Ruby master - Feature #8191

Short-hand syntax for duck-typing

03/31/2013 01:13 PM - wardrop (Tom Wardrop)

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

Description

=begin
As a duck-typed language, Ruby doesn't provide any succinct way of safely calling a potentially non-existant method. I often find myself doing (({obj.respond_to? :empty ? obj.empty : nil})), or if I'm feeling lazy, (({obj.empty? rescue nil})). Surely we can provide a less repetitive way of achieving duck-typing, e.g. I don't care what object you are, but if you (the object) can't tell me whether you're empty, I'm going to assume some value, or do something else instead.

I'm not sure what the best way to implement this is. The easiest would be to just define a new conditional send method:

```
obj.send_if(:empty?, *args) { nil }

obj.try(:empty?, *args) { nil }
```

But that's really not much of an improvement; it's ugly. Preferably, it'd be nice to build it into the language given how fundamental duck-typing is to Ruby. One potential syntax is:

obj.empty? otherwise nil

The ((|otherwise|)) keyword would be like a logical or, but instead of short-circuiting on true, it short-circuits on some other condition. That condition can be one of two things. It can either wait for a NoMethodError (like an implicit (({rescue NoMethodError}))), proceeding to the next expression if one is raised, or it can do a pre-test using (({respond_to?})). Each option has its pro's and con's.

The implicit rescue allows you to include expressions, e.g.

```
obj.empty? otherwise obj.length == 0 otherwise true
```

Going with the implicit (({respond_to?})) implementation probably wouldn't allow that. You'd instead need to limit it just to method calls, which is not as useful. The only problem with implicitly rescuing NoMethodError's though, is that you'd need to ensure the NoMethodError was raised within the target object, and not some dependancy, as you could potentially swallow valid exceptions.

The benefit of this over current methods of duck-typing, is that you're not testing a condition, then running an action, you're instead doing both at the same time making it much more DRY.

One other potential syntax however is a double question mark, or question mark prefix. This could act as an implicit (({respond_to?})) pre-condition, returning nil if the method doesn't exist.

```
obj.empty??? || obj.length?? == 0 || nil

obj.?empty? || obj.?length == 0 || nil
```

I'm not completely satisfied with either syntax, so at this point I'm merely hoping to start a discussion.

Thoughts?
=end

Related issues:

Related to Ruby master - Feature #8237: Logical method chaining via inferred ...	**Closed**	
Related to Ruby master - Feature #11537: Introduce "Safe navigation operator"	**Closed**	
Is duplicate of Ruby master - Feature #1122: request for: Object#try	**Rejected**	**02/07/2009**

History

#1 - 03/31/2013 05:56 PM - duerst (Martin Dürst)

You call something like

```
obj.respond_to? :empty ? obj.empty : nil
```

"achieving duck typing", but that's not how duck typing works. If you mix all kinds of unrelated stuff in your program and then have to check whether they implement a message, you should probably think better about how to organize this stuff. On the other hand, if you have objects that might be empty in one way or another, but currently are not, then you should add an empty? (including the question mark) method to the respective class. In the extreme, that could go as far as doing:

```
class Object
  def empty()  nil;  end
end
```

So in my book, if you have lots of statements like the above in your code, you either haven't understood duck typing yet, or you are not using it well. Of course, we would have to look at actual code samples to find out what should be changed.

#2 - 04/01/2013 09:14 AM - wardrop (Tom Wardrop)

I know what duck-typing is, and it's a bit of a stretch to call this duck-typing, but it follows the duck-typing philosophy - "I don't care what you are, but if you can tell me something, I'll use that information. Otherwise I'll make an assumptions or take some other action."

There's nothing inherently wrong with "mixing all kinds of stuff". For example, the logic in a template should, if possible, not care about the type of data it's about to print out, the #to_s method handles that. However, a template may want to avoid printing something if it thinks it'll print something empty or useless. This is what the #blank? method tries to address in Rails. The following are all comparable:

```
unless obj.blank?
   "<span class='something'>#{obj}</span>"
end

unless obj.empty??? || !obj
  "<span class='something'>#{obj}</span>"
end

unless obj.empty? otherwise !obj
  "<span class='something'>#{obj}</span>"
end

unless !obj || (obj.empty? if obj.respond_to? :empty?)
  "<span class='something'>#{obj}</span>"
end
```

Such a construct as this would also allow you to deal with API inconsistency or differences as well. One set of objects may expose a #length method, while another set may use #count. They may different semantics, but as a programmer, you may not care about those semantic differences in a particular scenario, and just want to get a value.

I don't hit this ALL the time, but enough to make me want to raise this issue. Ruby shouldn't punish me for asking some unknown object for some non-critical information, which it does either by raising a NoMethodError, or making me use cumbersome pre-conditions. If an object can tell me something, I can use that information, if available, to produce a more desirable outcome.

