Ruby trunk - Feature #10095

Object#as

07/26/2014 06:53 AM - matsuda (Akira Matsuda)

Status:	Closed
Priority:	Normal
Assignee:	
Target version:	

Description

We've had so many times of feature requests for a method similar to Object#tap that doesn't return self but returns the given block's execution result (e.g. #7388, #6684, #6721).

I'm talking about something like this in Ruby of course:

```
Object.class_eval { def as() yield(self) end }
```

IIRC Matz is not against introducing this feature but he didn't like any of the names proposed in the past, such as embed, do, identity, ergo, reference, yield_self, itself, apply, map, tap!, etc.

So, let us propose a new name, Object#as today.
It's named from the aspect of the feature that it gives the receiver a new name "as" a block local variable.
For instance, the code reads so natural and intuitive like this:

```
(1 + 2 + 3 + 4).as {|x| x ** 2}
=> 100

Array.new.as {|a| a << 1; a << 2}
=> [1, 2]
```

Related issues:

Is duplicate of Ruby trunk - Feature #6721: Object#yield_self	**Closed**
Has duplicate Ruby trunk - Feature #12760: Optional block argument for `itself`	**Closed**
Has duplicate Ruby trunk - Feature #13172: Method that yields object to block...	**Closed**

History

#1 - 07/26/2014 09:10 AM - sawa (Tsuyoshi Sawada)

I would like to propose the name chain.

```
[1, 2, 3, 4].select(&:odd?).chain{|x| {total: x.count, data: x}}
```

As I looked in the related threads, it looks like there is a consensus that what this method would do is method chaining. I think chain is the word that describes the nature of this method.

#2 - 07/26/2014 09:37 AM - knu (Akinori MUSHA)

Here's some of the candidates I thought up in today's dev meeting:

- into
- turn
- map1
- apply

#3 - 07/26/2014 06:26 PM - ko1 (Koichi Sasada)

I found that Ocaml has a function "revapply".

```
revapply x f (* it means f(x) *)
```

How about "rap" (shorter name of revapply)?
Similar to "tap" :)

(bike shed)

#4 - 07/28/2014 02:30 AM - sawa (Tsuyoshi Sawada)

What about unifying this feature with this feature: https://bugs.ruby-lang.org/issues/6373? Let the block be optional; when there is one, it returns what this thread originally expected, and when there is no block, it returns the receiver. In that case, a name like self_by may make sense.

```
[1, 2, 3, 4].select(&:odd?).self_by{|x| {total: x.count, data: x}}

[1, 2, 3, 2, 2, 1].group_by(&:self_by)
```

#5 - 08/07/2014 01:33 PM - shevegen (Robert A. Heiler)

Guys, your chosen names are awful so far. :)

revapply is ugly, it does not fit to ruby.

#as is not descriptive enough.

In english language, remember:

"I am as big as him."

So using "as" is not a good name either.

.chain is not bad but not ideal either, because when I read it, I wonder where the chain is (remember: foo.bar.bla.ble is also a chain, a method invocation chain)

I quite like the idea behind self_by, not sure I like self_by but I think it is a better name than .as

how about .blocK_self ?

or perhaps a yield_ name ...

yield_block

#6 - 08/07/2014 10:07 PM - avit (Andrew Vit)

I think this should be added as an optional block on the newly added #itself method, and not a new method.

```
"ruby".itself                 #=> "ruby"
"ruby".itself(&nil)           #=> "ruby"
"ruby".itself {|s| s.upcase } #=> "RUBY"
```

Since there are other method names proposed here, I originally proposed the method name just #yield for this, but there was no feedback on that. I think it reads better than #itself and has symmetry with the yield keyword.

```
# yield as a noun means "the result": an object's result (its yield) is itself
"ruby".yield  #=> "ruby"

# yield as a verb means "give way to" or "produce": the object gives way to the block
"ruby".yield {|s| s.upcase } #=> "RUBY"
```

Was this considered, and were there objections? Maybe self.yield vs. yield might be confusing.

