Ruby - Feature #19520

Support for `Module.new(name)` and `Class.new(superclass, name)`.

03/09/2023 08:50 AM - ioquatix (Samuel Williams)

Status:	Rejected
Priority:	Normal
Assignee:	
Target version:	

Description

See https://bugs.ruby-lang.org/issues/19450 for previous discussion and motivation.

This proposal introduces the name parameter to Class.new and Module.new:

```
Class.new(superclass, name)
Module.new(name)
```

As a slight change, we could use keyword arguments instead.

Example usage

The current Ruby test suite has code which shows the usefulness of this new method:

```
  def labeled_module(name, &block)
    Module.new do
      singleton_class.class_eval {
        define_method(:to_s) {name}
        alias inspect to_s
        alias name to_s
      }
      class_eval(&block) if block
    end
  end
  module_function :labeled_module

  def labeled_class(name, superclass = Object, &block)
    Class.new(superclass) do
      singleton_class.class_eval {
        define_method(:to_s) {name}
        alias inspect to_s
        alias name to_s
      }
      class_eval(&block) if block
    end
  end
  module_function :labeled_class
```

The updated code would look like this:

```
  def labeled_module(name, &block)
    Module.new(name, &block)
  end

  def labeled_class(name, superclass = Object, &block)
    Class.new(superclass, name, &block)
  end
  module_function :labeled_class
```

Related issues:

Related to Ruby - Feature #19521: Support for `Module#name=` and `Class#name=`.	**Closed**
Related to Ruby - Feature #19450: Is there an official way to set a class nam...	**Closed**
Related to Ruby - Feature #18285: NoMethodError#message uses a lot of CPU/is ...	**Closed**

History

#1 - 03/09/2023 08:55 AM - ioquatix (Samuel Williams)

- Description updated

#2 - 03/09/2023 04:23 PM - Eregon (Benoit Daloze)

- Related to Feature #19521: Support for `Module#name=` and `Class#name=`. added

#3 - 03/09/2023 04:24 PM - Eregon (Benoit Daloze)

- Related to Feature #19450: Is there an official way to set a class name without setting a constant? added

#4 - 03/09/2023 04:32 PM - Eregon (Benoit Daloze)

In general I'm against this functionality as explained in https://bugs.ruby-lang.org/issues/19450#note-14.
It can lead to confusion and lies about the program state, e.g., pretend there is Foo::Bar when there isn't, even when remove_const/const_set are not used.

I think what's missing in the description is why can't you just assign those modules to a constant?
That is the way of giving a name to a Module, and it has a huge advantage: Module#name tells you how to refer that module.
If e.g. someone does Module.new("Foo::Bar") and someone else debugs some test, sees Foo::Bar in the output but they can't even p Foo::Bar they will become crazy.
And I think that's a clear illustration why this feature is harmful.
(yes, there is remove_const/const_set but those are extremely rarely used and still at least the module was actually in that constant at some point vs never)

#5 - 03/09/2023 06:24 PM - ufuk (Ufuk Kayserilioglu)

I am in complete agreement with @Eregon (Benoit Daloze) and I would also be against Class#name and Module#name method behaviour changing in this way.

I am one of the maintainers of the Tapioca gem, which, among other things, does a lot of runtime introspection to discover constants, methods, mixins defined by gem so that it can generate interface files for them (RBI files for today). Currently, the only protection that we have for finding the real name of a constant is the ability to rebind the Module#name method to the constant in question. If the result is nil, then the constant is anonymous, otherwise we are guaranteed that the name maps to the constant (since we reach the constant via "a" name in the first place).

Given that context, if Module#name starts returning any arbitrary string that does not in any form map to the actual name of the constant, there is no alternative method that Tapioca can use to get that information.

Having said all of this, I understand that there are some use-cases where it might be good to be able to display a more friendly name for a constant for the user (in this case, a developer as the user). I would argue that we don't have to mess with the name method to get that benefit. The use-case presented can equally be achieved by adding something like a Module#display_name method (and even a setter for it) which to_s and inspect could use when name is nil. In my opinion, doing so would achieve what the use-case is trying to do, without changing any existing behaviour.

#6 - 03/09/2023 08:30 PM - ioquatix (Samuel Williams)

@ufuk (Ufuk Kayserilioglu) It's already the case that it's trivial to override Class#name and have it return something other than a constant.

```
c = Class.new

def c.name = "Hello World"

c.name
=> "Hello World"
```

The problem is Ruby does not use this internally, so you end up with inconsistent output:

```
instance = c.new
=> #<#<Class:0x00007f7f11aee960>:0x00007f7f11a16180>

instance.class.name
"hello world"
```

One option would be to fix this bug so that Class#name is used in this context. However, since class names are cached as part of the class path, overriding this method can be problematic. It's best to set it once when the class/module is created.