#3 - 04/01/2013 11:14 AM - wardrop (Tom Wardrop)

The more I think about this and it's potential use cases, the more I like the inline double question-mark syntax. This would make the implementation even more useful, addressing the following scenario:

```
if user && user.profile && user.profile.website && user.profile.website.thumbnail
  # do something
end

if user.profile??.website??.thumbnail
  # do something
end
```

Could even be applied to handling potentially undefined variables, which again, is a situation you may encounter with templates and the "locals" idiom:

```
if user??
  # do something
end
```

#4 - 04/02/2013 06:19 PM - naruse (Yui NARUSE)

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

#5 - 04/02/2013 08:34 PM - phluid61 (Matthew Kerwin)

wardrop (Tom Wardrop) wrote:

```
if user && user.profile && user.profile.website && user.profile.website.thumbnail
  # do something
end

if user.profile??.website??.thumbnail
  # do something
end
```

I think you're missing some question-marks there, unless you intend nil to (potentially) define a `thumbnail' method.

```
if user.profile??.website??.thumbnail??
```

#6 - 04/02/2013 11:02 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I believe you didn't understand the proposal, Matthew. a??.b would mean (a.nil? ? nil : a.b). So user.profile??.website??.thumbnail is equivalent to:

user.profile.nil? ? nil : (user.profile.website.nil? ? nil : user.profile.website.thumbnail)

I'm +1 for this proposal by the way. CoffeeScript and Groovy already support this using a single question mark, but that is not possible in Ruby since methods may end with a question mark. I find double question marks for that feature acceptable.

#7 - 04/03/2013 05:59 AM - phluid61 (Matthew Kerwin)

On Apr 3, 2013 12:03 AM, "rosenfeld (Rodrigo Rosenfeld Rosas)" <
rr.rosas@gmail.com> wrote:

> I believe you didn't understand the proposal, Matthew. a??.b would mean
> (a.nil? ? nil : a.b). So user.profile??.website??.thumbnail is equivalent
> to:
>
> user.profile.nil? ? nil : (user.profile.website.nil? ? nil :
> user.profile.website.thumbnail)

You are correct, I thought a.b?? meant (a.respond_to? :b ? a.b : nil) since
I thought I saw earlier something like `foo.empty???´

In that case the missing question marks were earlier in the sequence:

```
if user??.profile??.website??.thumbnail
```

#8 - 04/03/2013 06:23 AM - davidderyldowney (David Deryl Downey)

I'm sitting here lurking on this thread and to me the proposal makes
everything seem convoluted. Though user??.website??.profile?? could
feasibly be worked out by a new to intermediate user as asking
progressively if user was present and if so was a website object associated
with that user and if so was a profile associated to the associated website
present. But the syntax of that semantic isn't really clear or 'natural'.
Its starting to feel like things are moving towards being overly terse
rather than clarity being a retainable goal.
On Apr 2, 2013 4:57 PM, "Matthew Kerwin" matthew@kerwin.net.au wrote:

> On Apr 3, 2013 12:03 AM, "rosenfeld (Rodrigo Rosenfeld Rosas)" <
> rr.rosas@gmail.com> wrote:
>
> > I believe you didn't understand the proposal, Matthew. a??.b would mean
> > (a.nil? ? nil : a.b). So user.profile??.website??.thumbnail is equivalent
> > to:
> >
> > user.profile.nil? ? nil : (user.profile.website.nil? ? nil :
> > user.profile.website.thumbnail)
>
> You are correct, I thought a.b?? meant (a.respond_to? :b ? a.b : nil)
> since I thought I saw earlier something like `foo.empty???´
>
> In that case the missing question marks were earlier in the sequence:
>
> ```
> if user??.profile??.website??.thumbnail
> ```

#9 - 04/03/2013 09:54 AM - wardrop (Tom Wardrop)

=begin
In my example using (({user.profile??.website??.thumbnail})), it assumes that if the (({website})) method exists, then whatever it returns will have a (({thumbnail})) method. It'd probably make more sense to add the double question marks to (({thumbnail})) as well: (({user.profile??.website??.thumbnail??})). Depends on your API of course.

davidderyldowney (David Deryl Downey), ideally, a single question mark would probably make that clearer, but alas, a single question mark serves another purpose. The question mark prefix is always an option though as well, e.g.

```
user.?profile.?website.?thumbnail
```

But that seems uglier and even more confusing to me. The other option is to go back to the implicit "rescue NoMethodError", (({missing})) is also a potential keyword candidate e.g.

```
user.profile.website.thumbnail missing nil
```

I still prefer the double question mark syntax though. Remember, you're not going to see this plastered everywhere. You'll only see it in place of uglier code such as:

```
user && user.profile && user.profile.website && user.profile.website.thumbnail

user.profile.website.thumbnail rescue nil
```

Even the above two examples are not as robust as this proposal. The first assumes that if (({user})) is truthy, it'll respond to (({profile})) and so on, while the second example assumes that the only possible exception that could be raised in a NoMethodError caused by the method chain itself, potentially suppressing all kinds of valid exceptions. The equivalent long-hand syntax for my proposed short-hand syntax is in fact:

```
if user.respond_to? :profile  && user.profile.respond_to? :website && user.profile.website.respond_to? :thumbn
ail
  user.profile.website.thumbnail
end
```

Even that syntax isn't a perfect match, as if for example the (({user})) method is process-heavy, calling it 4 times is going to be slow. So in fact, here's what you'd have to do to achieve the same results as (({user.profile??.website??.thumbnail??})):

```
if (user_obj = user).respond_to?(:profile) && (profile_obj = user_obj.profile).respond_to?(:website) && (websi
te_obj = profile_obj.website).respond_to?(:thumbnail)
  website_obj.thumbnail
end
```

My point I guess, is that there's a clear benefit to be had here.
=end

#10 - 04/03/2013 10:53 AM - duerst (Martin Dürst)

On 2013/04/03 9:54, wardrop (Tom Wardrop) wrote:

> Issue #8191 has been updated by wardrop (Tom Wardrop).
>
> davidderyldowney (David Deryl Downey), ideally, a single question mark would probably make that clearer, but alas, a single question mark serves another purpose. The question mark prefix is always an option though as well, e.g.