That said, I'm fine with #itself. Let's add a block option?

#7 - 08/08/2014 01:31 AM - phluid61 (Matthew Kerwin)

Andrew Vit wrote:

> I think this should be added as an optional block on the newly added #itself method, and not a new method.

I agree with a block form of #itself.

#8 - 08/08/2014 02:13 AM - phluid61 (Matthew Kerwin)

- File itself-block.patch added

Here is a patch that adds a block to #itself, with tests.

(Sorry, my editor inserted a TAB instead of spaces)

#9 - 08/08/2014 03:05 PM - rafaelfranca (Rafael França)

I believe using #itself for this feature will cause confusion. By what I could understand of the original proposal its idea is to return the result of the block instead of the objects itself.

The idea behind #itself is to return the object. If we add support to a block and make the method return the result of the block we are just going against the original idea of #itself.

#10 - 08/08/2014 10:09 PM - phluid61 (Matthew Kerwin)

On 09/08/2014, rafaelmfranca@gmail.comrafaelmfranca@gmail.com wrote:

> Issue #10095 has been updated by Rafael França.
>
> I believe using #itself for this feature will cause confusion. By what I could understand of the original proposal its idea is to return the result of the block instead of the objects itself.
>
> The idea behind #itself is to return the object. If we add support to a block and make the method return the result of the block we are just going against the original idea of #itself.

I suppose examples will help illustrate what seems most clear. This is a short snippet along the lines of what I think makes this proposed method useful: a (long?) chain of methods that are prefixed/wrapped towards the end:

```
n = gets.chomp.as{|i| Integer(i)}
n = gets.chomp.itself{|i| Integer(i)}

n = gets.chomp.as do |i|
  Integer(i)
end
n = gets.chomp.itself do |i|
  Integer(i)
end
```

In short short form #as seems appealing, but the long form it seems to me to be too messy. #itself never seems ambiguous to me, but then I'm already aware of #tap and can see that this is different.

My preference is still for #itself

Incidentally, I think #yield is a bad name because it does the exact opposite thing depending on whether you pass it a block:

```
def foo &b
  yield {|o| ... }
  yield ...
end
```

--
Matthew Kerwin
http://matthew.kerwin.net.au/

#11 - 08/19/2014 01:00 PM - trans (Thomas Sawyer)

I agree with Rafael, #itself isn't the right method.

It reminds me of #send more than anything else.

```
(2 + 3).send{ |x| x + 2 }
```

#12 - 08/20/2014 04:52 AM - Leonid (Leo Vi)

Since it's a core language extension, it might as well be a syntax extension, something like:

```
(2+3).~> {|e| e+2}.~> {|num| Mail.find(num)}.~>{|mail| Messager.send mail }
```

and maybe a shortcut optional version for that would be even better

```
(2+3)~> { @+2}~>{ Mail.find(@) } ~> { Messager.send @ }
```

(In our projects we use Object#tap! for that.
(class Object; def tap!; yield self end end)

It's very logical since many people use ActiveSupport language extensions as a de facto standard, and it already has a Object#tap version.)

#13 - 09/20/2014 02:48 AM - jihwans (Jihwan Song)

I wonder what is the biggest reason that yield may not be the word....

```
class Object
  def yield(*args, &blk)
    yield(self, *args)
  end
end

puts 1.yield(2, 3) { |one, two, three| one + two + three }
[1, 2, 3, 4].select(&:odd?).yield {|x| {total: x.count, data: x}}
```

#14 - 09/20/2014 07:35 AM - phluid61 (Matthew Kerwin)

Jihwan Song wrote:

> I wonder what is the biggest reason that yield may not be the word....

My biggest concern remains that the keyword yield works in exactly the opposite direction from your proposed method. The keyword passes execution back up to a function from a higher scope, the method passes it down to a lower. I don't know how to clearly articulate it, but it feels completely wrong to me.