Adding a display name might work, but it's the same problem as outlined above.

Probably the real way to solve this issue would be to know whether a class or module is anonymous, e.g. Class#anonymous? so you could exclude them from any type checking or handle them differently.

#7 - 03/09/2023 08:37 PM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-6:

> @ufuk (Ufuk Kayserilioglu) It's already the case that it's trivial to override Class#name and have it return something other than a constant.

As @ufuk (Ufuk Kayserilioglu) said in his comment, he uses Module.instance_method(:name).bind_call(mod), and that's unaffected by def c.name but it would be by this new feature.

#8 - 03/09/2023 08:46 PM - Eregon (Benoit Daloze)

Maybe I should say it more plainly: I believe adding this feature would be a clear language design mistake, nothing less.
It "breaks" the language for only a very minor benefit.

Usage in tests is not convincing to me, one could just assign those to e.g. Testing::Foo or so.

#9 - 03/09/2023 10:17 PM - ioquatix (Samuel Williams)

> Usage in tests is not convincing to me, one could just assign those to e.g. Testing::Foo or so.

Your suggestion doesn't work well in practice, e.g.
https://github.com/rspec/rspec-core/blob/d722da4a175f0347e4be1ba16c0eb763de48f07c/lib/rspec/core/example_group.rb#L895-L903

> uses Module.instance_method(:name).bind_call(mod), and that's unaffected by def c.name but it would be by this new feature.

This already appears broken to me:

```
m = Module.new
=> #<Module:0x00007f4ea0f5d160>

m.class_eval("class Bar;end")
bar = m::Bar
=> #<Module:0x00007f4ea0f5d160>::Bar

Module.instance_method(:name).bind_call(bar)
=> "#<Module:0x00007f4ea0f5d160>::Bar"
```

The name given here is not valid.

One way to resolve the above issue is to expose when a name is a valid permanent global name, or when it's a temporary anonymous name. e.g. m.anonymous? at least allows you to rule out types which are not expected to be resolved.

Unfortunately as I already demonstrated, even permanent global names can be fake/resolvable and it is trivial to construct such a case, so !m.anonymous? does not mean that eval(m.name) will result in anything useful.

#10 - 03/09/2023 10:38 PM - ioquatix (Samuel Williams)

Just thinking out loud, maybe we need to stop assuming strings (that can be fooled) represent class paths/namespaces.

```
m = Module.new
module M
end

m.class_eval("class N; end")
M.class_eval("class N; end")

m.anonymous? # true
M.anonymous? # false

m.namespace # [m]
M.namespace # [M]

m::N.anonymous? # true
M::N.anonymous? # false

m::N.namespace # [m, N]
M::N.namespace # [M, N]
```

In addition, maybe remove_const should correctly update the class names. However, it would require a complete traversal of all constants to clear out their names (since they are potentially cached).

#11 - 03/09/2023 11:28 PM - ufuk (Ufuk Kayserilioglu)

ioquatix (Samuel Williams) wrote in #note-6:

> The problem is Ruby does not use this internally
> ...
> Adding a display name might work, but it's the same problem as outlined above.

Yes, that has the same problem, but my suggestion was to make to_s and inspect respect the value of display_name if it was set, so that it would not have the same problem outlined above.

I still don't understand why we would want to risk breaking so many assumptions existing code has already made about how constants and constant names work (regardless of the soundness of it) to add a feature that is a nice-to-have. Especially when there is a backward-compatible way to provide such a feature that is being suggested.

#12 - 03/10/2023 12:55 AM - ioquatix (Samuel Williams)

> Yes, that has the same problem, but my suggestion was to make to_s and inspect respect the value of display_name if it was set, so that it would not have the same problem outlined above.

We could certainly explore this option, but I'm sure the change is more extensive.

> I still don't understand why we would want to risk breaking so many assumptions existing code has already made about how constants and constant names work

I don't understand this argument at all. The assumptions are already broken by the trivial examples already given.

#13 - 03/10/2023 03:24 PM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-12:

> I don't understand this argument at all. The assumptions are already broken by the trivial examples already given.

They are not or very rarely broken in practice, that matters a lot.

> => "#Module:0x00007f4ea0f5d160::Bar"
> The name given here is not valid.

That's fine, this is easy to recognize there an anonymous module in there, there is no confusion.
Unlike a Foo::Bar name with this feature and yet never having a Foo constant defined on Object.

#14 - 03/10/2023 04:23 PM - ufuk (Ufuk Kayserilioglu)

ioquatix (Samuel Williams) wrote in #note-12:

> I don't understand this argument at all. The assumptions are already broken by the trivial examples already given.