How does this work with methods that already have a ? at the end? Will we get something like include???, or what?

> Even that syntax isn't a perfect match, as if for example the (({user})) method is process-heavy, calling it 4 times is going to be slow.

Slow is still not the worst. If one of these methods has side effects, that would be even worse.

I think this kind of pattern appears once in a while, but I think we should be careful about introducing shortcuts like these (and wrongly calling them duck-typing), because there may often be a better way to organize the code (and use real duck-typing).

Regards, Martin.

#11 - 04/03/2013 10:55 AM - parndt (Philip Arndt)

I'm very -1 on this for the same reason Object#try (#1122) was rejected and also because I can't see how this could result in quality code at all (Law of Demeter keeps coming to mind).

#12 - 04/03/2013 10:56 AM - wardrop (Tom Wardrop)

duerst (Martin Dürst)
Correct. Methods already ending in a question mark, such as (({empty?})), will have two extra question marks appended, (({empty???}))

On your second point, very true. Side-effects would be worse.

Finally, I agree that we should sit on this for a while and discuss it through. It's funny though as since I raised this issue, I've since hit all kinds of scenarios in which this would be very useful. For example, I had a method to convert a given object to something appropriate for a log file. I would have liked to be able to do something like this...

```
obj.name?? || obj.class.name || "Anonymous class"
```

I did end up doing that, albiet with more code. It would have been another nice little win. I don't believe code organisation and best-practices would negate the practicality of such a construct.

#13 - 04/03/2013 11:01 AM - JonRowe (Jon Rowe)

Although this is just my 2¢... I think this is a bad idea... and here's why...

1) One of the principles of good software development is "tell don't ask", meaning that we should be passing messages around objects, telling others to do things for us, not checking incessantly wether we should do something based on a value. We should be giving our code objects that respond to the things we wish to use. in your example, it would be better to render different templates based on the population of the object. E.g. render an 'empty' partial or a 'full' partial.

2) There is already a tendency in Ruby based code to abuse nil and return values thus just as object.do_something rescue nil is a code smell, so will object.do_something otherwise nil become.

3) To quote someone on twitter... "Demeter says no."

It would be better to encourage our design of code to not use nil and instead tend towards things like NullObjects rather than adding another layer of conditional protections.

If frameworks built on top of Ruby wish to implement such things (such as Rails' Object#try) they are open to do so, but I strongly believe we shouldn't pollute the language with them.

#14 - 04/03/2013 11:03 AM - jamesotron (James Harton)

OP's original examples involve Rails views, where I'd have to say that a helper or presenter would reduce the view logic in the desired way whereas his solution (which merely replicates Object#try) just golfs the logic in place.

#15 - 04/03/2013 11:10 AM - wardrop (Tom Wardrop)

parndt (Philip Arndt) Law of Demeter essentially states that method chaining is bad. I personally don't see it's relevance to Ruby.

I believe issue #1122 was rejected because, A) the main proposal was to implement an additional method; something any library could implement itself, and B) the whole discussion was unconvincing; very few uses cases given, etc.

The success of such a feature can only be maximised by implementing this as a native language construct. A method prefix or suffix is the simplest to understand logically, and is the most succinct syntactically.

#16 - 04/03/2013 11:23 AM - wardrop (Tom Wardrop)

=begin
JonRowe (Jon Rowe) That's a very high-level idealistic point of view. Telling others to do things for us is all well and good, but at some point someone - the thing being told - has to actually do it. To use this example (({user.profile.website.thumbnail})), you might say "the user object should take care of fetching the thumbnail", but then that only palms the problem off to another object.

I don't think it's helpful to discuss programming patterns here. Ruby allows for all kinds of potentially nasty things, like #instance_variable_set, but the reason we all love Ruby is that it treats us like adults. It gives us the tools unconditionally, so we can decide how to best and most responsibly use them to solve our problems. You're also forgetting that Ruby is an excellent scripting language. One liners get work done. We shouldn't be making assumptions about how Ruby is used, or how it should be used.
=end

#17 - 04/03/2013 11:49 AM - JonRowe (Jon Rowe)

wardrop (Tom Wardrop) The reason we all love Ruby is because it makes us happy. This suggestion makes me unhappy, so I'm saying so, and I'm attempting to do so constructively. Ruby has an ethos around writing elegant clean code which deserves protection. I think your suggestion will just create more spaghetti code, which we should be keen to avoid.

"Law of Demeter essentially states that method chaining is bad. I personally don't see it's relevance to Ruby."

Yet obeying the Law of Demeter in your example, user.profile.website.thumbnail, would have meant you wouldn't have this problem. You are attempting to palm off the consequences of your decision to write code in this fashion to the language, rather than refactoring your code to clean it up.

"I don't think it's helpful to discuss programming patterns here."