However I also have some linguistic/semantic problems, because of the definitions of the word "yield". As an intransient verb it means "surrender, or step back"; as a transient verb (with a direct object) it means "provide ".

To arrive at the first definition, the subject has to be something that is active and capable of stepping back. For example a Thread, or a process handle, or something. I'd expect Thread.current.yield to bump the current thread back, and let any waiting threads execute.

For the second definition, you either need to tell it what to yield (as in a property accessor), or have it be obvious (for example a collection like an Array would obviously yield its members).

```
puts 1.yield(2, 3) { |one, two, three| one + two + three }
```

To me -- knowing the definition of the word "yield", but not knowing the method -- the expression 1.yield(2, 3) doesn't make sense. The example only works because you called the variables one, two, and three. Incidentally it's not a very good example because the arguments are provided left-to-right and used left-to-right ... (i.e. why bother, just use puts 1 + 2 + 3)

```
[1, 2, 3, 4].select(&:odd?).yield {|x| {total: x.count, data: x}}
```

At first I had trouble understanding the intent of this example; to my mind Array#yield looks like some sort of #each operation, or in this case (because of the apparent aggregate "count" operation) something like #group_by

I'm starting to think a non-word like revapply is better, because it doesn't conflict with a keyword, and it doesn't carry any expectation from any natural languages. My preferences are:

- if it doesn't accept args, use #itself
- if it does accept args, call it something like #revapply, and have the non-block form return an Enumerator

#15 - 09/20/2014 11:19 AM - jihwans (Jihwan Song)

Matt,

I guess I had totally different understanding of the Ruby keyword 'yield'. When I first saw the language years back, at the statement saying 'yield' I was speaking to myself "yield what??" -- I always took it as meaning "produce" (yeah, I always thought the 'yield' was a bad choice as the keyword for it in Ruby because it can be confused with other meanings of the word 'yield'... well, on the other hand, it may be the best word for it for the same reason :))

Once I understand the language, I always took it as transient verb with implied direct object, which is the block passed to it.

Actually, the examples I included runs beautifully in current definition of Ruby language. The point of my sample code is to show you that it is totally coherent with current definition of Ruby language. i.e. to define new 'yield' I only used current 'yield'

Now, with this extension of current 'yield', without hurting its original meaning both in English and Ruby, it can now specify subject and direct object explicitly... that's all.

Yea, ten acres of vineyard shall yield one bath, and the seed of an homer shall yield an ephah.

- Isaiah 5:10

```
class Object
  def yield(*args, &blk)
    yield(self, *args)
  end
```

```
end

Acre.new(10).yield {|acre| acre.count / 10 * bath}
Homer.new(1).yield {|homer| homer.count * ephah}
```

#16 - 09/20/2014 11:54 AM - phluid61 (Matthew Kerwin)

Jihwan Song wrote:

> Matt,
>
> I guess I had totally different understanding of the Ruby keyword 'yield'. When I first saw the language years back, at the statement saying 'yield' I was speaking to myself "yield what??" -- I always took it as meaning "produce" (yeah, I always thought the 'yield' was a bad choice as the keyword for it in Ruby because it can be confused with other meanings of the word 'yield'... well, on the other hand, it may be the best word for it for the same reason :))

How often do you see the keyword yield without arguments? Even in your examples you are clearly passing arguments (direct objects) to it.

> Once I understand the language, I always took it as transient verb with implied direct object, which is the block passed to it.

But you don't pass a block to the keyword; you're not saying "yield this block." Rather, you're instructing the scope (i.e. the function) to yield *values* (direct objects) *to the block* (indirect object).

Translating from Ruby to English:

```
def foo *a, &b
  yield *a
end
```

...becomes...

"foo, yield *a to &b"

So the subject is the function, the direct object is the args (which defaults to an empty list), and the indirect object is the block. If you explicitly name the subject, that subject has to "possess" the direct object, and then yield it to the indirect object (e.g. a function can resolve the variable a to an object and yield it; an array can resolve the index 0 to an object and yield it; etc.)