Let me try to explain: I think we are focusing too much on the broken case and not enough on the non-broken cases. Under normal circumstances constants tell you the names they are bound to when asked via the Module#name method. Certain classes/modules override that to change what is displayed, but one can always get the name that Ruby knows them by through the Module.instance_method(:name).bind_call(mod) call.

If we allow what is returned by Module#name to be any arbitrary string that the user chooses (for example, the original request had examples of Module(/foo/bar/baz.rb) or similar), then there won't be a way to ever get the actual name of that module, regardless of it was anonymous or named. It would forever return the name that the user specified. I think this is the part of the proposal that @Eregon (Benoit Daloze) and me are particularly against, that it would break the current meaning and operation of Module#name, regardless of if the semantics of what is returned by it today is broken in some cases or not.

Again, I understand the need to give better display names for Modules, I have needed to use that myself at various points. That's why I am suggesting to maybe think about implementing this in a way that doesn't mess with the current Module#name method. I hope this makes the argument a little bit more clear.

#15 - 03/10/2023 09:02 PM - ioquatix (Samuel Williams)

> there won't be a way to ever get the actual name of that module, regardless of it was anonymous or named. It would forever return the name that the user specified.

That's not how the proposal here works. One you assign a permanent name, it replaces any fake/temporary name.

```
m = Module.new("fake")
=> fake

m.name
=> "fake"

M = m
=> M

m.name
=> "M"

Module.instance_method(:name).bind_call(m)
=> "M"
```

> That's fine, this is easy to recognize there an anonymous module in there, there is no confusion.
> Unlike a Foo::Bar name with this feature and yet never having a Foo constant defined on Object.

Don't live code reloading systems work exactly like this, creating a whole bunch of "orphaned" constants?

#16 - 03/10/2023 09:04 PM - ioquatix (Samuel Williams)

Yep, Zeitwerk does exactly that:

> When unloading, Zeitwerk issues Module#remove_const calls. Classes and modules are no longer reachable through their constants, and on_unload callbacks are executed right before those calls.

https://github.com/fxn/zeitwerk#technical-details

So, I'm really not convinced by the argument that "They are not or very rarely broken in practice, that matters a lot." as every Rails app potentially does this on a regular basis.

#17 - 03/14/2023 08:48 PM - Eregon (Benoit Daloze)

I had a call with @ioquatix (Samuel Williams), trying to explain the importance of the name that Ruby shows us, for example:
undefined method 'zzz' for #<Foo::Bar:0x00007efc38711fc0> (NoMethodError)
people will of course expect that Foo::Bar refers to the class of the object.
For instance maybe they want to create a new instance of Foo::Bar, so writing Foo::Bar in the code must refer to that class.
Yes, it's possible to break it with remove_const/const_set, but in practice this almost never happens, and if it does I would consider it a very serious bug of whatever uses remove_const/const_set to break it.

And the same for Zeitwerk, while I guess it's possible to break the constant path<->module mapping there with reloading e.g. maybe by storing an old Class in a global variable and doing that only once per process (e.g. only if the global is unset), it's just extremely uncommon.
I don't think Rails users get this problem often at all (e.g. we'd see more issues on rails or zeitwerk if they did).

One thing that could help is for this name to be visually different than a regular constant path, so e.g. it cannot start with an uppercase letter (as said in https://bugs.ruby-lang.org/issues/19450#note-17), and probably start with some symbol to make it even more obvious.
(Actually C code can define lowercase constants, so just not uppercase first letter is not enough, e.g. IO::generic_writable seen from StringIO.ancestors).

> One you assign a permanent name, it replaces any fake/temporary name.

That wasn't clear to me and the description doesn't seem to mention it.
It means there is yet another state of naming for modules in addition to the existing fully-anonymous (#<Module:0x0123>), knows its own name but not its lexical parent/nesting (#<Module:0x0123>::A) and fully-named (A::B).
So that's some extra complexity both for the user and for implementations.
I thought the "fake name" would be considered fully-named. IIRC naming of constants under a module is only done if a module is fully-named, and I think that's the feature you want here.

What I didn't have time to discuss on the call is what's the use-case for this besides tests which want to label anonymous modules/classes to make it easier to debug them.
@ioquatix (Samuel Williams) mentioned his own reloading logic, but I'm not sure how that works, e.g., how do you refer to another model if not all models have a fully-named constant path?

#18 - 03/14/2023 08:58 PM - Eregon (Benoit Daloze)

Another thought: maybe a much simpler way to solve most of these use-cases is adding Module#source_location, which is the [file, line] at which the Module was created.
That could also work for anonymous modules, they could capture at which file, line Module.new was called.

