is true or false
If it is false, behavior is not changed at all.
If it is true, comparison is delegated to === method of an argument with self as an argument.

This technique can help in constructing RSpec-style matchers for case statement. Example:

```
# usual method call
arr = %w[cat dog rat bat]
puts arr.end_with?(%w[dog bat])   # ==> false
puts arr.end_with?(%w[rat bat])   # ==> true
puts arr.end_with?(%w[bat])       # ==> true

# predicate-style case
case %w[cat dog rat bat].end_with?
when %w[dog bat]
puts '..., dog, bat'
when %w[rat bat]
puts '..., rat, bat'
when %w[bat]
puts '..., bat'
else
puts 'smth else'
end
# ==> ..., rat, bat
```

Code needed to run this is not very complex:

```
class Object
def reverse_comparison?(other)
false
end
alias_method :'old===', :'==='
def ===(other)
(other.reverse_comparison?(self) ? (other.send 'old===',self) : (self.send 'old===',other))
end
end

class Predicate
def initialize(&block)
@block = block
end
def reverse_comparison?(other)
true
end
def ===(*args)
@block.call(*args)
end
end

class Array
alias_method :'old===', :'==='
def ===(other)
```

other.reverse_comparison?(self) ? (other.send('===',self)) : (self.send('old===',other))
end

```
def end_with?(expected_elements = nil)
  return last(expected_elements.size) == expected_elements  if expected_elements
  Predicate.new{|suffix| last(suffix.size) == suffix }
end
```

end

This technique looks powerful and beautiful for me. One detail is that obj#reverse_comparison? can distinguish different types of arguments and returns true only for certain types of given object. Also this can be used to prevent double-mirroring (as shown below)

The problem is that many base classes already defined custom === operator, so each of those classes (Fixnum, Float, String, Regexp, Range etc) should be redefined in such a way to make a solution full-fledged.
Another problem is case that both objects defined reverse_comparison? to return true. In my solution Predicate#=== just ignores result of revese_comparison? which is not consistent.
Another possible way is to raise errors on double mirroring:
def reverse_comparison?(other)
raise 'double mirroring' if **@mirroring_started**
@mirroring_started = true
return true unless other.reverse_comparison?(self)
false
ensure
remove_instance_variable :@__mirroring_started
end

My proposal is to add reverse_comparison? method and change base classes operator === to use its result as shown above. May be it's worth also to make a class analogous to Predicate in stdlib.
=end

History

#1 - 12/23/2012 06:33 PM - Anonymous

Your proposal reminds me of trying to extend #coerce behavior. What you call "mirroring", happens with #coerce. "Double mirrorring" is prevented by simply by #coerce being required to return a compatible pair. That being said, I did have times, when I wanted operator-specific #coerce (eg. different physical quantities do not add or compare, but do multiply). Essentially, you are proposing:

(1.) Let us have operator-specific #coerce (for #=== at least).
(2.) Let us have #=== actually using its specific coerce for some chosen argument types.

To me, achieving (1.) is imaginable as either #coerce taking an optional second argument, as in other.coerce(self, :===), or as having special #coerce_plus, #coerce_asterisk, #coerce_double_equal_sign, #coerce_triple_equal_sign etc.

Achieving (2.) is more difficult. As you pointed out, many classes have their own #===. But it is a general case that operator methods should be written with #coerce in mind.

Having thus reframed your proposal, let me also express my personal opinion about it: I would be in favor of cautiously implementing (1.), while (2.) means a bit work for everyone. I noticed that Marc Andre was also concerned about #coerce specification.

