CommonRuby - Feature #13581

Syntax sugar for method reference

05/19/2017 12:44 PM - americodls (Americo Duarte)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		

Description

Some another programming languages (even Java, in version 8) has a cool way to refer a method as a reference.

I wrote some examples here: https://gist.github.com/americodls/20981b2864d166eee8d231904303f24b

I miss this thing in ruby.

I would thinking if is possible some like this:

```
roots = [1, 4, 9].map &Math.method(:sqrt)
```

Could be like this:

```
roots = [1, 4, 9].map Math->method
```

What do you guys thinking about it?

Related issues:

Related to Ruby master - Feature #16275: Revert `.:` syntax	**Closed**
Is duplicate of Ruby master - Feature #12125: Proposal: Shorthand operator fo...	**Closed**

History

#1 - 05/19/2017 01:37 PM - Hanmac (Hans Mackowiak)

that might collide with -> {} a lambda syntax
so i think the chances are low that ruby gets something like that.

so ruby probably does think its "Math(->sqrt)" and thats a Syntax error.

#2 - 05/19/2017 02:16 PM - americodls (Americo Duarte)

The **->** was just a suggestion... Could be another symbol or combination of symbols like **Math->>sqrt**, **Math=>>sqrt**, **Math+>sqrt**, **Math$>sqrt**, **Math:>sqrt**, etc
I just think could have another way to write it than not a method calling with a symbol as argument, something more concise and expressive.

Hanmac (Hans Mackowiak) wrote:

> that might collide with -> {} a lambda syntax
> so i think the chances are low that ruby gets something like that.
>
> so ruby probably does think its "Math(->sqrt)" and thats a Syntax error.

#3 - 05/19/2017 02:43 PM - Hanmac (Hans Mackowiak)

my current thinking is if that short form should do symbol support.

if Math->sym should be supported, than the normal variant need to be Math->:sqrt

i currently think if we also could abuse the call method.

so we could harness "Math.(:sqrt)" into something.

#4 - 05/20/2017 12:27 AM - americodls (Americo Duarte)

Why the version with symbol (Math->:sqrt) needs to be supported?

#5 - 08/31/2017 06:31 AM - matz (Yukihiro Matsumoto)

I am for adding syntax sugar for method reference. But I don't like proposed syntax (e.g. ->).
Any other idea?

Matz.

#6 - 08/31/2017 08:48 AM - nobu (Nobuyoshi Nakada)

obj\.method

#7 - 08/31/2017 10:10 AM - Hanmac (Hans Mackowiak)

nobu (Nobuyoshi Nakada) wrote:

> obj\.method

i am not sure about that:

```
obj\
.method
```

is already valid ruby code, so i am not sure

PS: when using "&obj.method(:symbol)" should that be optimized if able?

#8 - 08/31/2017 12:10 PM - zverok (Victor Shepelev)

> I am for adding syntax sugar for method reference. But I don't like proposed syntax (e.g. ->).
> Any other idea?

In my pet projects, I often alias method as m. It is readable enough, short enough and easy to remember, once you've seen it:

```
roots = [1, 4, 9].map(&Math.m(:sqrt))
%w[foo bar baz].each(&m(:puts))
```

..., and, if introduced into language core, can be easily backported to earlier versions (through something like backports or polyfill gem).

Another weird-ish idea, following the first one closely, is .:, which (for me) looks guessable:

```
[1,2,3].map(&Math.:sqrt)
%w[foo bar baz].each(&.:puts)
```

(BTW, object-less form should also be considered, when weighing proposals, don't you think?)

#9 - 08/31/2017 12:30 PM - k0kubun (Takashi Kokubun)

Another idea: &obj:method

It just puts receiver between & and : from existing one. I'm not sure it conflicts with existing syntax or not but I feel it's consistent with &:foo syntax.

```
roots = [1, 4, 9].map(&Math:sqrt)
%w[foo bar baz].each(&self:puts)
```

#10 - 08/31/2017 12:56 PM - Hanmac (Hans Mackowiak)

k0kubun (Takashi Kokubun) wrote:

> Another idea: &obj:method

hm i like that idea, but think that might be a bit conflicting, that depends on if obj is an object or not?

```
obj = Object.new
obj:method #=> syntax error
```

but notice that: (xyz not known)

```
xyz:method #=> undefined method 'xyz' for main
```

it thinks that it's xyz(:method)

#11 - 08/31/2017 01:12 PM - k0kubun (Takashi Kokubun)

Oh, I'm so sad to hear obj:method (without &) is already valid. I still have hope to have it only when it's put with & in the last of arguments because that case is not valid for now.

```
irb(main):001:0> def obj(method); method; end
=> :obj
irb(main):002:0> obj:method
=> :method
irb(main):003:0> obj(&obj:method)
SyntaxError: (irb):3: syntax error, unexpected tSYMBEG, expecting keyword_do or '{' or '('
obj(&obj:method)
         ^
        from /home/k0kubun/.rbenv/versions/2.4.1/bin/irb:11:in `<main>'
irb(main):004:0> obj(&(obj:method))
SyntaxError: (irb):4: syntax error, unexpected tLABEL
obj(&(obj:method))
          ^
        from /home/k0kubun/.rbenv/versions/2.4.1/bin/irb:11:in `<main>'
irb(main):005:0> obj &obj:method
SyntaxError: (irb):5: syntax error, unexpected tSYMBEG, expecting keyword_do or '{' or '('
obj &obj:method
         ^
        from /home/k0kubun/.rbenv/versions/2.4.1/bin/irb:11:in `<main>'
irb(main):006:0> obj &(obj:method)
SyntaxError: (irb):6: syntax error, unexpected tLABEL
obj &(obj:method)
          ^
        from /home/k0kubun/.rbenv/versions/2.4.1/bin/irb:11:in `<main>'
```

I've never seen a person who writes a(:b) as a:b (without space or parenthesis before ":") and personally I don't expect &a:b to be &a(:b).

