### Ruby master - Feature #18273

**Class#subclasses**

10/27/2021 01:27 PM - byroot (Jean Boussier)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

#### Description

Ref: [https://github.com/rails/rails/pull/43481](https://github.com/rails/rails/pull/43481)

Something we forgot to mention in [Feature #14394](https://github.com/rails/rails/pull/43481), is either a parameter or another method to only get direct descendants.

Active Support has been offering Class.subclasses as:

```ruby
def subclasses
  descendants.select { |descendant| descendant.superclass == self }
end
```

It seems a bit silly to grab all descendants and then restrict the list when Class#descendants had to do some recursion to get them all in the first place.

#### Proposal

We could either implement Class#subclasses directly, or accept a parameter in Class#descendants, e.g. descendants(immediate = false).

cc [Eregon (Benoit Daloze)](https://github.com)

#### Related issues:

- Related to Ruby master - Feature #14394: Class.descendants

#### Associated revisions

Revision c0c2b31a - 11/23/2021 09:50 AM - byroot (Jean Boussier)

Add Class#subclasses

Implements [Feature #18273]

Returns an array containing the receiver's direct subclasses without singleton classes.

#### History

**#1 - 10/27/2021 02:40 PM - byroot (Jean Boussier)**

- Subject changed from Class.subclasses to Class#subclasses

**#2 - 10/27/2021 02:40 PM - byroot (Jean Boussier)**

- Related to Feature #14394: Class.descendants added

**#3 - 10/28/2021 11:31 AM - Eregon (Benoit Daloze)**

+1 for Class#subclasses, I think it makes sense since we have Class#descendants.

Class#subclasses is then also the "complement" of Class#superclass, just like Class#descendants is the complement of Class#ancestors.

**#4 - 10/28/2021 12:10 PM - byroot (Jean Boussier)**

I took a stab at it: [https://github.com/ruby/ruby/pull/5045](https://github.com/ruby/ruby/pull/5045)

**#5 - 11/16/2021 07:41 AM - mame (Yusuke Endoh)**

I am afraid if the name "subclasses" is suitable. When we have three classes C, D that inherits from C, and E that inherits from D, we say "E is a subclass of C".

---

03/24/2022
In fact, the rdoc of Module#< says

```
mod < other → true, false, or nil
```

Returns true if mod is a subclass of other.

and E < C returns true.

```
class C; end
class D < C; end
class E < D; end
```

```
p E < C #=> true
```

So, Class#subclasses sounds the same as Class#descendants.

If we take family line as an analogy, Class#children might be good. ko1 (Koichi Sasada) suggested Class#direct_subclasses.

---

#6 - 11/16/2021 08:49 AM - byroot (Jean Boussier)

Fine by me, we'll have to go through a deprecation period in Rails but that's not a big deal.

What about it being a boolean parameter on descendants? Was this option considered?

#7 - 11/16/2021 08:55 AM - sawa (Tsuyoshi Sawada)

I agree with mame, which brings in another question: Was the method name Class#superclass appropriate? Perhaps this method should also be renamed, say, to Class#parent.

An alternative is to allow an optional keyword direct (or immediate) on both Class#ancestors and Class#descendants.

#8 - 11/16/2021 09:02 AM - mame (Yusuke Endoh)

byroot (Jean Boussier) wrote in #note-6:

```
What about it being a boolean parameter on descendants? Was this option considered?
```

descendants(immediate = false) looks reasonable to me. A keyword argument might be preferable to an optional argument.

sawa (Tsuyoshi Sawada) wrote in #note-7:

```
Was the method name Class#superclass appropriate?
```

I think we don't say "C is a superclass of E" in the same situation, but I'm unsure.

#9 - 11/16/2021 09:07 AM - mame (Yusuke Endoh)

mame (Yusuke Endoh) wrote in #note-8:

```
sawa (Tsuyoshi Sawada) wrote in #note-7:
```

```
Was the method name Class#superclass appropriate?
```

I think we don't say "C is a superclass of E" in the same situation, but I'm unsure.

Ah, I misunderstood. I have no idea whether an alias Class#parent is needed.

#10 - 11/16/2021 12:24 PM - Eregon (Benoit Daloze)

AFAIK superclass is always understand as the direct superclass.

And given we already have Class#descendants and Module#ancestors, I feel there is very little room for confusion for Class#subclasses (i.e., almost no confusion possible).

Of course we can document it clearly in that method's documentation it's only "classes which have C has the superclass", and link to Class#descendants.

IMHO Class#direct_subclasses is verbose and not helpful. Rails already established that subclasses is a clear meaning for this.

Regarding a boolean argument for Class#descendants I'm neutral, I think that's less good and one definitely need to read the docs just to find the meaning of the boolean, plus they might not guess it even takes an argument.
Class#subclasses is clearer and is already an established name for this.