That I think would remove the need to label modules/classes in tests, since the file line would then refer to where it was created, and at that file:line there is likely a local variable which makes it clear what's the role of this anonymous module/class (e.g. parent = Class.new).

The anonymous reloading use-case would also be helped by having that file path attached to anonymous modules.

We could then maybe show this file:line automatically for Module#inspect and in exception messages if the module is anonymous.
If the module is named it's likely of much lower value so there probably not change anything, but anyone could still call Module#source_location to find more about the module/class at hand.

#19 - 03/14/2023 10:08 PM - ioquatix (Samuel Williams)

> I had a call with @ioquatix (Samuel Williams) (Samuel Williams), trying to explain the importance of the name that Ruby shows us, for example:
> undefined method 'zzz' for #<Foo::Bar:0x00007efc38711fc0> (NoMethodError)
> people will of course expect that Foo::Bar refers to the class of the object.

As demonstrated, if users want to do this, it's already possible.

```
Foo = Module.new
#=> Foo

Foo.class_eval("class Bar;end")

bar = Foo::Bar.new
#=> #<Foo::Bar:0x000055c0ae3c1e68>

Object.send(:remove_const, Foo.name)
#=> Foo

bar.zzz
#=> undefined method `zzz' for #<Foo::Bar:0x000055c0ae3c1e68> (NoMethodError)

Foo::Bar
#=> uninitialized constant Foo (NameError)
```

> (Actually C code can define lowercase constants, so just not uppercase first letter is not enough, e.g. IO::generic_writable seen from StringIO.ancestors).

It's not possible to access such constants using the normal constant lookup:

```
StringIO.ancestors
#=> [StringIO, IO::generic_writable, IO::generic_readable, Enumerable, Data, Object, PP::ObjectMixin, Kernel, BasicObject]

IO::generic_writable
#=> undefined method `generic_writable' for IO:Class (NoMethodError)
```

But I also don't think that matters much for this proposal anyway.

> "One you assign a permanent name, it replaces any fake/temporary name." -> That wasn't clear to me and the description doesn't seem to mention it. So that's some extra complexity both for the user and for implementations.

This is how Ruby already works internally, this is not part of my PR. This is how anonymous modules already work.
https://github.com/ruby/ruby/blob/868f03cce1a2d7a4df9b03b8338e3af4c69041d0/internal/class.h#L214 is the implementation that already exists. This is a way to cache the class names. It is used to inform child modules that the class name won't change in the future.

```
m = Module.new # internally, the name is not permanent.
M = m # now it becomes permanent and any child constants in m should update their names to be permanent too.
```

This is part of the reason why one could consider remove_const to be buggy.

> And the same for Zeitwerk, while I guess it's possible to break the constant path<->module mapping there with reloading e.g. maybe by storing an old > Class in a global variable and doing that only once per process (e.g. only if the global is unset), it's just extremely uncommon.

I don't think it's uncommon to cache instances of a class in some global mapping. Does Zeitwerk reload the entire namespace or just ones that changed? I don't know enough about it. @fxn any thoughts on how this is handled? I'm assuming Zeitwerk reloading can create orphaned constants (i.e. it's calling remove_const).

> One thing that could help is for this name to be visually different than a regular constant path, so e.g. it cannot start with an uppercase letter > (as said in https://bugs.ruby-lang.org/issues/19450#note-17), and probably start with some symbol to make it even more obvious.
> (Actually C code can define lowercase constants, so just not uppercase first letter is not enough, e.g. IO::generic_writable seen from StringIO.ancestors).

This is the current convention for anonymous modules, to a certain extent. If one overrides Class#name, that's no longer true.

> What I didn't have time to discuss on the call is what's the use-case for this besides tests which want to label anonymous modules/classes to make it easier to debug them.

Well, I think this is already explained multiple times, i.e. the examples I gave + Ruby's own CI.

> Another thought: maybe a much simpler way to solve most of these use-cases is adding Module#source_location, which is the [file, line] at which the Module was created.
> That could also work for anonymous modules, they could capture at which file, line Module.new was called.

Do I think this is potentially a good idea? Yes.

Does that work for all the example use cases I gave? No.

I think what might make more sense is:

- Introducing anonymous? as a predicate for whether a given class/module is itself rooted in the global namespace or not.
- (Consider) changing remove_const to correctly convert class/module back to anonymous.
- (Consider) decorating class names when they are anonymous, e.g.

```
module Module
  def to_s
    if anonymous?
      "\##{@name}"
    else
      @name
    end
  end
end
```

#20 - 03/14/2023 10:19 PM - ioquatix (Samuel Williams)

Also, it looks like Zeitwerk absolutely can reload code and create confusion, and in addition, appending file:line information would not avoid that confusion.

https://github.com/ioquatix/zeitwerk-reload

Adjusting remove_const to turn the previous Foo::Bar instance into an anonymous one would allow ups to present it differently (e.g. #Foo::Bar or some other appropriate syntax). But that's a separate issue of this PR, and I we'd need to be careful about how we implement such a feature.