It really is, discussing how we write code affects how we make decisions about the language. If you ignore how we actually write code, and how we should be striving to produce better code, then you are ignoring the consequences of your decisions. Requests made to change the language need to be carefully considered to see how they will affect the way we right code. They should have a net positive effect and not just add cruft for the sake of it.

Additionally this feature has already been rejected once, #1122, because:

"Matz first said he hesitated to extend the syntax for this feature. He then said there is no good reason to make this feature built-in;" so this is just a further syntax extension, and there is still no good reason to make this feature built in."

#18 - 04/03/2013 12:18 PM - wardrop (Tom Wardrop)

You've chopped off a bit of Matz quote there (actually, a quote of a quote of Matz): "He then said there is no good reason to make this feature built-in; people can use it as a gem (e.g., ActiveSupport).". As mentioned, the main proposal in that issue was the addition of a method, i.e. something identical to #try from Rails. Also, as I mentioned, that particular issue did not raise a very convincing case.

I'm curious how you would re-write or restructure your code to negate the need for method chaining cases such as (({user.profile.website.thumbnail})). It's one thing to say you shouldn't do it, it's another to actually give an example of what you should be doing. You don't even need to chain methods for this proposal to be useful though. I've already given examples of that.

If some people don't like this proposal, it'd be helpful for them to fully explain why. If you don't think the proposal is necessary, please invalidate my use cases. That way we can all learn and understand.

#19 - 04/03/2013 01:10 PM - phluid61 (Matthew Kerwin)

=begin
Throwing this out there for discussion: how about a completely different syntax?

user&&.profile&&.website&&.thumbnail

This was inspired by the original line (({user && user.profile && user.profile.website && user.profile.website.thumbnail}))

That line looks "ugly" because of the geometric growth of the line with every additional nesting, however it becomes even uglier (and can potentially cause unexpected behaviours) if any of those methods have side-effects.

I imagine the implementation of my suggested syntax to be equivalent to:

(tmp=user) && (tmp=tmp.profile) && (tmp=tmp.website) && tmp.thumbnail

Thus it is familiar (similar to existing (({&&=})) syntax), compact, and doesn't rely on multiple invocations of shallower-nested methods.

As far as I know, there is no way for (({&&})) to be legally followed by (({.})), so it shouldn't cause any ambiguities.

=end

#20 - 04/03/2013 02:47 PM - wardrop (Tom Wardrop)

=begin
I don't mind that. In fact, it gives me an idea for something even more generic, and with more potential application:

```
user && .profile && .website && .thumbnail
```

Ruby could implement this as new syntactic rule. A dot operator at the beginning of an expression, proceeded by a valid method name, would be called on the result of the last expression in the current scope. Technically, it would allow this:

```
lowercase = 'string'
uppercase = .upcase
```

That's pretty useless, but the point is that the syntax would be generic and unassuming, allowing for all kinds of interesting uses. Here's a random example. Not saying it's the best way to do this, but just demonstrating the potential:

```
catch(:pass) do
  # Do some stuff here
end

puts "The block was passed" if .nil?
```

In this case, the last expression in the scope was the return value of (({catch})). That's what #nil? ends up being called on.

Unfortunately, it doesn't completely solve the issues raised. For example, it does nothing for the use case (({!obj || obj.empty?})) which will still bomb if ((|obj|)) doesn't respond to (({empty?})). The double-question mark would still solve this issue (({!obj || obj.empty???})), but I still like the implied method target idea, but for different reasons.
=end

#21 - 04/03/2013 05:29 PM - regularfry (Alex Young)

On Wed, 2013-04-03 at 14:47 +0900, wardrop (Tom Wardrop) wrote:

> Issue #8191 has been updated by wardrop (Tom Wardrop).
>
> I don't mind that. In fact, it gives me an idea for something even more generic, and with more potential application:
>
> ```
> user && .profile && .website && .thumbnail
> ```

I don't see this mentioned upthread anywhere, but as far as I can tell this proposal is basically covered by https://github.com/raganwald/andand.git . Objections to this are objections to the Maybe monad. Given that, if we're going for new syntax, for the #nil? case I quite like the look of:

user.&&.profile.&&.website.&&.thumbnail

but making '&&' a valid method name might be a trifle difficult to get right...

--
Alex

> Ruby could implement this as new syntactic rule. A dot operator at the beginning of an expression, proceeded by a valid method name, would be called on the result of the last expression in the current scope. Technically, it would allow this:
>
> ```
> lowercase = 'string'
> uppercase = .upcase
> ```
>
> That's pretty useless, but the point is that the syntax would be generic and unassuming, allowing for all kinds of interesting uses. Here's a random example. Not saying it's the best way to do this, but just demonstrating the potential:
>
> ```
> catch(:pass) do
>   # Do some stuff here
> end
>
> puts "The block was passed" if .nil?
> ```
>
> In this case, the last expression in the scope was the return value of (({catch})). That's what #nil? ends up being called on.

Unfortunately, it doesn't completely solve the issues raised. For example, it does nothing for the use case ({!obj || obj.empty?})) which will still bomb if ((|obj|)) doesn't respond to (({empty?})}. The double-question mark would still solve this issue (({!obj || obj.empty???})), but I still like the implied method target idea, but for different reasons.

