Ruby master - Bug #18021

Mixins in Refinements: possibly multiple bugs, workarounds are awkward

07/04/2021 07:10 AM - josh.cheek (Josh Cheek)

<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>ruby -v: ruby 3.0.1p64 (2021-04-05 revision 0fb782ee38) [arm64-darwin20]</td>
</tr>
<tr>
<td>Backport:</td>
<td>2.6: UNKNOWN, 2.7: UNKNOWN, 3.0: UNKNOWN</td>
</tr>
</tbody>
</table>

Description

Maybe bug 1

Refinements must be created after methods are defined.

```ruby
# This one seems buggy
module M1
  refine(Object) { include M1 }
  def m() = M1
  using M1
  m rescue $! # => #<NameError: undefined local variable or method `m' for M1:Module>
end

# It works if you refine it after the method is defined
module M2
  def m() = M2
  refine(Object) { include M2 }
  using M2
  m #=> M2
end

# If it wasn't in a refinement, it would work
module M3
  Object.class_eval { include M3 }
  def m() = M3
  m #=> M3
end
```

Maybe bug 2

Here, a module can't delegate to another method it defines, while using a refinement. Presumably this is because the refinement is lexically scoped and where the method is defined, it is not within that scope.

```ruby
module M
  def a() = 1
  def b() = a #=> NameError: undefined local variable or method `a' for main:Object
  refine(Object) { include M }
end

using M
a #=> 1
b #=>
```

Maybe bug 3

Refinements declared after methods are defined are not available to those methods, even when they use the refinement. We try to use the module to address #2, but because of #1, the refine must go after the definitions.

```ruby
module M
  using M #=> this line is added to the example from #2
  def a() = 1
```

03/24/2022
def b() = a # ~> NameError: undefined local variable or method `a' for main:Object
end

using M
a # => 1
b # => 1

Awkward workarounds:

So, refine depends on def, which depends on using, which depends on refine, which is where we started.

Here are several ways I found to break the cycle of dependance:

1. "declare" methods by predefining them

module M
  # Predefine method stubs
  def a() = nil
  def b() = nil
  
  # Now the refinement can see them, define the refinement
  refine(Object) { include M }
  
  # Now that the refinement exists, `using' will make it available
  using M
  
  # Now method definitions can see each other,
  # so overwrite the stubs with the real implementations
  def a() = 1
  def b() = a
end

  # And now things work as expected
  using M
  a # => 1
  b # => 1

2. Put the module body in a loop

First iteration defines them without visibility to each other, because the refinement hasn't been made yet. On the second iteration, the refinement has been made, so the second time they're defined, they can see each other.

module M
  2.times do
    using M # this can go either here or after `refine' below
    def a() = 1
    def b() = a
    
    refine(Object) { include M }
  end
end

  # And now things work as expected
  using M
  a # => 1
  b # => 1

3. Define the methods in a block and call it before and after refining

module M
  methods = lambda do
    using M
    def a() 1 end
    def b() a end
  end
methods.call
refine(Object) { include M }
methods.call
end

using M
b #=> 1

4. (this does not work) Include the refinement into the module

Including this one, b/c it feels like maybe some sort of combination refinement/mixin might be what is needed. Like generally,
methods defined in a refinement can see other methods, even across other blocks, if they're all on the same module. The problem is
there isn't an obvious way to get them back out so that they're visible in the module outside of the refinement.

module M
include refine(M) { # -> ArgumentError: refinement module is not allowed
def a() = 1
def b() = a
}
refine(Object) { _include M }
end

The problem with the obvious workaround

There is an obvious workaround: don't put them in a module, just refine directly:

module M
refine Object do
def a() = 1
def b() = a
end
end
using M
b #=> 1

However, then that behaviour isn't available for mixing into other classes and objects. So if I want both (which I often find I do), then
there don't seem to be any good options available.

For example, can anyone come up with a better way to write the example below? (without moving to_type to Object, as that can't be
expected to work for the next helper method).

module Types
2.times do
using Types
private def to_type(o) = (Symbol === o ? RespondTo.new(o) : o)
def |(type) = Or.new(to_type(self), to_type(type))
refine(Module) { include Types }
refine(Symbol) { include Types }
end

RespondTo = Struct.new(:name) {
def ===> (o) o.respond_to? name end
def inspect() = "respond_to?("+name.inspect)+")"
.include(Types)

Or = Struct.new(:left, :right) {
def ===> (o) left === o || right === o end
def inspect() = "(#{left.inspect} | #{right.inspect})"
.include(Types)
end

using Types
Integer | String | (to_int | to_str)
# -> (((Integer | String) | .respond_to?(to_int) | .respond_to?(to_str)))

Related issues:
Related to Ruby master - Bug #17429: Prohibit include/prepend in refinement m...  Closed
The short answer is: never use include or prepend inside refine, it is too confusing.
Your examples would be intuitive and work if include/prepend is not used, but methods are defined directly inside refine(SomeClass/Module) do ... end.

IMHO, we should deprecate include/prepend inside refine, starting with a warning and later raise an error.

See #17429

- Related to Bug #17429: Prohibit include/prepend in refinement modules added

- Status changed from Open to Closed

Refinement#include is now deprecated and will be removed in Ruby 3.2.