Ruby trunk - Feature #11665

Support nested functions for better code organization

11/07/2015 09:17 PM - keithrbennett (Keith Bennett)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

The wisdom of using local variables has been internalized in all of us from the beginning of our software careers. If we need a variable referring to data that is used only in a single method, we create a local variable for it.

Yet if it is logic to which we need to refer, we make it an instance method instead.

In my opinion, this is inconsistent and unfortunate. The result is a bloated set of instance methods that the reader must wade through to mentally parse the class. The fact that some of these methods are used only by one other method is never communicated by the code; the reader has to discover that for him/herself.

The number of possible interactions among the instance methods is one of many measures of our software's complexity. The number of possible instance method interactions is (method_count * (method_count) - 1). Using this formula, a class with 10 methods will have a complexity of 90. If 4 of those methods are used by only 1 other method, and we could move them inside those methods, the complexity would plummet to 30 (6 * (6 - 1)), a third of the original amount!

While it is possible to extract subsets of these methods into new smaller classes, this is not always practical, especially in the case of methods called only by the constructor.

Fortunately, we do have lambdas in Ruby, so I will sometimes create lambdas inside methods for this purpose. However, lambdas are not as isolated as methods, in that they can access and modify local variables previously defined outside their scope. Furthermore, the lambdas can be passed elsewhere in the program and modify those locals from afar! So using methods would be cleaner and safer.

Another weakness of using lambdas for this purpose is that, unlike methods that are created at interpret time, lambdas are objects created at runtime -- so if a method creating 2 lambdas is called a million times in a loop, you'll need to create and garbage collect another 2 million objects. (This can be circumvented by defining the lambdas as class constants or assigning them to instance variables, but then they might as well be instance methods.)

I realize that implementing this feature would be a substantial undertaking and may not be feasible at this time. That said, I think it would be useful to discuss this now so we might benefit from its implementation someday.

(Much of this content is communicated in my talk on Ruby lambdas; slide show is at https://speakerdeck.com/keithrbennett/ruby-lambdas-functional-conf-bangalore-oct-2014 and YouTube video of the presentation at FunctionalConf in Bangalore at https://www.youtube.com/watch?v=hyRgf6Qc5pw.)

Also, this post is also posted as a blog article at http://www.bbs-software.com/blog/2015/11/07/the-case-for-nested-methods-in-ruby/.

Related issues:

Related to Ruby trunk - Feature #11670: Show warning to make nested def obsol...	**Open**
Related to Ruby trunk - Bug #11754: Visibility scope is kept after lexical sc...	**Closed**

History

#1 - 11/07/2015 09:17 PM - keithrbennett (Keith Bennett)

- Tracker changed from Bug to Feature

#2 - 11/09/2015 06:43 AM - Hanmac (Hans Mackowiak)

currently its possible to define methods inside of methods:

```
class A

  def start
    def method1
    end
    def method2
    end
  end
end

a = A.new
a.methods #=> [start]
a.start
a.methods #=> [start, method1, method2]
```

but i think that might not what you want

#3 - 11/09/2015 07:37 AM - matz (Yukihiro Matsumoto)

I am not sure nested functions are what we need. Maybe we need "real functions" with Java|C++ private scope.
Besides that, semantics for nested function is totally new in Ruby language. It cannot be a method call nor lambda.
So I don't think we can have this feature or something similar in Ruby for the near future.

But at least, the current behavior of nested method definition is useless. It should be made obsolete to open up the future possibility (I'd vote for warning).

Matz.

#4 - 11/09/2015 07:37 AM - ko1 (Koichi Sasada)

Discussion: https://docs.google.com/document/d/1D0Eo5N7NE_unIySOKG9lVj_eyXf66BQPM4PKp7NvMyQ/pub

Feel free to continue discussion on this ticket.

#5 - 11/09/2015 07:40 AM - ko1 (Koichi Sasada)

- Related to Feature #11670: Show warning to make nested def obsolete added

#6 - 11/09/2015 12:06 PM - nobu (Nobuyoshi Nakada)

private or scope local method.

```
class X
  using Module.new {
    refine X do
      def a
        :a
      end
    end
  }
  def x;a; end
end
x = X.new
p x.x #=> :a
p x.__send__(:a) #=> NoMethodError
```

#7 - 11/09/2015 12:24 PM - mame (Yusuke Endoh)

> But at least, the current behavior of nested method definition is useless

I'd like to show some cases where the current behavior is actually useful. I'm not against the change, though.

```
def warn_foo
  warn "foo"
  def warn_foo
  end
end

3.times do |i|
  p i
  warn_foo #=> warn only once
end

def enable_log
  def log(s)
    puts s
  end
```

```
end

def disable_log
  def log(s)
  end
end

enable_log
log("foo")

disable_log
log("bar")

enable_log
log("baz")
```