Feature #8191: Short-hand syntax for duck-typing
https://bugs.ruby-lang.org/issues/8191#change-38152

Author: wardrop (Tom Wardrop)
Status: Assigned
Priority: Normal
Assignee: matz (Yukihiro Matsumoto)
Category:
Target version:

=begin
As a duck-typed language, Ruby doesn't provide any succinct way of safely calling a potentially non-existant method. I often find myself doing (({obj.respond_to? :empty ? obj.empty : nil})), or if I'm feeling lazy, (({obj.empty? rescue nil})). Surely we can provide a less repetitive way of achieving duck-typing, e.g. I don't care what object you are, but if you (the object) can't tell me whether you're empty, I'm going to assume some value, or do something else instead.

I'm not sure what the best way to implement this is. The easiest would be to just define a new conditional send method:

```
obj.send_if(:empty?, *args) { nil }

obj.try(:empty?, *args) { nil }
```

But that's really not much of an improvement; it's ugly. Preferably, it'd be nice to build it into the language given how fundamental duck-typing is to Ruby. One potential syntax is:

obj.empty? otherwise nil

The ((|otherwise|)) keyword would be like a logical or, but instead of short-circuiting on true, it short-circuits on some other condition. That

condition can be one of two things. It can either wait for a NoMethodError (like an implicit (({rescue NoMethodError}))), proceeding to the next expression if one is raised, or it can do a pre-test using (({respond_to?})). Each option has its pro's and con's.

The implicit rescue allows you to include expressions, e.g.

```
obj.empty? otherwise obj.length == 0 otherwise true
```

Going with the implicit (({respond_to?})) implementation probably wouldn't allow that. You'd instead need to limit it just to method calls, which is not as useful. The only problem with implicitly rescuing NoMethodError's though, is that you'd need to ensure the NoMethodError was raised within the target object, and not some dependancy, as you could potentially swallow valid exceptions.

The benefit of this over current methods of duck-typing, is that you're not testing a condition, then running an action, you're instead doing both at the same time making it much more DRY.

One other potential syntax however is a double question mark, or question mark prefix. This could act as an implicit (({respond_to?})) pre-condition, returning nil if the method doesn't exist.

```
obj.empty??? || obj.length?? == 0 || nil
```

```
obj.?empty? || obj.?length == 0 || nil
```

I'm not completely satisfied with either syntax, so at this point I'm merely hoping to start a discussion.

Thoughts?
=end

#22 - 04/03/2013 08:53 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 03-04-2013 01:10, phluid61 (Matthew Kerwin) escreveu:

> Issue #8191 has been updated by phluid61 (Matthew Kerwin).
>
> =begin
> Throwing this out there for discussion: how about a completely different syntax?
>
> user&&.profile&&.website&&.thumbnail

Why not simply user&.profile&.website&.thumbnail?

#23 - 04/03/2013 08:53 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Em 02-04-2013 22:39, "Martin J. Dürst" escreveu:

> On 2013/04/03 9:54, wardrop (Tom Wardrop) wrote:
>
> > Issue #8191 has been updated by wardrop (Tom Wardrop).
> >
> > davidderyldowney (David Deryl Downey), ideally, a single question mark would probably
> > make that clearer, but alas, a single question mark serves another
> > purpose. The question mark prefix is always an option though as well,
> > e.g.
>
> How does this work with methods that already have a ? at the end? Will
> we get something like include???, or what?

From a practical view this is unlikely to happen because a method ending with a question mark should return true or false. What method would you like to call from those values?

#24 - 04/03/2013 10:53 PM - phluid61 (Matthew Kerwin)

On Apr 3, 2013 9:30 PM, "Rodrigo Rosenfeld Rosas" rr.rosas@gmail.com wrote:

> Em 03-04-2013 01:10, phluid61 (Matthew Kerwin) escreveu:
>
> > Issue #8191 has been updated by phluid61 (Matthew Kerwin).
> >
> > Throwing this out there for discussion: how about a completely different
> > syntax?

> user&&.profile&&.website&&.thumbnail

> Why not simply user&.profile&.website&.thumbnail?

Just because at a glance most of us recognise && as a logical operation and
& as arithmetic.

#25 - 04/04/2013 09:28 AM - wardrop (Tom Wardrop)

=begin
I think whatever the solution, it needs to be generic. Amending the language is a pretty big deal. Whatever we implement needs to have many potential applications. The nice thing about the double question mark is it is generic. It's a "access method or name only if it exists, otherwise nil". It can be used for tentative method chaining, safely accessing potentially non-existant local variables, dealing with inconsistant API's, etc. That's what I like about it. I wouldn't want a solution that only addressed the method chaining problem for example.

Personally, the double question mark syntax is still the best proposal. The question marks do pretty well to indicate the uncertainty of the operation, where ase using ((|&|)) is potentially confusing because of logical && and bitwise &.

Double question marks can work well with arguments too...

```
obj.call??('blah') { 'bleh' }
```

=end

#26 - 04/04/2013 02:24 PM - phluid61 (Matthew Kerwin)

=begin
wardrop (Tom Wardrop) wrote:

> Personally, the double question mark syntax is still the best proposal.
> The question marks do pretty well to indicate the uncertainty of the
> operation, [...]