#12 - 08/31/2017 02:27 PM - nobu (Nobuyoshi Nakada)

Hanmac (Hans Mackowiak) wrote:

> i am not sure about that:
>
> ```
> obj\
> .method
> ```
>
> is already valid ruby code, so i am not sure

It's different at all.
My example is a token \., do not split.

> PS: when using "&obj.method(:symbol)" should that be optimized if able?

Probably, but it's not possible to guarantee that it will return the method object.

#13 - 08/31/2017 02:31 PM - nobu (Nobuyoshi Nakada)

k0kubun (Takashi Kokubun) wrote:

> Another idea: &obj:method

Consider more complex example, &(obj.some.method(args)):method, not only a simple reciever.
& and : are separated too far.

#14 - 08/31/2017 02:51 PM - Hanmac (Hans Mackowiak)

my idea for optimising &obj.method(:symbol)
is that it already creates a proc (object) without going over a Method object,
i don't know if that would be an good idea for that.

#15 - 09/01/2017 12:56 AM - mrkn (Kenta Murata)

How about obj.{method_name} for the syntax sugar of obj.method(:method_name)?

#16 - 09/01/2017 06:15 AM - zverok (Victor Shepelev)

Another pretty unholy idea: resemble the way Ruby docs document the methods:

```
[1,2,3].map(&Math#sqrt)
%w[foo bar baz].each(&#puts)
```

Yes, it conflicts with comment syntax, but in fact, no sane person should join the comment sign immediately after non-space symbol.

And we already have parsing ambiguities like this:

- foo -bar → foo(-bar);
- foo - bar → foo.-(bar).

#17 - 09/01/2017 06:36 AM - tom_dalling (Tom Dalling)

What about triple colon :::?

```
[1,2,3].map(&Math:::sqrt)
[1,2,3].each(&:::puts)
```

:: is for looking up constants, so it kind of makes sense that ::: is for looking up methods.

#18 - 09/01/2017 08:30 AM - tom-lord (Tom Lord)

Consider the following:

```
def get_method(sym, object = Object)
  object.send(:method, sym)
end
```

This allows us to write code like:

```
[1, 4, 9].map &get_method(:sqrt, Math)
```

So as an idea, how about introducing some syntax sugar for the above - such as:

```
[1, 4, 9].map &~>(:sqrt, Math)
```

Where in general, ~> (or whatever) could be the "method lookup operator".

For example, this would yield:

```
~>(:puts)       # => #<Method: Class(Kernel)#puts>
~>(:sqrt, Math) # => #<Method: Math.sqrt>
~>(:sqrt)       # => NameError: undefined method `sqrt' for class `#<Class:Object>'
```

#19 - 09/29/2017 12:19 AM - k0kubun (Takashi Kokubun)

- Is duplicate of Feature #12125: Proposal: Shorthand operator for Object#method added

#20 - 10/19/2017 01:46 AM - americodls (Americo Duarte)

matz (Yukihiro Matsumoto) wrote:

> I am for adding syntax sugar for method reference. But I don't like proposed syntax (e.g. ->).
> Any other idea?
>
> Matz.

What do you think about: Kernel:puts, Kernel~>puts, Kernel:>puts ?

#21 - 01/24/2018 09:49 AM - zverok (Victor Shepelev)

Just to push this forward, here are all the syntaxes from this and duplicate #12125.
I am taking Math.sqrt and puts as examples:

- map(&Math->sqrt) (and just each(&->puts) probably?) -- Matz is explicitly against it;
- map(&Math\.sqrt) (not sure about puts);
- map(&Math.m(:sqrt)), each(&m(:puts)) (just shortening, no language syntax change)
- map(&Math.:sqrt), each(&.:puts)
- map(&Math:sqrt), each(&self:puts)
- map(&Math#sqrt), each(&#puts) (it was my proposal, "just how it looks in docs", but I should reconsider: in docs it is Math::sqrt, in fact)
- map(&Math:::sqrt), each(&:::puts)
- map(&~>(:sqrt, Math)), each(&~>(:puts))
- several by Papierkorb (Stefan Merettig):
  - map(&Math.>sqrt), each(&.>puts) (nobu (Nobuyoshi Nakada): conflicts with existing syntax)
  - map(&Math<sqrt>), each(&<puts>) (nobu (Nobuyoshi Nakada): conflicts with existing syntax)

- map(&Math&>sqrt), each(&&>puts)
- map(&Math|>sqrt), each(&|>puts) (too confusable with Elixir-like pipe, probably)

Can please please somebody lift this question to next Developer Meeting and make an Executive Decision?..

I personally really like .: (called "tetris operator" in other ticket).

#22 - 01/24/2018 02:48 PM - dsferreira (Daniel Ferreira)

zverok (Victor Shepelev) wrote:

> map(&Math|>sqrt), each(&|>puts) (too confusable with Elixir-like pipe, probably)

I tend to agree with that.
In fact I was hoping to get the pipe operator introduced in ruby. (Created an issue with that in mind: https://bugs.ruby-lang.org/issues/14392).

Taking that pipe operator example as an operator that sends messages maybe we can use it to send a message to the class or module. Like this:

```
:sqrt |> Math
```

Since Math is not a method the message would extract the method from the object.

If this is not practical maybe we could invert the operator and do:

```
Math <| :sqrt
```

#23 - 01/25/2018 12:23 PM - nobu (Nobuyoshi Nakada)

zverok (Victor Shepelev) wrote:

- map(&Math.>sqrt), each(&.>puts)

This conflicts with existing syntax.

- map(&Math&>sqrt), each(&&>puts) (nobu (Nobuyoshi Nakada): conflicts with existing syntax)

Not this.