> Actually, the examples I included runs beautifully in current definition of Ruby language. The point of my sample code is to show you that it is totally coherent with current definition of Ruby language.

I still assert that it isn't coherent. Since the method receiver is the first (or only) of the direct objects, the message you are sending it should be "be yielded ...", not "yield".

> Now, with the extension of current work yield, without hurting the original meaning both in English and Ruby, it now can specify subject and direct object explicitly... that's all.

No, because 1.yield(2) means: "1, yield the value 2 to ..." which is nonsensical. What you want to say is: "ruby, yield the values (1,2) to ..." or put another way "1, be yielded along with 2 to ..."

Best to leave the word "yield" out of the verb altogether.

#17 - 09/20/2014 12:15 PM - jihwans (Jihwan Song)

Matthew Kerwin wrote:

> How often do you see the keyword yield without arguments? Even in your examples you are clearly passing arguments (direct objects) to it.
> Well, arguments are not he one 'yield' shall yield, but the block is...
>
> But you don't pass a block to the keyword; you're not saying "yield this block." Rather, you're instructing the scope (i.e. the function) to yield *values* (direct objects) *to the block* (indirect object).

Actually, I always thought I instructed Ruby to yield the block, arguments being specifics.

> Translating from Ruby to English:
>
> ```
> def foo *a, &b
>   yield *a
> end
> ```
>
> ...becomes...

> "foo, yield *a to &b"

Rather, "yield &b with *a" -- or properly, within a context of OOP, "o.yield(*a) { how-to-yield-instruction }" reads "o shall yield the block(instructions given), with parameter *a". in short:

"o, yield &b with *a"

> So the subject is the function, the direct object is the args (which defaults to an empty list), and the indirect object is the block. .....

I'd say.. the subject is the class instance, direct object is the block args are specifics. like "I.yield(red) {|o| bread} if morning"

> I still assert that it isn't coherent. Since the method receiver is the first (or only) of the direct objects, the message you are sending it should be "be yielded ...", not "yield".

The following code almost reads "Object, yield is to yield with more specific." showing coherency.

```
class Object
  def yield(*args, &blk)
    yield(self, *args)
  end
end
```

Well, I did not like the part subject being the first passed parameter to the block. That, I did it that way only to show how it can be done with current Ruby. However, I think it may be much better if the new 'yield' verb passes the subject implied. So that following should be enough:

```
Milk.yield {butter = stir(30.minutes)}
Vineyard(10.acre).yield { Grape((count / 10).bath) }
```

#18 - 09/20/2014 10:19 PM - jihwans (Jihwan Song)

I realized that what is desired here is somewhat different than what can be achieved using class_eval, instance_eval and/or yield.

What is really desired is a way to define an unnamed instance method that has access only to public methods, that gets defined immediately and disappears(undefined) right after it was invoked once.

Let's say, we have this tiny functionality that probably never will be used again and need not be included in the definition of the class, but we wanna do it quickly by saying something like this:

```
john.yield(Time.now()) { |time| hungry? ? (time.is_morning? ? :eat_bagel : :eat_bread) : :sleep }
```

The context the block should be running in has to be within the instance, john. 'hungry?' is applied to john as if the block is defined as a instance method in the class.

However, this kind of usage of the instance should not break accessibility rule to make any exception. Although the block was programmed as if it was a def block in the class, it should not have any access to private methods.

#19 - 09/20/2014 10:29 PM - phluid61 (Matthew Kerwin)

Jihwan Song wrote:

> I realized that what is desired here is somewhat different than what can be achieved using class_eval, instance_eval and/or yield.
>
> What is really desired is a way to define an unnamed instance method that has access only to public methods, that gets defined immediately and disappears(undefined) right after it was invoked once.

What you're requesting is drastically different from the original proposal (i.e. a version of #tap that returns the result of the block, rather than the receiver). You should open a new ticket for it, if you want to pursue it.