#21 - 03/15/2023 11:03 AM - Eregon (Benoit Daloze)

> As demonstrated, if users want to do this, it's already possible.

Yes, and it just doesn't matter much in practice because it almost never or never happens in real code.
The point is if the name of a Module doesn't correspond to how to access it, then we simply cannot program in Ruby anymore because we can't use constant lookup anymore, e.g. we can't MyClass.new anymore, because the mapping of MyClass<->that Class instance is no longer as expected.
Anything that breaks this expectation that every Rubyist relies on is a serious bug.
Right now, bad usages of remove_const/const_set can do that, such bad usages should be fixed but OTOH as far as I see they are mostly from unrealistic cases.
Module.new("Fake") would make this all too easy, and I would not be surprised many people would accidentally use it like that without realizing it breaks the expectations above significantly.
So it's not that people would break these expectations on purpose, if they want to do that they can already do that (but nobody does).
It's that this new API would let people think Module.new("Fake") is a good idea when it's a terrible terrible one (there is no constant Fake with that, or worse a constant Fake pointing to a different module).

> This is how Ruby already works internally, this is not part of my PR. This is how anonymous modules already work.

That much I know. But nowhere in this description there is anything about a new state in between where the fake name is there but it can still be fully named after.
I thought and I think everyone else reading that description thought that you would consider the "fake" name the final name for a Module.

> If one overrides Class#name, that's no longer true.

Incorrect, as already said before, one cannot change the original Module#name, and exception messages from Ruby use that, not any override/monkey-patch.

One could of course override some inspect of some class and then it wouldn't show that object's class's constant path anymore, but that's then clearly the fault of that class' override.

#22 - 03/15/2023 11:06 AM - Eregon (Benoit Daloze)

> Also, it looks like Zeitwerk absolutely can reload code and create confusion

This is again an unrealistic completely made up example and I would think an invalid usage of Zeitwerk.
For real code, you don't reload in the middle of a file, so then e.g. local variables, constants, etc, cannot be used to leak old modules/classes.

It seems the part of the argument you are missing is what I said in my previous reply: So it's not that people ...

#23 - 03/15/2023 11:10 AM - Eregon (Benoit Daloze)

In fact, implementing labeled_module/labeled_class like you showed above would again break these expectations, illustrating how unsafe is this API, and it's clear many people writing such tests wouldn't realize this problem (they might not even know the labeled_module implementation change).
E.g. there are some m = labeled_module("M") and some c0 = EnvUtil.labeled_class('C0') do, but of course no constant M or C0.

#24 - 03/15/2023 11:30 PM - fxn (Xavier Noria)

> I don't think it's uncommon to cache instances of a class in some global mapping. Does Zeitwerk reload the entire namespace or just ones that changed? I don't know enough about it. @fxn (Xavier Noria) any thoughts on how this is handled?

Storing a class or module object whose constant is reloadable in a non-reloadable place is considered to be an error in Zeitwerk-based projects. You should not do that, period. It is not an error in the sense that you get an exception, but it is a logical error and we do not need to waste time thinking about such scenario. Users are not supposed to do this, and if they do, the consequences are out of scope, unsupported.

So, for example, let's say Foo is a reloadable module, and you include Foo in some place that is not reloadable, like ActiveRecord::Base. That is wrong, you'll have a stale module object in the ancestor chain of ActiveRecord::Base on reload that won't reflect edits to Foo. This is documented in (5) here.

> I'm assuming Zeitwerk reloading can create orphaned constants (i.e. it's calling remove_const).

Not really, let me explain.

If Foo is a top-level constant without autoloaded constants below, it is remove_const'ed. Fine.

If Admin is a top-level constant and the module it stores acts as a namespace so that we have Admin::UsersController, and Admin::Payments, Zeitwerk removes all of the constants. That is, it removes :UsersController and :Payments from the module stored in Admin, and then removes :Admin from Object.

In theory if the constants are used correctly, removing :Admin would be enough because everything else would not be reachable and eventually GCed. But just in case a user has a stale object cached somewhere, I want to make sure that at least its constants are gone.

Zeitwerk puts constraints on what projects can do to make autoloading/reloading solvable.