I think we don't say "C is a superclass of E" in the same situation, but I'm unsure.

If there is any sense in understanding the word "superclass" to mean only the direct ancestor, that is because it is in singular form, not because the word per se implies directness. The singular form leads a programmer to infer that it must mean something unique, hence the direct ancestor, excluding the indirect ones. The word itself does not imply directness. In fact, there are some definitions out there that explicitly mention indirect ancestors to be included in "superclass":

- https://www.whitman.edu/mathematics/java_tutorial/java/javaOO/subclasses.html
  The term superclass refers to a class's direct ancestor as well as all of its descendant classes.

- https://www.sanfoundry.com/java-mcqs-object-class/
  Object class is superclass of every class in Java.

  The Object class, which is stored in the java.lang package, is the ultimate superclass of all Java classes (except for Object).

- https://csawesome.runestone.academy/runestone/books/published/csawesome/Unit9-Inheritance/topic-9-7-Object.html
  The Object class is the superclass of all other classes in Java and a part of the built-in java.lang package.

  superclasses(ClassName) displays the names of all visible superclasses of the MATLAB® class with the name ClassName. Visible classes have a Hidden attribute value of false (the default).
  ...
  superclasses(obj) displays the names of all visible superclasses of object obj, where obj is an instance of a MATLAB class. obj can be either a scalar object or an array of objects.

  CommunityMember is the direct superclass of Employee, Student and Alumnus, and is an indirect superclass of all the other classes in the diagram.

When it comes to subclass, even the direct ones are not unique, which means that directness cannot be distinguished by referring to singularity, and it is this fact that is making it clearer that the word does not imply directness. The words subclasses and descendants mean the same thing. We cannot assign different meanings to them.

So my suggestion is that, as pointed out by mame, avoid the method name "subclasses", and as I previously suggested, deprecate "superclass" in favor of something else.

From compatibility and uniformity with other methods involving (in)directness such as Module#instance_methods, I now think it is a good idea to have the directness parameter for Class#ancestors and Class#descendants to be positional (and optional), with truthy value meaning "including indirect", and falsy meaning "only direct", default being truthy.

class C; end
class D < C; end
class F < C; end
class E < D; end

E.ancestors # => [E, D, C, Object, Kernel, BasicObject]
E.ancestors(false) # => D
C.descendants # => [C, D, F, E]
C.descendants(false) # => [D, F]

Class#subclasses sounds OK. Accepted.
Add Class#subclasses

Implements [Feature #18273]

Returns an array containing the receiver’s direct subclasses without singleton classes.

---

#14 - 01/24/2022 12:15 AM - fxn (Xavier Noria)

In my view, this method does not play well with object lifetime, semantically.

Take for example:

```ruby
C = Class.new
-> { Class.new(C); 1 }.call
pp C.subclasses #=> [#<Class:0x000000010c8f5a90>]
```

On line 3, the subclass of C does not exist anymore conceptually, but Class#subclasses depends on GC actually deleting the object. That is non-deterministic behaviour.

This affects reloading in Zeitwerk, which is implemented via remove_const. If the program using Zeitwerk is correct, after remove_const there are no strong references to the unloaded object. But if you define a subclass again on reload, C.subclasses may yield two objects instead of one.

#15 - 01/24/2022 12:32 AM - fxn (Xavier Noria)

Said in a different way:

```ruby
C = Class.new
-> { Class.new(C); 1 }.call
pp C.subclasses
pp C.subclasses
```

If GC kicks in between lines 3 and 4, you may get different results. In my opinion, this non-determinism is not good for an API like this one.

---

#16 - 01/24/2022 08:26 AM - byroot (Jean Boussier)

In my view, this method does not play well with object lifetime, semantically.

This is perfectly fine, Ruby isn't concerned about whether a Class is still present in the constant table or not (or if it ever was). That's also how the old Active Support implementation based on ObjectSpace.each_object behaved.

That's why DescendantTracker keeps a WeakMap of the unregistered classes to give the illusion in reload mode.

If GC kicks in between lines 3 and 4, you may get different results. In my opinion, this non-determinism is not good for an API like this one.

That too was considered and is fine.

#17 - 01/24/2022 10:01 AM - fxn (Xavier Noria)

This is perfectly fine, Ruby isn't concerned about whether a Class is still present in the constant table or not

I am talking about object lifetimes, my example does not store the class object in a constant.

In my view, this API is not consistent with the Ruby model. In the Ruby model, you have class objects, and they have superclasses. Instances hold a strong reference to their classes.

Whether you store those objects in variables or constants, or not at all (as in the example above), is orthogonal to that model.