#24 - 01/25/2018 12:31 PM - zverok (Victor Shepelev)

nobu (Nobuyoshi Nakada) Thanks, I've updated the list.

Can you please add it to next Developer Meeting's agenda?..

#25 - 02/01/2018 10:05 PM - mpapis (Michal Papis)

Not sure it's worth it - but while we are at this thinking of a shorthand one of the proposals &Math&>sqrt made me think if it could be automated and all the iterators could recognize methods and we could avoid the initial & to this map(Math...) - skipped the operator as it's not clear what's preferred.

My two cents to the operator - what about ! and @ would they conflict? (yes for the Kernel, not sure about Class'es).

#26 - 02/01/2018 11:13 PM - nobu (Nobuyoshi Nakada)

mpapis (Michal Papis) wrote:

> Not sure it's worth it - but while we are at this thinking of a shorthand one of the proposals &Math&>sqrt made me think if it could be automated and all the iterators could recognize methods and we could avoid the initial & to this map(Math...) - skipped the operator as it's not clear what's preferred.

It can't distinguish passing block and passing Method object.

> My two cents to the operator - what about ! and @ would they conflict? (yes for the Kernel, not sure about Class'es).

Math!sqrt and Math@sqrt?
They are valid syntax now.

You can try with ruby -c.

```
$ ruby -wc -e 'Math!sqrt'
Syntax OK
```

```
$ ruby -wc -e 'Math@sqrt'
Syntax OK
```

They are interpreted as a method call without a receiver, Math(!sqrt) and Math(@sqrt) respectively.

#27 - 02/02/2018 01:51 AM - duerst (Martin Dürst)

nobu (Nobuyoshi Nakada) wrote:

> ```
> $ ruby -wc -e 'Math!sqrt'
> Syntax OK
>
> $ ruby -wc -e 'Math@sqrt'
> Syntax OK
> ```
>
> They are interpreted as a method call without a receiver, Math(!sqrt) and Math(@sqrt) respectively.

This may be just me, but I think this kind of syntax without spaces could (or even should) be depreciated.

That doesn't mean that for the purpose of this issue, I like ! or @. But they might be usable for other purposes.

#28 - 02/02/2018 05:33 AM - nobu (Nobuyoshi Nakada)

duerst (Martin Dürst) wrote:

> This may be just me, but I think this kind of syntax without spaces could (or even should) be depreciated.

It would hurt code-golfers and quine-makers. :)

#29 - 02/02/2018 08:20 AM - Hanmac (Hans Mackowiak)

Question for nobu (Nobuyoshi Nakada) :

i don't know about the rubyVM but can xyz(&method(:symbol)) or xyz(&obj.method(:symbol)) be optimized like xyz(&:symbol) is?
the one with the Symbol was optimized to not create a Proc object if not needed.

can be something similar with the Method object? or if not overwritten maybe not even creating a Method object at all?

#30 - 02/02/2018 08:27 AM - nobu (Nobuyoshi Nakada)

Hanmac (Hans Mackowiak) wrote:

> i don't know about the rubyVM but can xyz(&method(:symbol)) or xyz(&obj.method(:symbol)) be optimized like xyz(&:symbol) is?

They have different meanings all.

xyz(&method(:symbol)) == xyz {|x| symbol(x)}
xyz(&obj.method(:symbol)) == xyz {|x| obj.symbol(x)}
xyz(&:symbol) == xyz {|x| x.symbol}

#31 - 02/02/2018 08:37 AM - Hanmac (Hans Mackowiak)

nobu (Nobuyoshi Nakada) wrote:

> Hanmac (Hans Mackowiak) wrote:
>
> > i don't know about the rubyVM but can xyz(&method(:symbol)) or xyz(&obj.method(:symbol)) be optimized like xyz(&:symbol) is?
>
> They have different meanings all.
>
> xyz(&method(:symbol)) == xyz {|x| symbol(x)}
> xyz(&obj.method(:symbol)) == xyz {|x| obj.symbol(x)}
> xyz(&:symbol) == xyz {|x| x.symbol}

i know they are different meanings,
i was just wondering if they can be optimized for the VM too, to make them run faster if able
like with not creating extra ruby objects if not needed

#32 - 02/02/2018 10:57 AM - nobu (Nobuyoshi Nakada)

Hanmac (Hans Mackowiak) wrote:

> i know they are different meanings,

Sorry, misread.

> i was just wondering if they can be optimized for the VM too, to make them run faster if able
> like with not creating extra ruby objects if not needed

Once a Method as the result of & to passing a block is allowed, optimization of calling a Method object might be possible by adding a new block handler type for methods.
No "extra ruby objects" would be the next step.

#33 - 02/04/2018 08:10 PM - landongrindheim (Landon Grindheim)

- map(&Math->sqrt) (and just each(&->puts) probably?) -- Matz is explicitly against it;

Is map(&Math.&(:sqrt) viable? Perhaps it would be confused with the safe navigation operator.

#34 - 02/04/2018 09:12 PM - jeremyevans0 (Jeremy Evans)

landongrindheim (Landon Grindheim) wrote:

> Is map(&Math.&(:sqrt) viable? Perhaps it would be confused with the safe navigation operator.