#25 - 03/15/2023 11:40 PM - fxn (Xavier Noria)

To be clear, Zeitwerk works only with constants. When you load foo.rb, the constant :Foo has to exist in Object, Zeitwerk does not care if the classes or modules went through an initial anonymous period. It also does not care if the file had a Foo constant reference or you did a const_set in the expected receiver. After loading, Foo has to exist. Otherwise, you get an exception.

#26 - 03/15/2023 11:50 PM - fxn (Xavier Noria)

This is a long thread, I was not aware of it.

Let me say that nobody can assume from the name of a class or module that the corresponding constant exists. Class and module objects get their name when they are first assigned to a constant, and the class and module keywords are in part constant assignments. We all know this.

We also know the coupling ends there. These entities ar highly decoupled in Ruby by design. I can have C = Class.new; c = C; remove_const :C, and the class in c is no longer reachable through the constant after its name. If a Ruby programmer expects that, they have to revise that expectaction because it is just baseless.

#27 - 03/16/2023 01:39 AM - Dan0042 (Daniel DeLorme)

Eregon (Benoit Daloze) wrote in #note-17:

> IIRC naming of constants under a module is only done if a module is fully-named

Actually that's incorrect since 3.0

```
a = Module.new
a::B = Module.new
a::B.name #=> "#<Module:0x00007f6c2d474078>::B"
```

Since a::B.name is neither nil nor resolvable to a constant, what does it matter if it's Controller(path/to/file.rb)::B instead of #<Module:0x00007f6c2d474078>::B ?
The "fake name" doesn't make the value of a::B.name any more or less unresolvable than it already was, but it does make it infinitely more debuggable.

#28 - 03/16/2023 11:48 AM - Eregon (Benoit Daloze)

fxn (Xavier Noria) wrote in #note-26:

> We also know the coupling ends there. These entities ar highly decoupled in Ruby by design. I can have C = Class.new; c = C; remove_const :C, and the class in c is no longer reachable through the constant after its name. If a Ruby programmer expects that, they have to revise that expectaction because it is just baseless.

I'm afraid you missed the point. Maybe https://bugs.ruby-lang.org/issues/19520#note-17 and https://bugs.ruby-lang.org/issues/19520#note-21 help to make it clearer.

Every Ruby programmer when they see e.g. undefined method 'zzz' for #<Foo::Bar:0x00007efc38711fc0> (NoMethodError) expects that Foo::Bar in code would refer to the class of that object.
Yes, it's not a guarantee. But it holds in practice 99.99+%. If it doesn't hold then it's a very serious bug for whatever breaks it, just like the cases you mentioned are unsupported with Zeitwerk, here it would be unsupported for developers sanity.

Hence Module.new(name) is harmful because it will make people non-consciously break that all the time.

#29 - 03/16/2023 11:52 AM - Eregon (Benoit Daloze)

@Dan0042 Right, the rule is not as simple, indeed.
Of course a::B is not a fully-named constant here, so this is "temporary name" for a::B, which is what I was thinking about.

> Since a::B.name is neither nil nor resolvable to a constant, what does it matter if it's Controller(path/to/file.rb)::B instead of #<Module:0x00007f6c2d474078>::B ?
> The "fake name" doesn't make the value of a::B.name any more or less unresolvable than it already was, but it does make it infinitely more debuggable.

This to me makes it look like the first module is accessible via constant path Controller. But it probably isn't, it's a lie and it's confusing.
I think there is no valid reason to have anonymous modules/classes used as namespaces in non-test code.
Those modules/classes used as namespaces should be named, otherwise no other part of the code can even refer to them, which would be too limiting for any real purpose.
(well, they might through const_missing but it's nonsense to occur that overhead just to not name them)

#30 - 03/16/2023 11:57 AM - Eregon (Benoit Daloze)

@ioquatix (Samuel Williams) I think you need to explain the design of your reloading and usages of anonymous modules and why it's so important that we should modify Ruby for it.
Also maybe why Zeitwerk is not enough for you.

For labeled_module and labeled_class as I showed in https://bugs.ruby-lang.org/issues/19520#note-23 it's harmful and not a good usage of this new API.
Or you would need to make them visually impossible to clash with real constant names at least.
I think for that usage, Module#source_location is a much simpler and cleaner solution (and it even doesn't require changes to existing code).

#31 - 04/05/2023 10:56 PM - psadauskas (Paul Sadauskas)

FWIW, I've run into a desire for this feature on two separate occasions recently. In both cases, I'm writing an HTTP Client for an API, and want to provide a nice interface to it. The products behind the APIs are very customizable, so different customers may see different fields returned by the API, and the APIs provide a "meta" API that describes the fields and types, etc... At runtime, I want to parse the output of the Meta API, and define a Ruby Class with attributes that match. Then when I consume the regular API, its easy to initialize instances of those Classes with the data.