#20 - 09/21/2014 01:47 PM - jihwans (Jihwan Song)

I think the goal is always to keep Ruby language beautiful.
The original request was done with certain motive. What is desired is to find a Ruby way, targetting the original motive. Simply following requests will end up with hairy language, I am afraid...

Not tha what I suggest must be the ideal way, but it is worth going through thisconversation to find the Ruby way.

Anyways.. I thought yield was good word for it, then later it could be thought of a unnamed method - then yield may not be the word.
I simply could not figure out a good name for it...

I also want to point out that this language structure is mostly for some simple stuff, i would imagine. Then allowing parameterized block may be overkill for it. Then we end up with a block with no parameter, the object being passed into the block implicitly, where the subject of all the verbs in the

block being the object passed. - in this case, yield is okay for the name... This may be more convinient because in this cae, using other objects defined outside the block is no issue, unlike unnamed method approach.

#21 - 09/21/2014 03:06 PM - jihwans (Jihwan Song)

I should admit that I never liked object to which a block method is called being passed as block parameter.

Even for tap, for example,

```
(1..10).tap{ |x| logger.debug x }
```

looks a bit < u guess >. I'd like to say

```
(1..10).tap{ logger.debug self }
```

This is totally different with other cases where block parameters are used:

```
members.each { |member| member.shake_hands }
```

Here, member is not same object as members.

However, in case of tap or this new feature we are discussing, or other cases where I had to pass self as block parameter, I always felt something is not right.

#22 - 09/26/2014 04:33 PM - avit (Andrew Vit)

This feature also looks a lot like a pipeline.

Here are some examples how people implemented that before, either using simple blocks or more complex generator/consumer queues:

https://gist.github.com/pcreux/2f87847e5e4aad37db02
https://github.com/meh/ruby-thread#pipe
http://pragdave.me/blog/2007/12/30/pipelines-using-fibers-in-ruby-19/

For the purposes of this feature the method name could be something like pipe_to. (Hopefully that's not confusing with IO.pipe.)

Eventually, this could be an opportunity for pipelines as a first-class concept with better syntax sugar, similar to one of the examples above, or like Elixir (|>). Someone else can make a good proposal for that, but to think ahead maybe this method name should express "pipe" as part of its name.

Here is my own naive example:

```
class Object
  def | other
    case other
    when Proc
      if self.is_a?(Proc)
        proc { |input| other.call( self.call(input) ) }
      else
        other.call(self)
      end
    else
      super
    end
  end
end

"ruby" |-> r { r.upcase }
#=> "RUBY"

pipeline = &:upcase | &:reverse
"ruby" | pipeline
#=> "YRUB"
```

#23 - 10/01/2014 12:13 AM - baweaver (Brandon Weaver)

Andrew Vit wrote:

> This feature also looks a lot like a pipeline.
>
> Here are some examples how people implemented that before, either using simple blocks or more complex generator/consumer queues:
>
> https://gist.github.com/pcreux/2f87847e5e4aad37db02
> https://github.com/meh/ruby-thread#pipe
> http://pragdave.me/blog/2007/12/30/pipelines-using-fibers-in-ruby-19/

For the purposes of this feature the method name could be something like pipe_to. (Hopefully that's not confusing with IO.pipe.)

Eventually, this could be an opportunity for pipelines as a first-class concept with better syntax sugar, similar to one of the examples above, or like Elixir (|>). Someone else can make a good proposal for that, but to think ahead maybe this method name should express "pipe" as part of its name.

Here is my own naive example:

```
class Object
  def | other
    case other
    when Proc
      if self.is_a?(Proc)
        proc { |input| other.call( self.call(input) ) }
      else
        other.call(self)
      end
    else
      super
    end
  end
end

"ruby" |-> r { r.upcase }
#=> "RUBY"

pipeline = &:upcase | &:reverse
"ruby" | pipeline
#=> "YRUB"
```

related: #10308

I've actually done it too before, I just literally named it pipe:

https://github.com/baweaver/pipeable

or Stream which was something to the same idea:

https://github.com/baweaver/streamable

Either way this has been hacked to death in multiple places. I'd think that just going for an Elixir like pipe would be the cleanest implementation honestly.