No. It would break backward compatibility, as that is currently interpreted as:

```
Math.&(:sqrt).to_proc
```

That code currently works if you do:

```
def Math.&(x) proc{|a| a} end
```

#35 - 02/05/2018 05:39 PM - sevos (Artur Roszczyk)

Have we ruled out map(&obj:method) syntax? Intuitively I find it consistent with Symbol#to_proc

```
class Foo
  def initialize(array)
    @array = array
  end

  def call
    @array
      .map(&Math:sqrt)
      .map(&self:magic)
      .map(&self:boo(2.0))
      .map(&:ceil)
      .each(&Kernel:puts)
  end

  private

  def magic(x)
    x ** 3
  end

  def boo(a, b)
    a / b
  end
end
```

Alternatively, I am for triple colon:

```
class Foo
  def initialize(array)
    @array = array
  end
```

```
  def call
    @array
      .map(&Math:::sqrt)
      .map(&self:::magic)
      .map(&:::boo(2.0)) # with triple colon we could omit self
      .map(&:ceil)
      .each(&Kernel:::puts)
  end

  private

  def magic(x)
    x ** 3
  end

  def boo(a, b)
    a / b
  end
end
```

This could translate to:

```
class Foo
  def initialize(array)
    @array = array
  end

  def call
    @array
      .map  { |x| Math.public_send(:sqrt, x) }
      .map  { |x| self.send(:magic, x) }
      .map  { |x| self.send(:boo, x, 2.0) }
      .map  { |x| x.ceil }
      .each { |x| Kernel.public_send(:puts, x) }
  end

  private

  def magic(x)
    x ** 3
  end

  def boo(a, b)
    a / b
  end
end
```

Applying additional arguments (aka .map(&self:boo(2.0)) is just a proposal - I am not sure if this should be even possible - Symbol#to_proc does not allow that.

Another interesting question which we need to answer is:

What visibility scope should be used when making a method call?

Given the syntax receiver:method or receiver:::method if the receiver is self then we should expand this syntax sugar to send() allowing accessing the private interface of the current object (which is not the item from the iterator - we would use symbol to proc in that case). However, if the receiver is something else, we should expand to public_send to disallow accessing private methods of other objects.

Just my two cents ;)

Cheers,
Artur

#36 - 02/06/2018 06:14 AM - nobu (Nobuyoshi Nakada)

Note that &: isn't a single operator, but combination of & prefix + a part of :symbol.
So it should be valid syntax solely without &.

#37 - 02/06/2018 08:19 AM - sevos (Artur Roszczyk)

After a while I am becoming a bigger fan of the triple colon operator. We could implement a class MethodSelector for handling the logic and the operator would be expected to return an instance of the class:

```
class MethodSelector
  def initialize(b, receiver, m)
```

```
    @binding = b
    @receiver = receiver
    @method = m
  end

  def call(*args, **kwargs, &block)
    # ...
  end

  def to_proc
    if @binding.eval("self") == @receiver
      proc do |*args, **kwargs, &block|
        if kwargs.empty?
          @receiver.send(@method, *args, &block)
        else
          @receiver.send(@method, *args, **kwargs, &block)
        end
      end
    else
      proc do |*args, **kwargs, &block|
        if kwargs.empty?
          @receiver.public_send(@method, *args, &block)
        else
          @receiver.public_send(@method, *args, **kwargs, &block)
        end
      end
    end
  end
end

# Instead of MS() method we should implement ::: operator (taking two argiments):
# receiver:::method expands to MS(binding, receiver, method)
class Object
  def MS(b, receiver, m)
    MethodSelector.new(b, receiver, m)
  end
end

# Example usage
> MS(binding, Kernel, :puts) # the triple colon operator should expand current binding by default
=> #<MethodSelector:0x007fdba89bd0a8 @binding=#<Binding:0x007fdba89bd0d0>, @receiver=Kernel, @method=:puts>
> [1,2,3].each(&MS(binding, Kernel, :puts))
1
2
3
=> nil
```

There is still the question how to enable meta-programming with triple colon operator.
Imagine the situation when the method name is dynamic. How to distinguish it from the symbol?

```
method = :puts

Kernel:::puts
Kernel:::method
```

The only logical solution to me is the presence of the fourth colon for the symbol:

```
method = :puts

Kernel::::puts # evaluates as Kernel:::(:puts)
Kernel:::method # evaluates as Kernel:::(method)
```

What are your thoughts?

#38 - 02/06/2018 08:39 AM - phluid61 (Matthew Kerwin)

sevos (Artur Roszczyk) wrote:

> What are your thoughts?

I have two:

1. As always: do we really need more magic symbols? I like reading Ruby because it's *not* Perl.

2. If you're adding new syntax, you don't have to be clever. Symbol has :"#{x}" so why not propose y:::"#{x}"? Not that it adds much over y.method(x)

#39 - 02/06/2018 10:52 AM - sevos (Artur Roszczyk)

phluid61 (Matthew Kerwin) wrote:

> sevos (Artur Roszczyk) wrote:
>
> > What are your thoughts?
>
> I have two:
>
> 1. As always: do we really need more magic symbols? I like reading Ruby because it's *not* Perl.
>   I totally agree, but we still like -> {} syntax for lambdas, right? Let's play with ideas, maybe we can find something nice for a method selector, too ;)
>
> 2. If you're adding new syntax, you don't have to be clever. Symbol has :"#{x}" so why not propose y:::"#{x}"? Not that it adds much over y.method(x)

You're totally right! I was looking at triple-colon as an operator taking two arguments. Your idea of looking at this as a double-colon lookup operator is actually great, look:

```
irb(main):006:0> a :: :to_s
SyntaxError: (irb):6: syntax error, unexpected tSYMBEG, expecting '('
a :: :to_s
      ^
    from /Users/sevos/.rbenv/versions/2.4.0/bin/irb:11:in `<main>'
irb(main):007:0> Kernel :: :t
SyntaxError: (irb):7: syntax error, unexpected tSYMBEG, expecting tCONSTANT
A :: :t
      ^
    from /Users/sevos/.rbenv/versions/2.4.0/bin/irb:11:in `<main>'
```

We already have a lookup operator which takes object, constant on the left side and method name or constand on the right side.
Maybe it would be possible to support symbols on the right side and expand them to method(:symbol) call?
I would like just to emphasise again the need of respecting the method-to-be-called visibility depending on the current binding.