So, in this model, Class#subclasses cannot be deterministic.
I don't think I understand what your issue is. subclasses returns all the live object (not yet GCed) that are direct descendants.

In my Ruby model it works as expected.

Do you mean that in your Ruby model, objects are "gced" as soon as they are unreachable? How do weakref / weakmap work in your model then?

What I mean is that Class#subclasses is effectively a cache that does not have proper invalidation.

The red flag for me is the example above, in a linear program:

```ruby
C.subclasses
C.subclasses
```

should return the same. It does not, so in my view this cache is not good enough to be in the language itself.

To me it behave just like weak references do:

```ruby
weakmap = ObjectSpace::WeakMap.new
weakmap[Object.new] = Object.new
weakmap.keys #=> [#<Object:0x00007ff6329049c8>]
# GC.start
weakmap.keys #=> []
```

I really see no problem with that.

That is, since objects are not gced as soon as they are unreachable, this implementation of Class#subclasses is non-deterministic, and therefore not good enough to be in the language itself. That is my point of view :).

Sure, and you are entitled to it, I just don't understand how you reconcile it with the existence of weak references.

Sure, and you are entitled to it, I just don't understand how you reconcile it with the existence of weak references.

I believe our legitimate differences in viewpoints are:

1. You say it works as expected, based on weak refs.
2. I say, true, but a posteriori, usage is not deterministic for the user, and therefore not a good core API.

The documentation should say: Returns the subclasses that have not been yet GCed. Multiple calls may yield different values even if you don't touch the hierarchy in between them.

To me, having to write such documentation is a suggesting this feature is dubious.

To me, this feature makes sense in a language where subclasses cannot disappear once created. In Ruby, they are regular objects. This is the root mismatch between the core language and this API.

fxn (Xavier Noria) Isn't it an inherent property of a dynamic language, that whatever you can introspect about the code structure, can change in runtime? So the methods truthfully reflecting the current state of the structure are perfectly sane?

I'd argue it is not a qualitative difference from a change of the subclasses because something was defined in runtime. I can see how it can be said to be different (because in the case of subclass vanishing, nothing on user level of the program can be pointed as a code that led to the change), but is this difference practicarly only affects "theoretical clarity"?

I assume that as a developer of the Zeitwerk you maybe can provide some insight on practical consequences?
For example, take this:

```ruby
C = Class.new
Object.subclasses #=> (1)
```

```ruby
D = Class.new
Object.subclasses #=> (2)
```

For the user, it makes sense that (1) includes C, and (2) includes C and D. They are guaranteed to be there, because Object has a strong reference to them via its constants table, and this fact belongs to the public API of Ruby. (My points are not about constants, but in that example, in order to say that (2) has C and D I need to say there's a strong reference somewhere.)

If you later on add a mixin:

```ruby
C.include!(M)
```

Of course, C.ancestors is supposed to reflect that change.

What would be strange is that

```ruby
C.ancestors
C.ancestors
```

could yield different results if you did not change the ancestor chain in between.

That is, you didn't do anything to change that collection.

but is this difference practical or only affects "theoretical clarity"

It's not marble tower theoretical clarity, it is just that the API of the language has to agree with the language model.

There are APIs that conceptually are about live objects, like ObjectSpace. But Class#subclasses is not at that level, conceptually. Class#subclasses is at the level of Module#ancestors, in my view.

I assume that as a developer of the Zeitwerk you maybe can provide some insight on practical consequences?

In my view, the API introduces a broken window, a method that behaves in a non-deterministic way. Broken windows leak.

#27 - 01/24/2022 12:19 PM - fxn (Xavier Noria)

Hmm, now that I think, maybe you don't see how this affects Zeitwerk directly.

In Zeitwerk, "reloading" means issuing remove_const calls to unload, and execute Module#autoload calls to be ready to load again fresh definitions. If the program is correct, those objects are unreachable, and therefore gone for all practical purposes (except for alive-APIs like ObjectSpace).

So, in a reload something like this happens:

```ruby
C = Class.new
D = Class.new(C)

# Reload.
Object.send(:remove_const, :D)
D = Class.new(C)

C.subclasses #=> [D, D]
```

In which situation would you find that? Well, in any situation where Class#subclasses is used in the context of reloading.

However, I add this comment to further clarify that specific question about Zeitwerk.

My remarks are not about Zeitwerk, they are more fundamental, at the language/core API level.

#28 - 01/24/2022 01:26 PM - zverok (Victor Shepelev)

TBH, I still fail to see the point of how the API clearly reflecting the reality should be considered a "broken window".

Two subsequent calls returning different results without user's code affecting these results is not that uncommon: if you'll try to introspect the current state/list of threads, or fibers, or ractors; if you try to look at files and directories while something else changed them; well, if you just Time.now two times in a row...