I agree, but it seems a little bit teenage-girl to me (omg?? really???). I prefer (({obj.?method})) because:
(1) not too much punctuation (compare (({foo.?empty?})) vs (({foo.empty???})), or (({bar.?upcase!})) vs (({bar.upcase!??})))
(2) it makes (({.?})) a special operator, which is like a questionable version of (({.}))
(3) putting the question-mark before the method name gives it the same syntactic (and semantic) order as (({obj && obj.method})), rather than (({obj.method rescue nil})) (i.e. it suggests a proactive check, rather than a reactive recovery)

I still have an issue with the semantics of chaining. Should (({a.?b.?c})) or however you want to write it be parsed like chained ternary operations or like chained method calls? And what about side-effects? I.e. should it be equivalent to
a.respond_to? :b ? (a.b.respond_to :c ? a.b.c : nil) : nil
or
a.respond_to? :b ? ((tmp = a.b).respond_to? c : tmp.c : nil) : nil
(being a side-effect-limiting version of the above), or
(tmp = a.respond_to? :b ? a.b : nil).respond_to? :c ? tmp.c : nil

I've always imagined the third, but some other comments in this thread suggest people have other ideas. It's the only one that behaves like the send_if method in the original proposal.
=end

#27 - 04/04/2013 09:08 PM - Anonymous

wardrop (Tom Wardrop): Tom, you must be a masochist. Proposing this here is like volunteering
to run a gauntlet :-), and like others, I also have no choice but to whip you
some :-) Jim Weirich just recently entertained me so much with this phrase
"chicken typing", I had to write about it on WikiWiki, or otherwise I would burst:

http://c2.com/cgi/wiki?ChickenTyping

And chicken typing is what your proposal is about. Since I stopped being a chicken
in my code, I feel so much liberated!!! It feels so good, when you are writing
this object, to leave behind your worries of whether that object will or will not
be able to correctly handle your message! I think that all the newbie users of
Ruby should take an anti-chicken course!

http://talklikeaduck.denhaven2.com/2007/10/22/chicken-typing-isnt-duck-typing

#28 - 04/04/2013 10:20 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I don't believe this proposal, as I understand it, is about chicken typing. It would be if a??.b was equivalent to a.respond_to?(:b) ? a.b : nil. But, as I understand, the proposal is about a.nil? ? nil : a.b.

This has proved to be useful in many real-world software I often write both in Groovy and CoffeeScript like:

```
def isEditableByUser(user){
this.lawFirm.id == user.lawFirm?.id
}
def lastClient = TimeSpentWithClient.find('from TimeSpentWithClient t where t.user = ? order by endTime desc', [user])?.client ?: ''
```

In CoffeeScript, something that helps a lot and I'd love to see in Ruby is that it allows us to declare a variable in a post-if and use that variable in the statement:

```
console.log a if a = b.length
```

#29 - 04/05/2013 08:49 AM - phluid61 (Matthew Kerwin)

=begin
rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> I don't believe this proposal, as I understand it, is about chicken
> typing. It would be if a??.b was equivalent to a.respond_to?(:b) ? a.b :
> nil. But, as I understand, the proposal is about a.nil? ? nil : a.b.

This really needs to be cleared up once and for all. Are we proposing a chicken-typing syntax, or an andand syntax? The OP never made it clear.

Assuming chicken-typing is Bad(tm), my proposal is one or either of the following (pending discussion):

1. conditional method invocation a.?b.?c.?d # (tmp2 = ((tmp1 = ((tmp0=a) && a.b) && tmp0.c) && tmp1.d) # (tmp2 = (tmp1 = (tmp0 = a.nil? nil : a.b).nil? nil : tmp0.c).nil? nil : tmp1.d)

This means that in (({foo.?bar.inspect})), the (({foo.bar})) call may work or not, but (({.inspect})) will always be called on the result, possibly resulting in (({"nil"})). There remains a question of whether or not ((|false|)) is considered "valid."

Essentially (({.?})) becomes a pretty syntax for (({send_if})).

1. logic short-circuit a&&.b&&.c&&.d # (tmp0=a) && (tmp1=tmp0.b) && (tmp2=tmp1.c) && tmp2.d

This means that in (({foo&&.bar.inspect})) we'll either get the inspected form of (({foo.bar})), or ((|nil|)).

I believe both have their uses.

==== Incidentally

> In CoffeeScript, something that helps a lot and I'd love to see in Ruby
> is that it allows us to declare a variable in a post-if and use that
> variable in the statement:
>
> console.log a if a = b.length

This already works, as long as ((|a|)) is already defined in scope as a variable:

```
irb(main):001:0> a = nil
irb(main):002:0> b = 'abc'
irb(main):003:0> puts a if a = b[1]
b
=> nil
irb(main):004:0> b = ''
irb(main):005:0> puts a if a = b[1]
=> nil
```

=end

#30 - 04/05/2013 02:21 PM - Anonymous

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> ... But, as I understand, the proposal is about a.nil? ? nil : a.b.

In that case, it envies the feature of what is known to me as null object.
It took me some time to understand the concept, but people out there usually
implement null object in such way, that in response to almost all messages,
it returns self. And they also say that nil is not a good null object. So
the issue discussed would become "shall we support null object pattern more
in the core?", afaiui.