#40 - 02/06/2018 01:15 PM - phluid61 (Matthew Kerwin)

sevos (Artur Roszczyk) wrote:

> phluid61 (Matthew Kerwin) wrote:
>
> > sevos (Artur Roszczyk) wrote:
> >
> > > What are your thoughts?
> >
> > I have two:
> >
> > 1. As always: do we really need more magic symbols? I like reading Ruby because it's *not* Perl.
>
> I totally agree, but we still like -> {} syntax for lambdas, right? Let's play with ideas, maybe we can find something nice for a method selector, too ;)

Personally I hate it, and never use it. I like my code to say lambda when I make a Lambda, and (more often) proc when I make a Proc.

> > 1. If you're adding new syntax, you don't have to be clever. Symbol has :"#{x}" so why not propose y:::"#{x}"? Not that it adds much over y.method(x)
>
> You're totally right! I was looking at triple-colon as an operator taking two arguments. Your idea of looking at this as a double-colon lookup operator is actually great, [...]

Actually I was just suggesting a simple token, like obj:::foo or obj.:foo; if you really want to accept variable method names why not obj:::"#{x}" or obj.:"#{x}"?

Although I doubt I'd ever use it.

(Personally I find the idea of partially applied methods more useful.)

Cheers

#41 - 02/12/2018 11:54 PM - cben (Beni Cherniavsky-Paskin)

A non-syntax idea: could Math.method.sqrt look significantly nicer than Math.method(:sqrt)?
That is, .method without args would return a magic object that for any message returns the bound method of that name.

```
[1, 4, 9].map(&Math.method.sqrt).each(&method.puts)
[1, 4, 9].map(&Math.method(:sqrt)).each(&method(:puts))
[1, 4, 9].map{|*a| Math.sqrt(*a)}.each{|*a| puts(*a)}
```

Naive implementation (some names don't work, eg. Math.method.method_missing, and doesn't take visibility and refinements into account):

```
class Methods < BasicObject
  def initialize(obj)
    @obj = obj
  end
  def method_missing(name)
    @obj.method(name)
  end
  def responds_to_missing?(name)
    true
  end
end

module MethodWithoutArgs
  def method(*args)
    if args.empty?
      Methods.new(self)
    else
      super
    end
  end
end
Object.prepend(MethodWithoutArgs)

[14] pry(main)> [1, 4, 9].map(&Math.method.sqrt).each(&method.puts)
1.0
2.0
3.0
=> [1.0, 2.0, 3.0]
```

BTW, what about refinements? Is .method(:foo) ignorant about them? A benefit of a real syntax might be that it could "see" methods from lexically active refinements.

As for syntax, I'm wondering if something *postfix* might work. The reason I say this is I'm thinking of both &: and this as shorthands for writing out a block.
&: can be read locally, it roughly "stands for" |x| x. :

```
[1, 2, 3].map{|x| x.to_s}
[1, 2, 3].map(&:to_s)
```

And with a bound method, we want to elide the argument declaration, plus the call that comes *after* the receiver.message:

```
[1, 4, 9].map{|*a| Math.sqrt(*a)}.each{|*a| puts(*a)}
[1, 4, 9].map(&Math.sqrt:).each(&puts:)  # half baked idea
[1, 4, 9].map(Math.sqrt&).each(puts&)    # quarter baked
```

OK, actually there is a more generic feature I'd love much more than a syntax for bound methods: implicit notation for block arg:

```
[1, 2, 3].map{|x| x.to_s}
[1, 2, 3].map{_.to_s}

[1, 4, 9].map{|x| Math.sqrt(x)}.each{|x| puts(x)}
[1, 4, 9].map{Math.sqrt(_)}.each{puts(_)}

[1, 2, 3].map{|x| 1/x}
[1, 2, 3].map{1/_}
```

(I don't think _ is possible, just an example)

The part I love most about this is that {} does *not* become (&...)!
This doesn't easily handle multiple args, like bound methods do, but I think one arg is sweet spot for such shorthand anyway.

- I've tried prototyping this once by defining Kernel._ that would look in caller frame, but didn't find any way to access arg in a block that didn't declare any |args|.

#42 - 02/14/2018 12:52 AM - sevos (Artur Roszczyk)

cben (Beni Cherniavsky-Paskin) wrote:

> A non-syntax idea: could Math.method.sqrt look significantly nicer than Math.method(:sqrt)?
> That is, .method without args would return a magic object that for any message returns the bound method of that name.

Hey Beni! Thank you! This is a great idea! For my taste it looks significantly better!

Also, I took the liberty of implementing a prototype gem and I've added my two cents:

- method visibility check
- arguments currying

You can check it out on Github

#43 - 03/26/2018 09:43 PM - pvande (Pieter van de Bruggen)

As a blue sky alternative, why not consider something like this:

```
[1, 2, 3].map(&> {Math.sqrt})

# or perhaps more simply

[1, 2, 3].map(&> Math.sqrt)
```

Pros:

- It's clean
- It's readable
- It reads like passing a block
  - Specifically, passing a lambda as a block
- It *also* reads something like the Elixir pipe operator

Cons:

- It's only *looks* like Ruby code
  - Is x.map(&> { a.b.c.d ; Math.sqrt }) valid? (I hope not.)
  - Is x.map(&> do; Math.sqrt; end) valid? (I hope not.)
  - Is x.map(&> { begin; rescue; end }) valid? (I hope not.)
  - Is x.map(&> { Math.sqrt if x }) valid? (I hope not.)
  - Is x.map(&> { Math.sqrt rescue nil }) valid? (I hope not.)
- It's not actually a shorthand for Object#method.
  - The two clearest implementation paths are as a "macro", and as an "atypical evaluation context"
  - The macro approach simply transforms the code into a "proper" block, and passes the argument implicitly (see below for an example)
  - The other approach requires the interpreter to all non-terminal method calls, then produce a block invoking the terminal call with the yielded argument (see below for an example)

Despite the "oddness" of this particular syntax, I think the clarity of expression is very much inline with the Ruby ideals, and is therefore worth discussing.