However, since each customer of mine may have a different set of fields, and these classes may be temporary or ephemeral, I don't want to end up with a bunch of constants defined like MyApiClient::Customer12345::Lead, that will never get garbage collected. But, I'd still like my temporary classes to have names, since some of the libraries I'm using like Rails and Dry::Types get grumpy if given an anonymous class.

I have a slight preference towards @ioquatix's proposal #19521, but either of these will solve this particular use-case.

#32 - 04/27/2023 01:09 AM - Dan0042 (Daniel DeLorme)

I also really want the ability to give better names to dynamically created classes. In particular for something that is created once via a registry, like x =

MyTemplate["path/to/file"], having x.new.bad produce an error like "undefined method 'bad' for #<MyTemplate["path/to/file"]:0x000055e017895038>" is super useful, because you can type MyTemplate["path/to/file"] in irb and get the correct class; even if it's not a constant name, it's still eval'able.

But I think the idea of changing #name is something like a X Y problem. The problem X is that we need a more useful #inspect for instances, and the perceived solution Y is to change the value of #name of the class, but there are other ways too. Changing only #inspect or #to_s on the class would be enough, as long as they are used where needed.

```
x = Class.new
def x.to_s; "(x.to_s)"; end
def x.name; "(x.name)"; end
def x.inspect; "(x.inspect)"; end
x.ancestors   #=> [(x.inspect), Object, Kernel, BasicObject]
x.new.inspect #=> #<#<Class:0x00005651b7e5f830>:0x00005651b8448968>
```

The #inspect above would be more helpful if it was "#<(x.to_s):0x00005651b8448968>", and the problem here is that the default Object#inspect displays the class with builtin rb_class_name which ignores the #to_s and #inspect methods defined for the class. I think a good solution here would be to have rb_class_name try to call #to_s, and maybe resort to a failsafe if the result of #to_s is invalid (too long, etc).

Basically what I'm saying is that there are ways to solve this problem without changing the semantics of #name.

#33 - 04/27/2023 07:00 AM - ioquatix (Samuel Williams)

@Dan0042 unfortunately your proposal doesn't really work for nested classes without major performance issues.

#34 - 04/27/2023 12:35 PM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-33:

> doesn't really work for nested classes

Why not? Can you elaborate?

I think @Dan0042's idea is much less invasive than "breaking" Class#name and much more likely to be acceptable.
Especially sine we already changed NoMethodError messages for 3.3 (e.g. undefined method 'indent' for an instance of String).

#35 - 04/27/2023 01:27 PM - Dan0042 (Daniel DeLorme)

ioquatix (Samuel Williams) wrote in #note-33:

> @Dan0042 unfortunately your proposal doesn't really work for nested classes without major performance issues.

I sort of understand what you mean, but I think it's premature to dismiss an idea just based on the fear that it **might** have bad performance. The focus should be on whether the design/API is ok. Then **if** the implementation turns out measurably too slow **and** there is no way to remedy, the idea can be scrapped.

But let's say it really is too slow to do a rb_funcall for each nested namespace; I can already think of an easy way to mitigate that: only call #to_s if #name is undefined. That way, performance will be unchanged for regular classes assigned to constants. And that's just the first workaround that came to mind; there may even be other ways.

```
x = Class.new
x.new.inspect #=> #<#<Class:0x00005597d624bd28>:0x00005597d61aa3d8>
def x.to_s; "hey"; end
x.new.inspect #=> #<hey:0x00005597d6860998>
NAMED = x
x.new.inspect #=> #<NAMED:0x00005597d6845968>
```

#36 - 04/27/2023 03:36 PM - Eregon (Benoit Daloze)

Dan0042 (Daniel DeLorme) wrote in #note-35:

> But let's say it really is too slow to do a rb_funcall for each nested namespace;

Module#name is cached (and returns a frozen string) and should likely remain cached.
But you are not suggesting to change Module#name anyway.

However on a case such as a NoMethodError, there is no notion of parent namespace.
We would just call inspect on the class/module.
It is the responsibility of that class/module inspect to show the parent namespaces, like the default Module#inspect does.

So I don't see any performance issue here, besides having to call inspect on the Module (if overridden) for NoMethodError#message.
That is indeed a potential concern, see #18285, but it seems less problematic than object.inspect which was done previously.

#37 - 04/27/2023 03:36 PM - Eregon (Benoit Daloze)

- Related to Feature #18285: NoMethodError#message uses a lot of CPU/is really expensive to call added

#38 - 04/27/2023 09:49 PM - ioquatix (Samuel Williams)

The difference between a dynamically defined inspect which can do anything vs Module#name which is cached is significant, including but not limited to execution time, exception handling and memory allocations. As already linked, https://bugs.ruby-lang.org/issues/18285 is a direct example of this issue.