#2 - 12/24/2012 12:49 AM - prijutme4ty (Ilya Vorontsov)

boris_stitnicky (Boris Stitnicky) wrote:

> Your proposal reminds me of trying to extend #coerce behavior. What you call "mirroring", happens with #coerce. "Double mirrorring" is prevented by simply by #coerce being required to return a compatible pair. That being said, I did have times, when I wanted operator-specific #coerce (eg. different physical quantities do not add or compare, but do multiply). Essentially, you are proposing:
>
> (1.) Let us have operator-specific #coerce (for #=== at least).
> (2.) Let us have #=== actually using its specific coerce for some chosen argument types.
>
> To me, achieving (1.) is imaginable as either #coerce taking an optional second argument, as in other.coerce(self, :===), or as having special #coerce_plus, #coerce_asterisk, #coerce_double_equal_sign, #coerce_triple_equal_sign etc.
>
> Achieving (2.) is more difficult. As you pointed out, many classes have their own #===. But it is a general case that operator methods should be written with #coerce in mind.
>
> Having thus reframed your proposal, let me also express my personal opinion about it: I would be in favor of cautiously implementing (1.), while (2.) means a bit work for everyone. I noticed that Marc Andre was also concerned about #coerce specification.

I like the idea of #coerce having additional argument(first time I thought whether current behavior of coerce can help me in solving this problem). Coercion implies that code of operators like + or === in built-in should be changed as in (2) case. I think that your solution can be actually much more flexible than mine. Also I can't realize any benefits of (2) over (1).

#3 - 12/25/2012 03:49 AM - Anonymous

(2) and (1) are two steps of the same campaign, to make the behavior you described possible, but (1) might be easier and mildly useful on its own. Current #coerce would solve the problem provided that you make it return special objects with customized multiple operator methods, similar to your Predicate. Why not make a coerce-based gem demonstrating this? I would be interested in using it personally. You would have to find and patch those scattered #=== methods, while I am more interested in :+, :-, :, :/, :*, and :<=>. We could have common special object for all of these.

#4 - 12/26/2012 11:33 AM - prijutme4ty (Ilya Vorontsov)

boris_stitnicky (Boris Stitnicky) wrote:

> (2) and (1) are two steps of the same campaign, to make the behavior you described possible, but (1) might be easier and mildly useful on its own. Current #coerce would solve the problem provided that you make it return special objects with customized multiple operator methods, similar to your Predicate. Why not make a coerce-based gem demonstrating this? I would be interested in using it personally. You would have to find and patch those scattered #=== methods, while I am more interested in :+, :-, :, :/, :*, and :<=>. We could have common special object for all of these.

I will create a proof-of-concept gem, but not sure that I'll be able to create a native extension. So arithmetical operations can become much slower.

#5 - 12/26/2012 10:38 PM - Anonymous

Let me know when you make the first commit.

#6 - 01/25/2013 12:52 PM - ko1 (Koichi Sasada)

- Category set to core

- Target version set to 2.6

#7 - 02/22/2013 09:22 AM - ko1 (Koichi Sasada)

- Assignee set to matz (Yukihiro Matsumoto)

#8 - 03/09/2013 06:41 AM - prijutme4ty (Ilya Vorontsov)

boris_stitnicky (Boris Stitnicky) wrote:

> Let me know when you make the first commit.

I released proof-of-concept gem. https://github.com/prijutme4ty/flex_coerce It makes no changes in behavior of base classes, you need to patch only your own class. But actually I didn't found any use-cases of this (e.g. physical quantities more naturally looks if there is a special quantity representing unity). I hope you'll hint me some good applications of this gem.
It's sad but #=== doesn't use coerce so this gem can't help me solve my task. So I'll soon create another gem that patches === method.

#9 - 03/13/2013 08:02 PM - Anonymous

ilya (Ilya Boltnev): I have noticed your post, I'll pay closer attention after next week.

#10 - 04/12/2013 01:38 AM - Anonymous

I'have started working on it (first I have to handle switch to 2.0 on my machine).

#11 - 04/12/2013 04:59 AM - headius (Charles Nutter)

As a feature that affects all Ruby implementations, this should probably move to CommonRuby: https://bugs.ruby-lang.org/projects/common-ruby

#12 - 12/25/2017 06:15 PM - naruse (Yui NARUSE)

- Target version deleted (2.6)