Macro Example

```
fn(&> { a.b.c.d })
# => fn() { |__x| a.b.c.d(__x) })
```

Atypical Evaluation Example

```
fn(&> { a.b.c.d })
# => __target = a.b.c; fn() { |__x| __target.d(__x) })
```

#44 - 04/23/2018 07:00 AM - baweaver (Brandon Weaver)

sevos (Artur Roszczyk) wrote:

> After a while I am becoming a bigger fan of the triple colon operator. We could implement a class MethodSelector for handling the logic and the operator would be expected to return an instance of the class:

```
class MethodSelector
  def initialize(b, receiver, m)
    @binding = b
    @receiver = receiver
    @method = m
  end

  def call(*args, **kwargs, &block)
    # ...
  end

  def to_proc
    if @binding.eval("self") == @receiver
      proc do |*args, **kwargs, &block|
        if kwargs.empty?
          @receiver.send(@method, *args, &block)
        else
          @receiver.send(@method, *args, **kwargs, &block)
        end
      end
    else
      proc do |*args, **kwargs, &block|
        if kwargs.empty?
          @receiver.public_send(@method, *args, &block)
        else
          @receiver.public_send(@method, *args, **kwargs, &block)
        end
      end
    end
  end
end

# Instead of MS() method we should implement ::: operator (taking two argiments):
# receiver:::method expands to MS(binding, receiver, method)
class Object
  def MS(b, receiver, m)
    MethodSelector.new(b, receiver, m)
  end
end

# Example usage
> MS(binding, Kernel, :puts) # the triple colon operator should expand current binding by default
=>
#<MethodSelector:0x007fdba89bd0a8 @binding=#<Binding:0x007fdba89bd0d0>, @receiver=Kernel, @method=:puts>
> [1,2,3].each(&MS(binding, Kernel, :puts))
1
2
3
=> nil
```

There is still the question how to enable meta-programming with triple colon operator.
Imagine the situation when the method name is dynamic. How to distinguish it from the symbol?

```
method = :puts

Kernel:::puts
Kernel:::method
```

The only logical solution to me is the presence of the fourth colon for the symbol:

```
method = :puts

Kernel::::puts # evaluates as Kernel:::(:puts)
Kernel:::method # evaluates as Kernel:::(method)
```

What are your thoughts?

I like the idea of triple-colon as well for succinctness.

Most of the alternative in terms of succinctness would involve discussing the no-parens syntax which is likely a non-starter for obvious compatibility reasons.

That is, unless there's a way to stop paren-free method calling in the presence of an & or to_proc:

```
[1,2,3].map(&Math.sqrt)
Math.sqrt.to_proc
```

...but that feels like an excess of black magic in the parser and would likely be prone to bugs.

I really do like what Scala does with underscores:

```
[1,2,3].map(_ * 10)
```

...but I also understand that that would also be hugely breaking in terms of syntax as well.

Really though I think given what Ruby already does the triple-colon is the cleanest route for now.

#45 - 05/17/2018 06:45 AM - matz (Yukihiro Matsumoto)

Out of ruby-core:85038 candidates, .: looks best to me (followed by :::).
Let me consider it for a while.

Matz.

#46 - 05/23/2018 09:47 AM - cben (Beni Cherniavsky-Paskin)

Matz, could you give your thoughts on obj::method (with lowercase on right side) syntax?
AFAICT it's synonym to obj.method?
Does anybody use :: for method calls in practice?

I understand breaking compatibility can't be justified here, but I'm just curious — do you see this syntax as a feature to be preserved, or something to be avoided that might be (slowly) deprecated one day?

#47 - 11/10/2018 05:01 AM - ianks (Ian Ker-Seymer)

matz (Yukihiro Matsumoto) wrote:

> Out of ruby-core:85038 candidates, .: looks best to me (followed by :::).
> Let me consider it for a while.
>
> Matz.

Would love to see see either one implemented at this point. Not having an ergonomic way for functional composition is a pain.

My dream:

```
slug = title
  .then(&:strip)
  .then(&:downcase)
  .then(I18n:::transliterate)
  .then(Utils:::hyphenate)
  .then(Validations:::check_length)
  .tap(PostLogger:::info)
```

#48 - 11/12/2018 07:20 AM - nobu (Nobuyoshi Nakada)

https://github.com/nobu/ruby/tree/feature/13581-methref_op

#49 - 11/12/2018 10:29 AM - zverok (Victor Shepelev)

nobu (Nobuyoshi Nakada) Awesome!

Am I correct that receiver-less call, like something.map(&.:puts), will be impossible?
Is it a voluntary design decision, or limitation of what can be parsed?

#50 - 11/14/2018 02:46 PM - shevegen (Robert A. Heiler)

I think .: is better than ::: but it is not very pretty either. I have
no better suggestion, though. Good syntax is not easy to use. :(

I agree with the functionality by the way.

#51 - 11/15/2018 01:50 AM - nobu (Nobuyoshi Nakada)

zverok (Victor Shepelev) wrote:

> Am I correct that receiver-less call, like something.map(&.:puts), will be impossible?

To allow that, .:puts should be a sole expression by itself.
However ruby has the line continuation for “fluent interface” (like https://bugs.ruby-lang.org/issues/13581#change-74822), for a decade.
If .: will be introduced, I think it should obey that syntax too, and allowing it without the receiver feels confusing.

> Is it a voluntary design decision, or limitation of what can be parsed?