From my perspective (being obsessed with docs and teaching people) it is rather fortunate that with Class#subclasses we'll have a clear and obvious place to write: yes, subclasses may vanish, please beware and remember; it is an important thing to say explicitly!
I don't think "that would be strange" is a strong enough argument here, if it clearly reflects what's really going on.

#29 - 01/24/2022 01:38 PM - fxn (Xavier Noria)

zverok (Victor Shepelev) one thing is "subclasses can vanish", because the language is dynamic (like ancestors can change), and a different thing is that in this program:

```ruby
Class.new
Object.subclasses
```

you cannot tell me what is the 2nd line going to return.

Personally, I find this non-deterministic behavior unacceptable for an API of this level.

#30 - 01/24/2022 01:48 PM - zverok (Victor Shepelev)

We are going in circles here :( It is "non-deterministic" all right, but also reflects the reality.

It is more or less the same as with

```ruby
Thread.new {}
run
Thread.list
```

...isn't it? We can't predict what the 2nd line is going to return.

Should we argue that Time.now value can't be predicted and therefore non-deterministic, and therefore a bad API?

(I am not trolling, I am trying to see what lies in the gap between just repeating that it is non-deterministic and therefore not acceptable from one side, and it is how it really is and therefore totally sane API from another.)

#31 - 01/24/2022 01:55 PM - fxn (Xavier Noria)

Sure!

I see your point, it works as it is supposed to be.

My point is, Class#subclasses should be deterministic like Module#ancestors. In the sense that, as a programmer, if I do not change the collection, it should stay the same.

Since that is not possible (at least in this implementation), in my logic, that means Class#subclasses should not be added to Ruby.

#32 - 01/24/2022 02:11 PM - fxn (Xavier Noria)

In the sense that, as a programmer, if I do not change the collection, it should stay the same.

Of course, removing a subclass is ill-defined and that is the root problem I see here. And in Ruby that makes sense, because the link only went upwards till 3.1.

On the other hand, Module#ancestors can be a linearized cache because there is no API or way to remove ancestors.

Ruby doesn't have API/semantics to provide a similar Class#subclasses.

#33 - 01/24/2022 02:52 PM - zverok (Victor Shepelev)

fxn (Xavier Noria) You know what?

I think I actually got your point:

- Class#ancestors represent a "really existing thing", that is, objective ancestors chain. E.g. it is an API to an internal structures, helping to make sense of them;
- Class#subclasses look the same way, therefore implying it is also an API to internally existing structures (for example, to each class keeping track of its descendants because it is a part of an object model)—but this implication is completely untrue! The #subclasses is a dynamic thing—and in more formal language, maybe should've at least been called #enumerate_subclasses (implying by a verb form that it will do something, not just expose internally existing reality)

Is that it?

#34 - 01/24/2022 02:52 PM - fxn (Xavier Noria)

zverok (Victor Shepelev) I think our disagreement is clear. Perhaps your position is also the one of byroot (Jean Boussier).

To me, Class#subclasses cannot be defined because in Ruby there's no such concept as "removing a subclass". And this is an API for which I would
expect determinism.

Of course, Time.now or ObjectSpace are different kinds of APIs.

I believe we'll need to agree on disagreeing :).

#35 - 01/24/2022 02:53 PM - fxn (Xavier Noria)
zverok (Victor Shepelev) Sorry, we were typing at the same time.

Exactly! That is my point of view!

#36 - 01/24/2022 03:45 PM - fxn (Xavier Noria)

More than the existence of a real collection (which is a private implementation detail), it is the existence of a clear collection API.

In ancestors, you have API for adding, and no API for removing or updating in-place. So, the programmer can always know what is Module#ancestors just by looking at the code.

Ruby has API for subclassing. And you have strong pointer from class to superclass, that is all. That is all you got.

So, the concept of "subclasses" at any given point is fuzzy, because the virtual collection of subclasses is ill-defined, because adding is well-defined, but removing is not.

This cache is therefore using the only tool it has at hand, alive objects and garbage collection. But to me, the resulting semantics are as fuzzy in consequence, for something that shouldn't be, in my opinion.

Look at the docs, you have the happy path using constants. These docs should be more complete, warning the user, and introducing garbage collection in their description. At this point, you (generic you :) may agree the existence of these docs indicates the method should not exist.

#37 - 01/24/2022 10:40 PM - zverok (Victor Shepelev)

I see, and tend to agree—to some extent, at least!

Not sure how much can be done here—some amount of "semantically not really clear, yet convenient" methods trickle from Rails/ActiveSupport. (My personal "favorite" is Array#intersect and related methods, making Array API pretending it is set instead of enhancing Set and making it more popular.)

I could at least do this: https://github.com/ruby/ruby/pull/5480, WDYT?