#31 - 04/05/2013 10:19 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

phluid61 (Matthew Kerwin) wrote:

> ==== Incidentally
>
> > In CoffeeScript, something that helps a lot and I'd love to see in Ruby
> > is that it allows us to declare a variable in a post-if and use that
> > variable in the statement:
> >
> > console.log a if a = b.length
>
> This already works, as long as ((|a|)) is already defined in scope as a variable:

This is exactly the point, Matthew. I don't want to have to use "a = nil" before just for the sake of having a declared variable. I believe the post-if should declare a variable that could be used by the statement if the variable doesn't exist yet, since that code should be executed before the statement anyway.

#32 - 04/06/2013 08:53 AM - phluid61 (Matthew Kerwin)

On Apr 5, 2013 11:20 PM, "rosenfeld (Rodrigo Rosenfeld Rosas)" <
rr.rosas@gmail.com> wrote:

> Issue #8191 has been updated by rosenfeld (Rodrigo Rosenfeld Rosas).
>
> phluid61 (Matthew Kerwin) wrote:
>
> > ==== Incidentally
> >
> > > In CoffeeScript, something that helps a lot and I'd love to see in
> > > Ruby
> > > is that it allows us to declare a variable in a post-if and use that
> > > variable in the statement:
> > >
> > > console.log a if a = b.length
> >
> > This already works, as long as ((|a|)) is already defined in scope as a
> > variable:
> > =end
>
> This is exactly the point, Matthew. I don't want to have to use "a = nil"
> before just for the sake of having a declared variable. I believe the
> post-if should declare a variable that could be used by the statement if
> the variable doesn't exist yet, since that code should be executed before
> the statement anyway.

This is off track for this thread, but to have it do what you want you'd
have to reengineer the parser to use look-aheads. This is not a trivial
change. Currently by the time it sees a=´ it's already had to make a
decision about whether the a inp a´ is a variable or method.

There are other tickets about this topic (I can't link them from my phone)
but this part of the discussion would be better had over there.

#33 - 04/06/2013 08:58 AM - wardrop (Tom Wardrop)

=begin
To clarify, there's no single official proposal here. My original post and subsequents posts touch on various syntactical and behavioural possibilities. Ignoring syntax for the moment, there seems to be 3 potential behaviours we could implement:

(1) Abort on nil, or on falsey
(2) Abort when #respond_to? is false
(3) Abort on NoMethodError

I think I'm starting to feel that #respond_to? is too fragile. You really don't know if an object will respond to anything until it's called. Hence option's 1 and 3 seem more practical, though they're quite different. We may even find them to be complimentary and decide to implement both.

```
user && .profile && .name
```

In the previous example, we only want to call the next method if the previous expression returned truthy. We want to know about any NoMethodError's in this case, as if #profile returns truthy, but it doesn't respond to the expected method, there's a problem with the code.

...I have to go for the moment, but the question mark syntax is for cases where you know very little about the object(s) you're dealing with, or the environment your code is run in (e.g. templates). I feel I need to give better use cases for this particular set of use cases though.
=end

#34 - 04/06/2013 09:18 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Matthew, I agree this is off-topic and I've already read about why it's hard to support that in another issue... Sorry to bring that to this discussion. Let's ignore it from now on.

Just to be clear, I'm +1 only for a??.b meaning (a.nil? ? nil : a). Any other alternative I'd vote against. I guess maybe the best thing to do now is to create 4 new tickets so that we can be sure what to discuss about:

1. Abort on nil
2. Abort on falsey
3. Abort when respond_to? is false
4. Abort on NoMethodError

While the discussion about the syntax for the short-hand is valid, I strongly believe Matz will reject 2, 3 and 4 promptly, so we could focus on discussing 1 unless Matz rejects it as well regardless of the chosen syntax.

After creating the other tickets I think this one should be closed.

#35 - 04/06/2013 09:19 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I forgot to state why I believe 2 shouldn't be accepted. It doesn't make sense in any of the real-world software I've worked with to call any method on "false", so what is the point of 2?

#36 - 04/06/2013 11:01 AM - phluid61 (Matthew Kerwin)

=begin
rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> 1. Abort on nil
> 2. Abort on falsey
> 3. Abort when respond_to? is false
> 4. Abort on NoMethodError

I agree about 3 and 4, it's too chicken-typey and covers up legitimate architectural issues.

However this list fails to address the other dimension of the possible proposals. When you say "abort on", does that mean that in (({a.b.c.d.e})) (modulo syntax), if (({.c})) "aborts," do (({.d})) and (({.e})) get called or not? If not, do we have to use (({(a.b.c).d.e})) to force them to run?

I'd say the (up to) four proposals should be:

(1) abort on nil
(2) abort on falsey (which I described earlier as (({&&.})))
(3) send_if not nil (which I described earlier as (({.?})))
(4) send_if not falsey

I'm happy for 4 to be ignored. The reason I put forward 2 over 1 is that it matches the (({a && a.b && a.b.c})) pattern.
=end

#37 - 04/06/2013 11:11 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Matthew, I'm curious. What method would you call over "false"? It would help if could provide a real-world use case showing how 2 could be useful.