#24 - 05/21/2015 08:55 AM - tallakt (Tallak Tveide)

I created a gem galled object_as to provide this functionality outside of Ruby core. I also would like to see this included in Ruby core. I think the name Object#as is short and sweet.

#25 - 11/20/2015 06:06 PM - nobu (Nobuyoshi Nakada)

- Related to Feature #6721: Object#yield_self added

#26 - 11/20/2015 06:07 PM - nobu (Nobuyoshi Nakada)

- Related to deleted (Feature #6721: Object#yield_self)

#27 - 11/20/2015 06:08 PM - nobu (Nobuyoshi Nakada)

- Is duplicate of Feature #6721: Object#yield_self added

#28 - 04/27/2016 02:01 PM - soutaro (Soutaro Matsumoto)

How about Object#continue?

```
(1 + 2 + 3 + 4).continue {|x| x ** 2}
```

The name is brought from the concept of continuation; instead of assigning the value of expression to variable, pass the value to given computation (continuation).

#29 - 04/27/2016 04:47 PM - sawa (Tsuyoshi Sawada)

I think this feature is already realized by instance_eval given that it accepts an optional block parameter, but people do not want to use it because it is slow (and perhaps because its name is too long). The reason instance_eval is slow is because of the cost of switching the context. However, I don't see any reason it has to switch the context when a block parameter is given. I cannot think of a use case where one wants to access the receiver

simultaneously through the switched self and the block parameter. Either one at a time is enough.

So, I would like to make a proposal such that, when instance_eval takes a block parameter, do not switch the context, and just pass the receiver through the block parameter. This will remove the slowness of the method in the relevant case, and provides the feature we want.

It is a backward incompatible change, but I suspect there are not much code that accesses the receiver through self and the block parameter at the same time.

#30 - 04/28/2016 05:45 AM - shyouhei (Shyouhei Urabe)

Tsuyoshi Sawada wrote:

> So, I would like to make a proposal such that, when instance_eval takes a block parameter, do not switch the context, and just pass the receiver through the block parameter. This will remove the slowness of the method in the relevant case, and provides the feature we want.

-1. You cannot access for example instance variables without changing context. instance_eval's changing context is the nature of eval. Please don't.

#31 - 09/20/2016 12:55 AM - nobu (Nobuyoshi Nakada)

- Has duplicate Feature #12760: Optional block argument for `itself` added

#32 - 01/31/2017 04:15 AM - nobu (Nobuyoshi Nakada)

- Has duplicate Feature #13172: Method that yields object to block and returns result added

#33 - 04/30/2017 07:23 PM - janosch-x (Janosch Müller)

The weakness of "as" is that it is not a verb. This puts it at odds with the majority of core methods and makes it confusing when followed up by more method calls.

Without a follow-up it is indeed quite readable:

```
number.as { |x| x ** 2 }
```

Treat the number as it's square. Fair enough!

With more stuff chained, though, it becomes confusing, because a sentence like "Treat A as B" has a certain finality to it. In a natural language you would not say something like "Treat A as B as C". Thus a longer chain becomes hard to read:

```
number.as { |x| x ** 2 }.next.as { |x| x ** 2 } # => 10201
```

Using a verb instead makes it much easier to visualize the flow of events:

```
number.convert { |x| x ** 2 }.next.convert { |x| x ** 2 }
```

#34 - 05/01/2017 07:50 AM - nobu (Nobuyoshi Nakada)

- Description updated

I think that #convert suggests something is changed by the method itself.
(I like #let)

Anyway, since Matz has accepted #yield_self months ago already1, now introduce it.

#35 - 05/01/2017 07:50 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset trunk|r58528.

---

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]

Files

itself-block.patch	1.35 KB	08/08/2014	phluid61 (Matthew Kerwin)