I was now just writing a native Ruby extension, where I wanted a custom rb_inspect output. The default code for rb_inspect actually invokes rb_class_name.

```
static VALUE
rb_obj_inspect(VALUE obj)
{
    if (rb_ivar_count(obj) > 0) {
        VALUE str;
        VALUE c = rb_class_name(CLASS_OF(obj));

        str = rb_sprintf("-<%"PRIsVALUE":%p", c, (void*)obj);
        return rb_exec_recursive(inspect_obj, obj, str);
    }
    else {
        return rb_any_to_s(obj);
    }
}
```

While we in theory can change this to:

```
        VALUE c = rb_inspect(CLASS_OF(obj));
```

which is in line with your (@Dan0042) proposal, I wonder how many existing code in other places would also be copying this or using rb_class_name.

You'd probably need to change the implementation of rb_class_name to call rb_inspect... which just seems like something that would cause regressions/issues to me.

This isn't just used in NoMethodError... Lots of tools report the class as part of an error or even as part of the normal execution (e.g. job class serialization, IRB, etc).

@Dan0042 why don't you make a PR for your proposal and we can try it out in the different scenarios and see if such an approach is viable?

In any case, this also affects the output of errors such that those names may not be "eval" able. @Eregon (Benoit Daloze) can you clarify if this is still a problem for you? If not, why not?

#39 - 04/28/2023 10:59 AM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-38:

> In any case, this also affects the output of errors such that those names may not be "eval" able. @Eregon (Benoit Daloze) can you clarify if this is still a problem for you? If not, why not?

Yes, it is a problem if the output just looks confusing or lies about the name.
For @Dan0042 's example in https://bugs.ruby-lang.org/issues/19520#note-32 #<MyTemplate["path/to/file"]>:0x000055e017895038> seems not so confusing, and it is eval-able.
I do wonder though, why create anonymous classes for this though?
Why not #<MyTemplate:0x000055e017895038 @path="path/to/file"> for an instance of MyTemplate and so no anonymous classes?

IMO creating anonymous classes is too expensive, it's something that makes sense for tests but not much more than that.
Similar to creating tons of singleton classes, that's just bad for performance.
And obviously anonymous classes are anonymous, they are not meant to be easy to refer to.
If you want that, why not name the class? What requires a new anonymous class instead of just using the superclass/a common named class for that use case?

#40 - 04/28/2023 01:33 PM - Dan0042 (Daniel DeLorme)

Eregon (Benoit Daloze) wrote in #note-39:

> Why not #<MyTemplate:0x000055e017895038 @path="path/to/file"> for an instance of MyTemplate and so no anonymous classes?

Because we want to compile the template to ruby once per process, and then use an instance for rendering:

```
tplclass = MyTemplate["path/to/file"] #create class and compile 'render' method (if not already done)
str = tplclass.new.render(**locals)
```

This is a pretty common pattern for template engines.

#41 - 04/28/2023 03:26 PM - Eregon (Benoit Daloze)

Ah right. So an anonymous class is used to ensure there is no conflict with the generated method name and e.g. it can just be render.
Another way to do this would be to compile to a lambda, then an instance of MyTemplate would be all that's needed (no extra classes).

I understand the desire to add this with this example.
OTOH, I feel it's enough to get the file:line from the backtrace to investigate/debug/fix when a NoMethodError or similar happens with such an instance of an anonymous class.

#42 - 05/19/2023 04:03 AM - ioquatix (Samuel Williams)

> OTOH, I feel it's enough to get the file:line from the backtrace to investigate/debug/fix when a NoMethodError or similar happens with such an instance of an anonymous class.

This isn't just about exception messages. Any time such an object is printed, e.g. via irb, a log message, any kind of formatted output, it is less informative without the proposed feature.

#43 - 05/19/2023 05:46 PM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-42:

> This isn't just about exception messages. Any time such an object is printed, e.g. via irb, a log message, any kind of formatted output, it is less informative without the proposed feature.

Is it not enough to override inspect (and maybe to_s) to affect those?

#44 - 06/08/2023 12:04 AM - ioquatix (Samuel Williams)

As I mentioned, I think using arbitrary #inspect is too risky / performance issues. If you are trying to log an error, it's much safer to use a pre-existing string than to call user code (which may fail).

#45 - 06/08/2023 06:58 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

I reject this idea, because adding a new optional name argument to new method is too easy to abuse. I am rather for the idea in #19521. Let's discuss there.

Matz.