Now that #to_h is the official method for explicit conversion to Hash, we should also add

Enumerable#to_h: Returns a hash for the yielded key-value pairs.

[[:name, 'Joe Smith'], [:age, 42]].to_h # => {name: 'Joe Smith', age: 42}

With the Ruby tradition of succinct documentation I suggest the documentation talk about key-value pairs and there is no need to be explicit about the uninteresting cases like:

{1..3}.to_h # => {1 => nil, 2 => nil, 3 => nil}

{[1, 2], [1, 3]}.to_h # => {1 => 3}

{[1, 2], []}.to_h # => {1 => 2, nil => nil}

I see some reactions of people reading about the upcoming 2.0 release like this one:
http://globaldev.co.uk/2012/11/ruby-2-0-0-preview-features/#dsq-comment-body-700242476
array.c: Add Array#to_h [Feature #7292]
enum.c: Add Enumerable#to_h

array.c: Add Array#to_h [Feature #7292]
enum.c: Add Enumerable#to_h

array.c: Add Array#to_h [Feature #7292]
enum.c: Add Enumerable#to_h

Revision 43401 - 10/23/2013 03:10 PM - marcandre (Marc-Andre Lafortune)

array.c: Add Array#to_h [Feature #7292]
enum.c: Add Enumerable#to_h

Revision 43401 - 10/23/2013 03:10 PM - marcandre (Marc-Andre Lafortune)

array.c: Add Array#to_h [Feature #7292]
enum.c: Add Enumerable#to_h

Revision 43401 - 10/23/2013 03:10 PM - marcandre (Marc-Andre Lafortune)

array.c: Add Array#to_h [Feature #7292]
enum.c: Add Enumerable#to_h

History

#1 - 11/07/2012 04:56 AM - nathan.f77 (Nathan Broadbent)
I agree, Enumerable#to_h would make sense and be quite useful.

(1..3).to_h would be a special case for the Range class, because [1, 2, 3].to_h should raise an exception.

Here's an example in Ruby:

```ruby
module Enumerable
  def to_h
    hash = {}
    each_with_index do |el, i|
      raise TypeError, "(at index #{i}) Element is not an Array" unless Array === el
      raise IndexError, "(at index #{i}) Array has more than 2 elements" if el.size > 2
      hash[el[0]] = el[1]
    end
    hash
  end
end
```

#2 - 11/07/2012 08:44 AM - matz (Yukihiro Matsumoto)
- Status changed from Open to Feedback
- Priority changed from Normal to 3

So what's the difference from rejected #7241?
So what's the difference from rejected #7241?

The main difference is that to_h wouldn't take a block or any arguments. It would be a simple conversion from Enumerable to Hash, and would only support a collection of arrays containing a maximum of 2 elements.

Use the traditional Hash[] in 2.0.0. I'm moving this ticket into the feature tracker.

```
p Hash[{:name, 'Joe Smith'}, {:age, 42}]
 #=> {name: 'Joe Smith', age: 42}
```

Yusuke Endoh mame@tsg.ne.jp

mame (Yusuke Endoh) wrote:

```
As Nathan said, #7241 (and #666) accept a block and are therefore more related to the more complex categorize/associate/... #4151.

The implementation for to_h would be as simple conceptually as possible. It would be equivalent to:

```ruby
module Enumerable
def to_h
  result = {}
  each do |key, value|
    result[key] = value
  end
  result
end
end
```

I believe this is the simplest definition one can think of. It doesn't try to do much, nor is it too strict (in the same way that "two".to_i returns 0).

mame (Yusuke Endoh) wrote:

```
Use the traditional Hash[] in 2.0.0.
```

Indeed, Hash[] can be used instead, except it's really really ugly.

I can't think of any other global method we use like this that should be an instance method. It's very natural to transform data into a hashes, but instead of chaining the transformations we have to reverse the flow for this step. E.g. source.map{...}.to_h.merge(...) reads naturally, but Hash[source.map{...}].merge(...) doesn't.

The only other example of SomeClass[] I can think of is for Set. In that case, it's understandable as Set doesn't have a dedicated creation syntax, so Set[1, 2, 3] has its charms. Are there other cases, besides Hash[]?

I'm moving this ticket into the feature tracker.
 Didn't I create it as a feature request?

mame (Yusuke Endoh) wrote:

```
I'm moving this ticket into the feature tracker.
```

marcandre (Marc-Andre Lafortune) wrote:

```
I'm moving this ticket into the feature tracker.
```

marcandre (Marc-Andre Lafortune) wrote:

```
I'm moving this ticket into the feature tracker.
```
Didn't I create it as a feature request?

Oops, I was mistaken. I just set the target to next minor. Sorry.

--
Yusuke Endoh mame@tsg.ne.jp

#8 - 11/08/2012 03:01 AM - prijutme4ty (Ilya Vorontsov)

Hash[] is one of most disastrous ruby methods, IMHO. Since we don't have hash_map it's common to write smth like
hsh = Hash[ hsh.map{|k,v| [k.to_sym, v.to_f]} ]
In some more complicated cases it makes any programmer, who looks at code, cry.
Actually I'd prefer to have both methods Enumerable#to_h and Hash#hash_map ( http://bugs.ruby-lang.org/issues/6669 )
Programmers anyway use analogues for this method, so it'd be a way to standardize their code. As marcandre said #to_i also isn't ideal but is very useful and each programmer understand it the same way.

#9 - 11/08/2012 03:58 AM - marcandre (Marc-Andre Lafortune)

I'm a strong supporter for different hash_map/associate/categorize, but let's not discuss these here please, they have their own tickets (#4151 & #6669).
This request is not meant to be a replacement for those requests. It is a small step, the simplest method to explicitly convert an Enumerable to a Hash.

#10 - 11/08/2012 06:32 AM - bitsweat (Jeremy Daer)

+1 to this.
I didn't like it at first because #to_h means coercion to me, and it doesn't make sense to coerce an Enumerable to a Hash. However, Array#to_h does seem like a good fit. Coerce this array of associated key/value pairs to a hash. Deal with edge cases in the same was as Hash[].
I'd immediately change a lot of code to use this if it was available. Ending a chain of enumerable methods with .to_h is much nicer than "going back" to wrap it in Hash[].
(Perhaps Enumerable#to_h could remain as a shortcut for to_a.to_h?)

#11 - 03/01/2013 09:50 PM - jipiBoily (Jean-Philippe Boily)

+1
This would just feel right and natural to me.

#12 - 03/13/2013 08:05 PM - newmen (Gleb Averchuk)

I think this is very cool feature, because I'm tired of writing something like this:
some_hash = Hash[some_hash.map { |k, v| [k, (v * scale).to_i] }]

P.S.
In actual fact is not very tired. :) And it may have a more elegant way that will change the Hash by using .map method.

#13 - 03/14/2013 02:12 AM - drbrain (Eric Hodel)

=begin
There is a potential for a security exploit with Enumerable#to_h:
user_input = %w[rm -rf /]
system ['ls', '-I', *user_input]
With system, the first argument is used as the environment if it can be converted to a Hash. With user input to system this may lead to arbitrary code execution.
=end

#14 - 03/14/2013 02:23 AM - marcandre (Marc-Andre Lafortune)
drbrain (Eric Hodel) wrote:

There is a potential for a security exploit with Enumerable#to_h:

```ruby
user_input = %w[rm -rf /]
system ['ls', '-l'], *user_input
```

With system, the first argument is used as the environment if it can be converted to a Hash. With user input to system this may lead to arbitrary code execution.

I think you are confusing to_h (explicit conversion) with to_hash (implicit conversion). system calls rb_check_hash_type which will attempt to call to_hash but will not send to_h on its argument.

So no, there is no such potential security risk here.

#15 - 04/05/2013 01:51 AM - rogerdpack (Roger Pack)
+1 from me. Sometimes after converting from an array to a hash I want to "convert back" to a hash and inevitably I reach for "to_h" just to discover it's not there.

#16 - 07/24/2013 06:34 PM - alexeymuranov (Alexey Muranov)
I have stumbled upon a need for a method like this, to chain transformations of a hash and get a hash as a result. Just a quick thought (please tell me if i have overlooked something): it seems to me that other "#to_?" methods are applicable to all or almost all instances of a class, whereas here the method would be applicable only to a special kind of arrays: the ones consisting of key-value pairs.

Maybe there is no need to call it "#to_h", and it is better to reserve "#to_h" for some operation applicable to all arrays? Maybe the proposed method can be called something like "#as_hash", "#as_h", or a different name?