--
Yusuke Endoh mame@ruby-lang.org

#8 - 11/10/2015 06:42 AM - Hanmac (Hans Mackowiak)

apropos Procs and lambda, can't we create one which does not have a binding/access to local variables on the outside?

such a construct might be even faster to run, and maybe even serialize-able

#9 - 11/11/2015 01:21 PM - danielpclark (Daniel P. Clark)

Oddly there are some people who've found the dynamic defining of methods this way useful as a state machine. This recent blog post demonstrates it http://weblog.jamisbuck.org/2015/10/17/dynamic-def.html

I am inclined to agree with you Keith about the added compounding of complexity and the potential side effects of lambdas. After thinking about this it sound like what you're looking for is a lot like refinements brought down from class level into method definitions. Nobuyoshi has written an excellent example with #6 . If that could be implemented in a meta-programming way as a method on Class, maybe as scoped_def :m, *a, &b you can use it anywhere.

Here's an example without using refinements

```
class A
  def example(x)

    class << self
      private def hello
        "hello"
      end
    end

    x.call(self)

  ensure # if proc call above fails we don't want to leak the method
    class << self
      undef :hello
    end

  end
end

a = A.new

a.example ->i{ puts i.send :hello}
#hello
# => nil

a.example ->i{ puts i.hello}
#NoMethodError: private method `hello' called for #<A:0x000000012486e0>

a.send :hello
#NoMethodError: undefined method `hello' for #<A:0x000000012486e0>
```

So having something to scope methods defined within a method could be as simple as an undef at the end of your scope. I liked using a private method approach above which can only be defined for the singleton instance... but if you weren't that worried about it being a private method "during its execution" then you could just use undef.

```
class B
  def example2(x)
    def hello2
```

```
      "hello2"
    end
    x.call(self)
    ensure undef :hello2
  end
end


b = B.new

b.example2 ->i{puts i.send :hello2}
#hello2
# => nil

b.example2 ->i{puts i.hello2}
#hello2
# => nil

b.hello2
#NoMethodError: undefined method `hello2' for #<B:0x0000000128a928>
```

One thing I really don't like about implementing methods/procs/lambdas within method definitions is it's not accessible directly for testing. Sometimes I have to break the logic out into external methods just to test it before getting it working and being able to re-insert the logic back in.

#10 - 11/15/2015 01:17 AM - keithrbennett (Keith Bennett)

I completely forgot about the notation of defining instance methods within other instance method.

My first reaction to rediscovering this was that it would be useful for me, since it would visually communicate the relationship between the inner and outer methods. However, that communication would be a fiction, because the inner defined method is just another instance method. Furthermore, my guess is that defining an instance method every time a method is called is much more expensive than defining a lambda.

So I agree with what I think Matz and Hans are saying (Matz, I probably don't completely understand what you mean by '"real functions" with Java|C++ private scope."' though) -- if we can create a construct that cannot access the binding in which it was defined, and can be created only once and not every time a method is called, then that would be just about as good as nested methods (maybe even better). (As a kludge, one could do something like: @fn_x ||= -> {...}, but it would be nice not to have to define an instance variable to refer to the local function.)

Yusuke, we could always use define_method for the cases you describe, though I agree with you that it is probably clearer to the reader to use 'def function_name'.

Regarding refinements, I confess that I do not understand them yet, but I'm wondering if something as simple as a context-free function should require a relatively complex construct.

And Dan, regarding the inability to test lambdas (and possibly the new inner functions), I suggest using methods where testing at that level is required, but I believe there are many cases where lambdas can do low level implementation details whose behavior can be adequately tested by testing the enclosing method. (One can also extract a class for a complex method containing several lambdas, of course.)

-- Keith

#11 - 11/30/2015 09:20 PM - mame (Yusuke Endoh)

- Related to Bug #11754: Visibility scope is kept after lexical scope is closed added

#12 - 11/30/2015 09:27 PM - mame (Yusuke Endoh)

According to #11754, this change seemed to cause an actual issue for some gems.

--
Yusuke Endoh mame@ruby-lang.org

#13 - 11/16/2016 03:06 PM - zotherstupidguy (mohamed fouad)

Yukihiro Matsumoto wrote:

> But at least, the current behavior of nested method definition is useless. It should be made obsolete to open up the future possibility (I'd vote for warning).

```
# nested methods allow enforcing dsl constructs scopes
def a &block
  p "a"
  def b &block
    p "b"
    def c &block
      p "c"
      instance_eval &block
    end
    instance_eval &block
    undef :c
  end
  instance_eval &block
  undef :b
end
# Works
a do
  b do
    c do
    end
  end
end

# Doesn't Work
b do
end
c do
end
```

I would favor the simplicity of functional programming over the complex OOP alternatives. Not only I am applying this in my own gems, I also find that - philosophically - it matches the intention of constructing DSLs grammar via nesting.

source: https://gist.github.com/zotherstupidguy/71e45dc0cb1de7e3eb38a89931c808cf