It is easy to add a receiver-less syntax.
https://github.com/ruby/ruby/commit/2307713962c3610f4e034e328af37b19be5c7c45

#52 - 11/15/2018 08:42 AM - zverok (Victor Shepelev)

nobu (Nobuyoshi Nakada)

> If .: will be introduced, I think it should obey that syntax too, and allowing it without the receiver feels confusing.

Can you please show some example of confusing statements? I can't think of any from the top of my head, it seems that (if the parser can handle it), the context for .:something and .something is always clearly different.

I am concerned about receiver-less version because in our current codebase we found this idiom to be particularly useful:

```
# in a large data-processing class
some_input
  .compact
  .map(&method(:process_item)) # it is private method of current class
  .reject(&method(:spoiled?))
  .tap(&method(:pp)) # temp debugging statement
  .group_by(&method(:grouping_criterion))
  .yield_self(&method(:postprocess))

# which I'd be really happy to see as
some_input
  .compact
  .map(&.:process_item)
  .reject(&.:spoiled?)
  .tap(&.:pp)
  .group_by(&.:grouping_criterion)
  .then(&.:postprocess)
```

Having to explicitly state map(&self.:process_item) is much less desirable.

#53 - 11/15/2018 10:02 AM - AlexWayfer (Alexander Popov)

zverok (Victor Shepelev) wrote:

> nobu (Nobuyoshi Nakada)
>
> > If .: will be introduced, I think it should obey that syntax too, and allowing it without the receiver feels confusing.
>
> Can you please show some example of confusing statements? I can't think of any from the top of my head, it seems that (if the parser can handle it), the context for .:something and .something is always clearly different.
>
> I am concerned about receiver-less version because in our current codebase we found this idiom to be particularly useful:
>
> ```
> # in a large data-processing class
> some_input
>   .compact
>   .map(&method(:process_item)) # it is private method of current class
>   .reject(&method(:spoiled?))
>   .tap(&method(:pp)) # temp debugging statement
>   .group_by(&method(:grouping_criterion))
>   .yield_self(&method(:postprocess))
>
> # which I'd be really happy to see as
> some_input
>   .compact
>   .map(&.:process_item)
>   .reject(&.:spoiled?)
>   .tap(&.:pp)
>   .group_by(&.:grouping_criterion)
>   .then(&.:postprocess)
> ```

> Having to explicitly state map(&self.:process_item) is much less desirable.

Just an opinion:

```
processed =
  some_input
    .compact
    .map { |element| ProcessingItem.new(element) } # or `.map(&ProcessingItem.method(:new))`
    .reject(&:spoiled?)
    .each(&:pp) # temp debugging statement
    .group_by(&:grouping_criterion)

postprocess processed
```

Or you can even use ProcessingItems collection class. With itself state and behavior. Instead of bunch of private methods in a processing class with the same (collection) argument.

#54 - 11/15/2018 10:09 AM - zverok (Victor Shepelev)

> Just an opinion

It is funny how when you show some imaginary code, quick-written just to illustrate the point of a language feature, people tend to discuss this code's design approaches instead.

Yes, obviously, in the situation like "several consecutive, algorithmically complex methods working on the same collection" it is typically wise to just wrap collection items. But that has absolutely nothing to do with the point of my example.

#55 - 11/15/2018 10:26 AM - AlexWayfer (Alexander Popov)

zverok (Victor Shepelev) wrote:

> > Just an opinion
>
> It is funny how when you show some imaginary code, quick-written just to illustrate the point of a language feature, people tend to discuss this code's design approaches instead.
>
> Yes, obviously, in the situation like "several consecutive, algorithmically complex methods working on the same collection" it is typically wise to just wrap collection items. But that has absolutely nothing to do with the point of my example.

I just try to use good (existing) sides of a language. Ruby already has nice Symbol#to_proc syntax. And yes, different "syntax sugars" allow to use different design approaches (classes vs functions, for example). But sometimes they also allow to create bad practices. I'm not sure, and I'm not against syntax sugar, but… I like solutions for real problems, not imaginary. With knowledge of solutions for imaginary problems we can create these problems later, without resolving them with other approaches.

#56 - 11/15/2018 10:58 AM - zverok (Victor Shepelev)

> I like solutions for real problems, not imaginary.

The map(&method(:local_method)) or yield_self(&method(:local_method)) pattern is absolutely real and very useful. My point was, we have plenty in our current codebase (and no, they are not "you just need to wrap collection items", my example was exaggerated for the illustrative purposes).

And I can definitely say it brings value for code design and clarity, and will be even more so with map(&.:local_method) syntax.

#57 - 11/26/2018 08:10 PM - shevegen (Robert A. Heiler)

> The map(&method(:local_method)) or yield_self(&method(:local_method)) pattern is absolutely real
> and very useful.

Everyone who suggests something tends to think of it as useful and often pretty too. :-)

I personally have become too picky perhaps. I already don't like yield_self much at all; "then"
is better than yield_self though.

In ruby it is possible to avoid a lot of things and still end up writing pretty code that is fun.

> My point was, we have plenty in our current codebase

I think a pretty syntax is great, but it is not necessarily ruby's primary design goal. It is difficult to say because I am not matz :) but I think matz has said before that ruby should be fun to use; and solve real problems; and help people. If you look at the safe navigation operator, this is a good example, in my opinion. Someone had a use case and suggested a solution to his problem, and matz agreed with the problem description and added the safe navigation operator.

There are lots of things I myself have not yet used, including the safe navigation operator. I also have not yet used the -> lambda variant. I no longer use @@variables either. I am sure other people have different means how to use and approach ruby. What I do use, and that has been somewhat newish (well not yet 10 years old I think), was the new additional hash syntax, since it has a net benefit - less to type, e. g.:

```
:foo => :bar
```

versus

```
foo: :bar
```

Especially if I have lots of entries, the second variant is indeed easier to use for me.

> And I can definitely say it brings value for code design and clarity, and will be even more so with map(&.:local_method) syntax.