```ruby
1, 2].as_hash # => {1=>2, 3=>4}
```

To generalize this, maybe "as_?" methods can be defined as left inverses of "to_?" methods (in method chaining, they should probably be called right inverses):

```ruby
{1=>2, 3=>4}.to_a.as_hash # => {1=>2, 3=>4}
{1=>2, 3=>4}.to_s.as_hash # => {1=>2, 3=>4}
"{1=>2, 3=>4}".as_hash # => {1=>2, 3=>4}
```

#17 - 07/25/2013 06:28 AM - marcandre (Marc-Andre Lafortune)
alexeymuranov (Alexey Muranov) wrote:

it seems to me that other "#to_?" methods are applicable to all or almost all instances of a class

String#to_i is not meaningful on most strings.

#18 - 07/26/2013 05:41 PM - alexeymuranov (Alexey Muranov)
Yes, thanks, i forgot. Then "to_h" would be fine with me.

In fact, for me it would be enough to have a method like "yield_self" #6721, then i would do "array.yield_self [{i|a|Hash[a]}]

#19 - 07/27/2013 03:07 PM - matz (Yukihiro Matsumoto)
- Status changed from Assigned to Feedback

the name 'to_h' is OK, simpler behavior is preferable compared with the past proposals.

But I am not sure the following simple implementation works OK, e.g. what if an element is a object, or number, or anything not two-element array.

```ruby
module Enumerable
def to_h
  result = {}
  each do |key, value|
    result[key] = value
  end
  result
end
end
```

Matz.
#20 - 07/28/2013 03:42 AM - alexeymuranov (Alexey Muranov)

begin
I would suggest

module Enumerable
def to_h
  h = []
each do |e|
    h[e.first] = e.last
  end
  h
end
end

#21 - 08/31/2013 07:38 AM - marcandre (Marc-Andre Lafortune)
- File to_h.pdf added
- Status changed from Feedback to Open

matz (Yukihiro Matsumoto) wrote:

But I am not sure the following simple implementation works OK, e.g. what if an element is a object, or number, or anything not two-element array.

Agreed.

I believe we should only treat elements that are array-like and of length 2. More explicitly, either the Enumerable yields one value that responds_to?(:to_ary) and returns a 2-element array, or the Enumerable yields exactly two values. Other cases should be ignored, in the same way that String#to_i ignores invalid characters.

Slide attached.

#22 - 08/31/2013 01:38 PM - matz (Yukihiro Matsumoto)
- Status changed from Open to Feedback

What I wanted was corner case behavior of #to_h, e.g. what if elements are not 2 elements arrays.

What kind of checks do you want to do?

The simplest implementation in #6 may work, but I'm not sure whether kind of accidental behavior definition is suffice.

Matz.

#23 - 08/31/2013 09:26 PM - trans (Thomas Sawyer)

[omit verbose intro] suffice to say we can figure the most fitting definition for ((Enumerable#to_h))) is simply:

module Enumerable
def to_h
  a = []
each_with_index.each { |e,i| a << i << e }
  Hash[*a]
end
end

[:a,:b].to_h #=> {0=>:a, 1=>:b}

We can answer why in the nicest of ways too: What is it we are converting to a hash table? It is an ((Enumerable)). So it only stands to reason that the conversion reflect the ((enumeration)). Another nice thing about this definition is there are no corner cases to worry about.

To convert an associative array to a hash, that is a different goal. And as Ruby currently stands, that is best addressed with ((Hash[*assoc.flatten(1)]))). For something better in that regard I would suggest the addition of a new method, maybe ((Hash.from_assoc(assoc)))).

end

#24 - 09/01/2013 04:05 AM - marcandre (Marc-Andre Lafortune)
- Status changed from Feedback to Open
- Priority changed from 3 to Normal
matz (Yukihiro Matsumoto) wrote:

What I wanted was corner case behavior of #to_h, e.g. what if elements are not 2 elements arrays.
What kind of checks do you want to do?

The simplest implementation in #6 may work, but I'm not sure whether kind of accidental behavior definition is suffice.

I think it might be best to ignore anything that is not a key-value pair. So we should use an implementation slightly different from #6. In Ruby:

```ruby
module Enumerable
  def to_h
    h = {}
    each_entry do |ary|
      next unless ary.respond_to?(:to_ary)
      ary = ary.to_ary
      raise TypeError unless ary.is_a?(Array)
      next unless ary.size == 2
      h[ary.first] = ary.last
    end
    h
  end
end
```

Note that I am using each_entry, so yield(:key, :value) is treated the same as yield([:key, :value]).

#25 - 09/02/2013 10:01 AM - matz (Yukihiro Matsumoto)
Acceptable. How others think about Marc's rule?

- elements should respond to #to_ary
- return value from #to_ary should be 2 elements array
- otherwise the element will be ignored (no TypeError exception)

If no one objects, I'd be fine. Marc, do you want to implement it by yourself, or ask somebody to do so?

Matz.

#26 - 09/02/2013 11:01 AM - marcandre (Marc-Andre Lafortune)
- Assignee changed from matz (Yukihiro Matsumoto) to marcandre (Marc-Andre Lafortune)

matz (Yukihiro Matsumoto) wrote:

Marc, do you want to implement it by yourself, or ask somebody to do so?

Great!
Sure, I can implement it.

#27 - 09/02/2013 11:59 AM - phluid61 (Matthew Kerwin)
On Sep 2, 2013 11:02 AM, "matz (Yukihiro Matsumoto)" matz@ruby-lang.org wrote:

Acceptable. How others think about Marc's rule?

- elements should respond to #to_ary
- return value from #to_ary should be 2 elements array
- otherwise the element will be ignored (no TypeError exception)

+1, this proposal is as good as any I've seen.

#28 - 09/02/2013 06:38 PM - alexeymuranov (Alexey Muranov)
Why #to_ary and not #to_a? Or just expect the elements of the enumerable collection to respond to #first and #last.

If someone implements a class OrderedPair, it is not sure in my opinion that the instances would respond to #to_ary.

#29 - 09/02/2013 06:52 PM - alexeymuranov (Alexey Muranov)
I understand that using #to_a or #first and #last directly would give an unexpected result when calling #to_h on a collection of ranges, for example, but one is not supposed to call #to_h on a collection of ranges, or #to_h should be preceded with #select.
The two #next and one #raise look a bit like defensive programming to me, and could cause an unnecessary slowdown. Wouldn't it be better to let the user decide when to precede #to_h with #select?

Edited

#30 - 09/03/2013 03:04 AM - alexeymuranov (Alexey Muranov)

Another alternative: since two-element arrays are used here as ordered pairs, maybe the Array class can be extended with #key and #value methods, which would be identical to #first and #last respectively on two-element arrays, and raise errors otherwise. Then #to_h can be implemented as

```ruby
module Enumerable
  def to_h
    h = {}
    each_entry do |pair|
      h[pair.key] = pair.value
    end
    h
  end
end
```

It would be then applicable to any collection of objects that respond to #key and #value.

If #key and #value seem to be overused as names, maybe better names can be found (e.g. #key_entry, #value_entry).

So, the idea is to extend Array simultaneously with Enumerable.

Edited

#31 - 09/04/2013 02:46 AM - trans (Thomas Sawyer)