With regards to the "abort on", please disregard any concrete meaning for it. I just kept Tom's terminology to avoid confusion. I believe it is a separate discussion whether we should "abort" or "send" accordingly to your terminology. I believe both discussions are valid. CoffeeScript takes the && (abort) approach:

a = null; a?.b.c is undefined

Groovy takes the "abort" approach:

a = null; a?.b.c // ERROR java.lang.NullPointerException: Cannot get property 'c' on null object

I haven't thought much yet which behavior I'd prefer as I can see utility in both. But first I'd like to eliminate any discussions about falsey evaluations or chicken-typey behavior...

By the way, CoffeeScript doesn't implements the falsey evaluation either as you can see:

a = false; a?.b.c # TypeError: Cannot read property 'c' of undefined

#38 - 04/06/2013 01:36 PM - phluid61 (Matthew Kerwin)

=begin
rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Matthew, I'm curious. What method would you call over "false"?

Sorry, I don't quite understand what you're asking. What method would I call on false? None, which is why it makes sense to me to avoid calling the method if the result is ((|false|)) or ((|nil|)). This is reinforced by the common pattern (({get_value && do_something})), as opposed to the much less common (({get_value.nil? ? nil : do_something}))

> It would help if could provide a real-world use case showing how 2 could
> be useful.

Earlier on Tom posted the code:

if user && user.profile && user.profile.website && user.profile.website.thumbnail

I've seen this pattern before so I assume it's fairly common. Option 2 is a nicer version of the same, which has less typing, and also eliminates repeated side-effects from any of the chained methods.

My proposal 3/4 is the one I'm less sure of, except that (as far as I can see) that's what ((*andand*)) does, and it's definitely the easier to implement. I suppose it could have a use in debugging:

puts user.?profile.?website.inspect # => 'www.example.com' or 'nil'

or adhering to an external schema that, let's say, uses ((|0|)) to represent null values:

dept_id = user.department.?id.to_i
=end

#39 - 04/06/2013 08:11 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Matthew, if you have tried to provide a real use case you'd understand what I meant. For user.profile.website.thumbnail, for instance, don't you agree that neither user, profile or website are expected to be possibly "false" values? You'd never call false.thumbnail, right? And a website should never be false, although it could be nil, right? That's what I'm talking about. There's no sense in checking for false values before calling a method on it because no one in any real application would be calling any meaningful method over the "false" object, right?

That's why neither CoffeeScript (abort approach) nor Groovy (send_if approach) check for false, but only for null.

#40 - 04/07/2013 12:53 AM - phluid61 (Matthew Kerwin)

On Apr 6, 2013 9:12 PM, "rosenfeld (Rodrigo Rosenfeld Rosas)" <
rr.rosas@gmail.com> wrote:

> Matthew, if you have tried to provide a real use case you'd understand
> what I meant. For user.profile.website.thumbnail, for instance, don't you
> agree that neither user, profile or website are expected to be possibly
> "false" values? You'd never call false.thumbnail, right? And a website
> should never be false, although it could be nil, right? That's what I'm
> talking about. There's no sense in checking for false values before calling
> a method on it because no one in any real application would be calling any
> meaningful method over the "false" object, right?

My guiding principle is: if the syntax mimics &&, the behaviour should test
truthiness. My proposed abortive mechanism was `&&.´

#41 - 04/08/2013 02:32 PM - wardrop (Tom Wardrop)

I guess we've determined that Ruby doesn't provide any fool proof means of detecting whether an object will or should respond to a method; any attempt to do so currently results in unreliable and potentially hazardous chicken-typing. This request has therefore shifted focus to logical method chaining which at least addresses part of the problem.

I've raised issue #8237 as an official proposal for logical method chaining.

I still feel that there should be some means to safely call a potentially non-existant method on an object. As we know, objects can be extended as singletons, so really, there needs to be a way to know whether some object has some method. The problem is that you can't know until you execute the method. An implicit catch of a NoMethodError is the only means you can tell whether a method call was successful or not.

#42 - 04/08/2013 09:44 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

The best-practices states that you should override responds_to? when you override method_missing. That way, calling responds_to? should be enough.

#43 - 04/09/2013 12:37 AM - marcandre (Marc-Andre Lafortune)

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> The best-practices states that you should override responds_to? when you override method_missing. That way, calling responds_to? should be enough.

Actually, since 1.9 the best practice is to ovverride respond_to_missing?, not respond_to?

#44 - 04/09/2013 12:47 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Sorry, I'm a bit outdated, since I never really felt the need for method_missing in my own code :)

#45 - 04/09/2013 02:41 AM - Anonymous

rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> Sorry, I'm a bit outdated, since I never really felt the need for method_missing in my own code :)

How different people can be! Programming without #method_missing would be maddening like
driving without functioning horn :-)

#46 - 04/09/2013 09:28 AM - wardrop (Tom Wardrop)

Maybe a more convenient syntax that makes use of #respond_to? would promote the use of #respond_to_missing. In that case, maybe the double question mark or question mark prefix can remain on the table. I'm still personally for it as I would make heavy use of such a convenience.

#47 - 09/18/2015 09:22 AM - akr (Akira Tanaka)

- Related to Feature #8237: Logical method chaining via inferred receiver added

#48 - 09/18/2015 09:52 AM - akr (Akira Tanaka)

- Related to Feature #11537: Introduce "Safe navigation operator" added

#49 - 11/12/2015 01:16 AM - hsbt (Hiroshi SHIBATA)

- Status changed from Assigned to Closed

this feature is supported at #11537