**Ruby master - Feature #6721**

**Object#yield_self**

07/11/2012 04:35 PM - alexeymuranov (Alexey Muranov)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>matz (Yukihiro Matsumoto)</td>
</tr>
</tbody>
</table>

**Description**

I think the following method is missing from Ruby:

```ruby
class Object
  def yield_self(*args)
    yield(self, *args)
  end
end
```

I do not know a good use case, but it looks very natural to me. It can be used in method chains.

What do you think? Is there an alternative?

**Related issues:**

- Related to Ruby master - Feature #7388: Object#embed
  - Rejected 11/19/2012
- Related to Ruby master - Feature #6684: Object#do
  - Rejected 07/02/2012
- Has duplicate Ruby master - Feature #11717: Object#trap -- pass object to blo...
  - Closed
- Has duplicate Ruby master - Feature #10095: Object#as
  - Closed
- Has duplicate Ruby master - Feature #12760: Optional block argument for ‘itself’
  - Closed

**Associated revisions**

Revision cec06682 - 05/01/2017 07:50 AM - nobu (Nobuyoshi Nakada)

object.c: Kernel#yield_self

- object.c (rb_obj_yield_self): new method which yields the receiver and returns the result.
  [ruby-core:46320] [Feature #6721]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@58528 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 58528 - 05/01/2017 07:50 AM - nobu (Nobuyoshi Nakada)

object.c: Kernel#yield_self

- object.c (rb_obj_yield_self): new method which yields the receiver and returns the result.
  [ruby-core:46320] [Feature #6721]

Revision 58528 - 05/01/2017 07:50 AM - nobu (Nobuyoshi Nakada)

object.c: Kernel#yield_self

- object.c (rb_obj_yield_self): new method which yields the receiver and returns the result.
  [ruby-core:46320] [Feature #6721]

Revision 58528 - 05/01/2017 07:50 AM - nobu (Nobuyoshi Nakada)

object.c: Kernel#yield_self

- object.c (rb_obj_yield_self): new method which yields the receiver and returns the result.
  [ruby-core:46320] [Feature #6721]

History
How is this significantly different than Object#tap?

It executes the block and returns its output. For example:

```
2.yield_self { |x| x*x } # => 4
```

I've come up with some use case for illustration. I have also looked into the Ruby on Rails Object#try method because it can serve a similar purpose. I think yield_self is more basic than try.

Here are two examples of a use case:

```
attr = object.associated_object.yield_self { |o| o.attribute unless o.nil? }

mailing_address = { :name => person[:name],
                  :street => person[:address].yield_self { |a| a[:street] if a.is_a?(Hash) }
}
```

Here is for comparison the implementation of Object#try in Ruby on Rails:

```
def try(*a, &b)
  if a.empty? && block_given?
    yield self
  else
    __send__(*a, &b)
  end
end
```

I'm not against the feature itself, but don't like the name.

nobu (Nobuyoshi Nakada) wrote:

I'm not against the feature itself, but don't like the name.

Or otherwise, #apply:

```
2.apply { |x| x*x } # => 4
```

Your current names are less clear than using a local variable. Using a local variable reveals your intentions very clearly:

```
o = object.associated_object
attr = o.attribute if o
```

It's obvious that attr is only set if the associated object exists.

For your second example there's just too much going on to clearly see what the intention is. By first separating data gathering from creating of the mailing_address Hash things become much clearer:

```
address = person[:address]
street = address[:street] if address.is_a?(Hash)

mailing_address = {
  :name => person[:name],
  :street => street,
}
```

As in the first example, your current names don't reveal what yield_self is supposed to do in a way that's clearer than using local variables for construction of mailing_address.
Your current names are less clear than using a local variable. Using a local variable reveals your intentions very clearly:

Well, using method chains with blocks is always less clear than using local variables, i think.

**#8 - 08/07/2012 06:14 AM - trans (Thomas Sawyer)**

This is basically #ergo in Ruby Facets. Essentially:

```ruby
def ergo
  return yield(self) if block_given?
  self
end
```

**#9 - 10/28/2012 12:18 AM - yhara (Yutaka HARA)**

- Category set to core
- Target version set to 2.6

**#10 - 11/10/2012 07:19 PM - alexeymuranov (Alexey Muranov)**

After commenting on #6284, i have a new proposition for this method's name: Object#^*. Also, i suggest to allow it to take a block, a proc, a lambda, or a symbol. I think this will not conflict with existing uses of #^, however the classes that implement it for certain argument types should not forget to call super if the argument type is not recognized by them.

For example:

```ruby
# Formatting a string:
format_as_title = lambda { |str| "Title: #{str.strip.capitalize}" }
title = "something to be a title" ^ format_as_title
# instead of "format_as_title("something to be a title")"

# Squaring the 2:
four = 2 ^ { |x| x*x }
# instead of `four = 2 * 2`

# Converting a string to an integer:
five = "5" ^ :to_i
# instead of `five = "5".to_i`
```

This is consistent with a rare mathematical notation for function application: sometimes instead of "f(x)", the "exponential" notation "x^f" is used.

This would also open a door to compose lambdas from left to right, if the majority decides so (this is being discussed in #6284).

**#11 - 11/18/2012 12:20 PM - Anonymous**

#ergo is a well-thought method name, I like it better than all others.

**#12 - 11/25/2012 02:28 AM - headius (Charles Nutter)**

It occurs to me #apply is used in some other languages to refer to the elements of a collection rather than to the collection itself.

```ruby
[1,2,3].apply {|n| puts n}
```

Did we ever decide if the #self method would be added? If it were, it would be simple to have it take a block:

```ruby
four = 2.self { |n| n * n }
```

That would make #self basically be #ergo as defined by Facets.