```ruby
=end
marcandre (Marc-Andre Lafortune) That implementation is limited by to_ary and it does some weird things.
[ [:a], [:b,1,2], [:c,3] ].to_h #=> {:c=>3}
```

I know what you want is to convert an associative array into a hash. That's a good thing to have, I agree! But Enumerable#to_h is not a good method for it. It doesn't "semant".

At most it should be Array#to_h and work like:

```ruby
[ [:a], [:b,1,2], [:c,3] ].to_h #=> {:a=>nil, :b=>1, :c=>3}
```

Or

```ruby
[ [:a], [:b,1,2], [:c,3] ].to_h #=> {:a=>nil, :b=>[1,2], :c=>3}
```

Or

```ruby
[ [:a], [:b,1,2], [:c,3] ].to_h #=> {:a=>nil, :b=>[1,2], :c=>[3]}
```

Probably it could take an option to select which mode is desired.

On the other hand, I am not so sure it shouldn't have a different name altogether, e.g. Array#assoc_hash.

=end

#32 - 09/04/2013 08:14 AM - matz (Yukihiro Matsumoto)

```ruby
trans (Thomas Sawyer) I am sure
[ [:a], [:b,1,2], [:c,3] ].to_h #=> {:a=>[], :b=>[1,2], :c=>[3]}
```

is not what we want. It destroys common cases for the sake of consistency.

If you want different behavior from proposed one, please show us rational more than vague impression.

For me, using #to_h on non 2 elements array is exceptional, so any behavior is OK if it's well-defined, and works for common cases.

Matz.

#33 - 09/04/2013 11:15 AM - trans (Thomas Sawyer)

```ruby
=end
matz (Yukihiro Matsumoto)
```
How does it "destroy common case"?

```ruby
[[:a, 1], [:b, 2], [:c, 3]].to_h #=> {:a=>1, :b=>2, :c=>3}
```

Would work just fine. That was my first example case.

The next two show what other basic conversions of assoc array to hash there can be. And the "consistent" case you mention certainly can be useful. So my suggestion was to have a parameter, e.g.

```ruby
class Array
  def to_h(type=nil)
    h = {}
    if type.nil?
      each{|k, v, *| h[k] = v }
    elsif type == :array
      each{|k, *v| h[k] = v }
    elsif type == :ones
      each{|k, *v| h[k] = v.size > 1 ? v : v[0] }
    else
      raise ArgumentError, "unknown conversion type for Array#to_h -- '#{type}'"
    end
    h
  end
end
```

That way all are possible.

---

**#34 - 09/04/2013 06:25 PM - alexeymuranov (Alexey Muranov)**

By the way, shouldn't the behavior be somewhat consistent with Array#assoc and Array#rassoc? Than would mean, in my opinion,

```ruby
[[:a, 1], [:a, 2], [:b, 3, 4]].to_h # => {:a=>1, :b=>3} or {:a=>1} or Error, but not {:a=>2}
```

**#35 - 09/04/2013 08:20 PM - Eregon (Benoit Daloze)**

#to_h should have no parameter, just a single well-defined behavior. 
#to_h is for converting for the most simple case(s), if more control is needed, just make your own conversion.

And I think it would be much easier if it was just Hash, but ignoring instead of raising an exception. That would be consistency.

Marc-André's rule is Hash[] for an Array with no exceptions and is fine in my opinion.

(One possible case not supported is even length arguments with no nested arrays (Hash[1,2,3,4]), but that is not so well defined as in Hash*. Depending on the first element being an Array or not to detect this case seems a bad idea).

**#36 - 09/04/2013 10:04 PM - matz (Yukihiro Matsumoto)**

Alexey, define "consistent" first. It's more difficult than you'd expect.
I don't usually vote for "consistency" except when there's clear benefit.

Matz.

**#37 - 09/05/2013 04:17 AM - alexeymuranov (Alexey Muranov)**

Matz,

it was just a reminder about #assoc and #rassoc, sorry if it was redundant. IMHO, they serve a similar purpose to #to_h: they allow to use an array of two-element arrays as a storage where selection "by key" or "by value" is possible.

I think, if #assoc and #to_h were introduced simultaneously, the following would have given identical results:

```ruby
[[:a, 1], [:a, 2]].assoc(:a)    # => [:a, 1]
[[:a, 1], [:a, 2]].to_h.assoc(:a) # => [:a, 2] with any of the suggested above implementations
```

I probably didn't use "consistent" correctly, not in mathematical sense. I meant something closer to "natural": that as many operations or diagrams commute as possible. That is, when appropriate, the result of `((x.foo.bar))` should be the same as that of `(((x.bar)))`, or if `(((#foo)))` applies some "essential" transformation to `(((x)))`, but there exists an operation `(((#baz))` applicable to `(((x.bar)))` that is a "counterpart" of `(((#foo)))`, then it would be nice if `(((x.foo.bar)))` was identical with `(((x.bar.baz)))`, if it makes sense. Here are the corresponding "commuting diagrams" (not exactly, but this gives an idea):

```ruby
x -- #foo --> x.foo
   |    |
   #bar #bar
```
x.bar == x.foo.bar

x ---------- #foo ----------> x.foo
| #bar | #bar
| V | V
x.bar -- #baz --> x.bar.baz == x.foo.bar

I do not insist, i am just trying to explain what i meant.

Update: I think instead of "#to_h is consistent with #assoc", it is more correct to say "#to_h agrees with #assoc".

#38 - 09/05/2013 06:03 AM - alexeymuranov (Alexey Muranov)
Wait! Shouldn't enum.to_h be the same as Hash[enum]?

#39 - 09/05/2013 11:14 AM - Anonymous
I think that there are two basic possibilities for Enumerable#to_h behavior:

Strict:
:a, 1], [:b, 2].to_h #=> { :a => 1, "b" => 2 }
Anything else raises a TypeError:
:a, 1], [:b, 1, 2].to_h #=> TypeError
:a, 1], [:b, 2, 3].to_h #=> TypeError

Lax:
:a, 1], [:b, 1, 2], [:c, 3].to_h #=> { :a=>[], :b=>[1,2], :c=>[3] }

"Strict" means, that the method strictly requires the arguments to be size 2 arrays.
"Lax" means, that the arguments are allowed to be arrays of any size >= 1.

I found it useful with plenty of usecases to also define Enumerable#>> as follows:

module Enumerable; def >> other; Hash[ zip other ] end end
[:a, :b, :c] >> [1, 2, 3] #=> {a: 1, b: 2, c: 3}

I also enjoyed to alias #first and #drop(1) with words #car and #cdr:

module Enumerable; def car; first end end
[:a, :b, :c].car #=> :a
module Enumerable; def cdr; drop 1 end end
[:a, :b, :c].cdr #=> [:b, :c]

The "lax" version of the proposed Enumerable#to_h can then be written as:

x = [:a, [:b, 1, 2], [:c, 3]
x.map( &:car ) >> x.map( &:cdr ) # <-- This is my opinion what Enumerable#to_h should do.

The last line does what I think that Enumerable#to_h should do. I realize that this opinion of mine directly contradicts what Matz said earlier. The argument for it would go somehow like this:

Since there are two basic possibilities for Enumerable#to_h behavior, and the strict one is already available as Hash[...], Enumerable#to_h should do the other useful thing: The "lax" version. I noticed similar design pattern between eg. #to_i and Integer(...): Both are useful, but not the same.

With apologies for arguing,
boris >(°.°)<

#40 - 09/06/2013 01:18 AM - rosenfeld (Rodrigo Rosenfeld Rosas)
I vote for raising an exception when trying to convert an invalid array to hash (considering the common case the valid array format).

#41 - 10/24/2013 12:10 AM - marcandre (Marc-Andre Lafortune)
This issue was solved with changeset r43401.
Marc-Andre, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

- array.c: Add Array#to_h [Feature #7292]
- enum.c: Add Enumerable#to_h

Files

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>Date</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_h.pdf</td>
<td>85.1 KB</td>
<td>08/31/2013</td>
<td>marcandre (Marc-Andre Lafortune)</td>
</tr>
</tbody>
</table>