Using this reasoning we can say that EVERY addition is GOOD and USEFUL because we have more features. But it is not quite like that. More and more features make a language harder to use and more complicated too. Some features also look strange. For example, I personally dislike the map(&.:local_method). I don't have an alternative suggestion that is nice to read, but it is hard for me to visually distinguish between what is going on there.

Ultimately it is up to matz how he wants to change ruby, but I really don't feel that in all the discussions the trade off was or is really worth it. This may be up to personal preferences or habits, yes - but ... I don't know.

When I look at things such as map(&.:local_method) then the original suggestion in the issue of:

```
roots = [1, 4, 9].map Math->method
```

becomes a LOT cleaner and easier to read. ;)

(Only thing that puts me off is that -> is used in e. g. LPC to invoke methods; I much prefer just the single "." notation. Do also note that I agree that we should be able to pass arguments to .map(&) but ... I don't know. It's visually not very pleasing to my eyes. :\ )

#58 - 12/09/2018 10:19 PM - shuber (Sean Huber)

> matz (Yukihiro Matsumoto) wrote:
>
> > Out of ruby-core:85038 candidates, .: looks best to me (followed by :::).
> > Let me consider it for a while.
> >
> > Matz.

matz (Yukihiro Matsumoto) and nobu (Nobuyoshi Nakada)

What do you guys think about this alternative syntax? (working proof of concept: https://github.com/LendingHome/pipe_operator)

```
-9.pipe { abs | Math.sqrt | to_i }
#=> 3

[9, 64].map(&Math.|.sqrt)
```

```
#=> [3.0, 8.0]

[9, 64].map(&Math.|.sqrt.to_i.to_s)
#=> ["3", "8"]

"https://api.github.com/repos/ruby/ruby".| do
  URI.parse
  Net::HTTP.get
  JSON.parse.fetch("stargazers_count")
  yield_self { |n| "Ruby has #{n} stars" }
  Kernel.puts
end
#=> Ruby has 15120 stars
```

There's nothing really new/special here - it's just a block of expressions like any other Ruby DSL and the pipe | operator has been around for decades!

The https://github.com/LendingHome/pipe_operator README contains many more examples and the implementation details - I would love to hear your thoughts!

Thanks,
Sean Huber

#59 - 12/10/2018 02:25 AM - nobu (Nobuyoshi Nakada)

shuber (Sean Huber) wrote:

> What do you guys think about this alternative syntax? (working proof of concept: https://github.com/LendingHome/pipe_operator)

It conflicts with existing | methods.

```
p 1.|2 #=> 3
```

#60 - 12/10/2018 04:12 AM - shuber (Sean Huber)

> It conflicts with existing | methods.
>
> ```
> p 1.|2 #=> 3
> ```

nobu (Nobuyoshi Nakada) That's just an alias for syntactic sugar - the actual method is named __pipe__! My thinking was that it's pretty similar to how we commonly use send (which can have conflicts in certain domains e.g. delivery services) but it's actually an "alias" of __send__ which is double underscored to avoid conflicts. Some objects may have their own definitions of | but we always have pipe or __pipe__ available to remove any ambiguity. Does that make sense?

```
def __pipe__(*args, &block)
  Pipe.new(self, *args, &block)
end

alias | __pipe__
alias pipe __pipe__
```

Within the context of a __pipe__ block we don't have to worry about conflicting | since we "own" that definition:

```
-9.pipe { abs | Math.sqrt | to_i } #=> 3
```

But like you stated, outside of a __pipe__ block the definition of | could conflict:

```
1.|2 #=> 3
```

One thought comes to mind though - this "pipe" behavior really only needs to trigger in two cases:

1) pipe | is called with NO arguments e.g. Math.|.sqrt
2) pipe | is called with ONLY a block e.g. `"test".| { Marshal.dump | Base64.encode64 }

Do you know of any existing implementations of | in the Ruby stdlib that either accept NO arguments or ONLY a block? I believe most if not all expect an argument to be passed? If so then that would make conflicts more rare if it's something we could build logic around!

#61 - 12/11/2018 08:28 AM - shuber (Sean Huber)

matz (Yukihiro Matsumoto) This pipe_operator syntax actually looks very similar to https://github.com/matz/streem!

```
url.pipe { URI.parse | Net::HTTP.get | JSON.parse }

"https://api.github.com/repos/ruby/ruby".pipe do
  URI.parse
  Net::HTTP.get
  JSON.parse.fetch("stargazers_count")
  then { |n| "Ruby has #{n} stars" }
  Kernel.puts
end
#=> Ruby has 15120 stars
```

https://github.com/LendingHome/pipe_operator

#62 - 12/13/2018 04:58 PM - shuber (Sean Huber)

Also discussing pipe operators in https://bugs.ruby-lang.org/issues/14392#note-26

#63 - 12/13/2018 05:51 PM - headius (Charles Nutter)

Triple : doesn't parse right now and has some synergy with constant references:

```
obj:::foo # => obj.method(:foo)
```

#64 - 12/19/2018 11:30 AM - zverok (Victor Shepelev)

From developer's meeting log:

> 2.7 or later
> knu: Introducing "…" (as in Lua) would allow for writing this way: ary.each { puts(...) }
> Matz: Allowing omission of "self" sounds like a bad idea because that makes each(&:puts) and each(&.:puts) look very similar but act so differently.

The last Matz's notice makes a lot of sense for me, so I withdraw my petition for self-less operator :)

#65 - 12/31/2018 03:00 PM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset trunk|r66667.

Method reference operator

Introduce the new operator for method reference, .:.
[Feature #12125] [Feature #13581]
[EXPERIMENTAL]

#66 - 11/20/2019 10:21 AM - znz (Kazuhiro NISHIYAMA)

- Related to Feature #16275: Revert `.:` syntax added