Worth noting that you can get nearly as concise syntax today, albeit in reverse order:

```ruby
four = ->{|n| n * n}.(2)
```

**#13 - 02/08/2013 10:41 PM - alexeymuranov (Alexey Muranov)**

Here is a "real life" use case. It again has to do with formatting strings.

I want to have a list of conference participants in the form:

Full Name (Affiliation, academic position)

but without empty parentheses or trailing comma if the person has not provided the affiliation or the position. So i did like this:

```ruby
class Participant
  def full_name_with_affiliation_and_position
    full_name +
    lambda { |x| x.empty? ? '' : " \{#{x}}" }[[affiliation, academic_position].compact.join(', ')]
end
```

09/01/2022
(I will appreciate any more elegant solution.)

With \texttt{#yield\_self} (or any other name for it), i would have written:

```ruby
class Participant
  def full_name_with_affiliation_and_position
    full_name = [affiliation, academic_position].compact.join(' , ').yield_self { |x| x.empty? ? '' : "(#{x})" } end
  end
end
```

This would be a bit more readable for me.

\emph{Edited 2013-02-09.}

**#14 - 02/09/2013 05:23 AM - Anonymous**

Why you can't simply do the following?

```ruby
def full_name_with_affiliation_and_position
  a_ap = "(#{a_ap})" unless [a_ap = [affiliation, academic_position].compact.join ' , '].empty?
  "#{full_name}#{a_ap}"
end
```

**#15 - 02/09/2013 08:42 PM - alexeymuranov (Alexey Muranov)**

Anonymous wrote:

```
def full_name_with_affiliation_and_position
  a_ap = "(#{a_ap})" unless [a_ap = [affiliation, academic_position].compact.join ' , '].empty?
  "#{full_name}#{a_ap}"
end
```

```
I can, but i guess i want it to look more like declarative programming, than like imperative.
```

**#16 - 02/18/2013 09:21 AM - ko1 (Koichi Sasada)**

- Assignee set to matz (Yukihiro Matsumoto)

**#17 - 05/17/2013 05:01 PM - aleph1 (Elias Levy)**

nobu (Nobuyoshi Nakada) wrote:

```
I'm not against the feature itself, but don't like the name.
```

At its core this feature relates to method chaining and transforming the object, something that cannot be done with Object\#tap.

Some suggested names then: transform, alter, mutate, map, morph.

map may be the best choice, as its already used in enumerables and this is a natural equivalent for single objects. That said, it may lead to unnoticed bugs if someone thinks they are applying a map operation on an enumerable but for some reason they do so against some other object. So maybe one of the other names is better to ensure such cases fail.

**#18 - 05/17/2013 05:53 PM - nobu (Nobuyoshi Nakada)**

(13/05/17 17:01), aleph1 (Elias Levy) wrote:

```
map may be the best choice, as its already used in enumerables and this is a natural equivalent for single objects. That said, it may lead to unnoticed bugs if someone thinks they are applying a map operation on an enumerable but for some reason they do so against some other object. So maybe one of the other names is better to ensure such cases fail.
```

If it were Kernel\#map, which would you expect by \{foo: 42\}.map{...}?

**#19 - 05/18/2013 05:41 AM - Anonymous**

nobu (Nobuyoshi Nakada) wrote:
I'm not against the feature itself, but don't like the name.

+1 to this opinion

#20 - 05/18/2013 05:21 PM - alexeymuranov (Alexey Muranov)
I have checked if by any chance Haskell had it, apparently it doesn't:
http://stackoverflow.com/questions/4090168/is-there-an-inverse-of-the-haskell-operator

I have found that in Alonzo Church's "The calculi of lambda-conversion", he uses "[M^N]" (with superscript) as an alternative notation for the "application" of a term N to a term M (in addition to the basic lambda-calculus notation "(NM)").

This operation would roughly correspond to apply in Scheme with reverse order of arguments. If i understand correctly, the apply in Scheme roughly corresponds to call in Ruby, so maybe reverse_call?

Edited 2013-08-31.

#21 - 09/01/2013 05:32 AM - alexeymuranov (Alexey Muranov)
Another idea for the name: Object#cast.

```ruby
class Object
  def cast(*args)
    if block_given?
      yield(self, *args)
    else
      p = args.pop
      unless p.respond_to?(:call)
        raise ArgumentError, 'the last argument should be callable when no block is given'
      end
      p.call(self, *args)
    end
  end
end
```

2.cast {|x| x**3 } #=> 8

```
p = proc {|x, y| x**y }
3.cast(2, p) #=> 9
```

Another option: Object#toss.

Edited 2013-09-01.

#22 - 09/29/2013 08:04 PM - abinoam (Abinoam P. Marques Jr.)
May I give a name suggestion?
Does "tap!" make sense in english?

```
2.tap {|x| x*2 } #=> 2
2.tap! {|x| x*2 } #=> 4
```

The exclamation mark alerts that the return value is being changed.

#23 - 11/20/2015 06:03 PM - nobu (Nobuyoshi Nakada)
- Related to Feature #6688: Object#replace added

#24 - 11/20/2015 06:04 PM - nobu (Nobuyoshi Nakada)
- Related to deleted (Feature #6688: Object#replace)

#25 - 11/20/2015 06:04 PM - nobu (Nobuyoshi Nakada)
- Related to Feature #6684: Object#do added

#26 - 11/20/2015 06:06 PM - nobu (Nobuyoshi Nakada)
- Related to Feature #10095: Object#as added

#27 - 11/20/2015 06:06 PM - nobu (Nobuyoshi Nakada)
- Has duplicate Feature #11717: Object#trap -- pass object to block and return result added
object.c: Kernel#yield_self

- object.c (rb_obj_yield_self): new method which yields the receiver and returns the result.
[ruby-core:48390] [Feature #6721]