Ruby - Feature #9992

Access Modifiers (Internal Interfaces)

06/28/2014 07:05 PM - dsferreira (Daniel Ferreira)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Hi,

I would like to discuss with you the concept of Internal Interfaces.

Currently ruby offers three access modifiers choices to set methods visibility:

- public
- protected
- private

Public methods define what we may call the Public Interface.
Private methods are private to the class and its subclasses.
Protected methods are public for the subclasses.

I would like to work with a new access modifier that would allow the creation of Internal methods.

Internal methods would be object methods that could only be called within the namespace.

Ex:

```
module Foo; end

class Foo::Bar

  def baz
     puts ‘baz’
  end

  internal :baz

end

class Foo::Qux

  def baz
     ::Foo::Bar.new.baz
  end

  public :baz

end
```

What about this use case:

```
module Foo; end

##
# Template class
#
# ==== Example
#
#          Foo::SubClass.run(:arg1)
#          #=> ...
```

```
class Foo::Base

  def initialize(arg)
    @arg = arg
  end

  def self.run(arg)
    self.new(arg).perform
  end

  public_class_method :run

  ##
  # Override abstract method

  def perform
    raise NotImplementedError
  end

  internal :perform

end

##
# Test subclass.
#
# ==== Example
#
#          Foo::Bar.run("Hello!")
#          # => My argument is Hello!

class Foo::Bar < Foo::Base

  def perform
    puts "My argument is: " + @arg
  end

  internal :perform

end
```

Is this something that we can think about in a future implementation of ruby?
An extra feature that would not break backward compatibility.

Cheers,
Daniel

History

#1 - 06/28/2014 07:17 PM - nobu (Nobuyoshi Nakada)

- Description updated

They don't seem to belong to same namespace.
Could you elaborate?

#2 - 06/28/2014 07:41 PM - dsferreira (Daniel Ferreira)

Hi Nobuyoshi,

The notion of **Internal Interface** is about being able to use the methods identified as **internal** only in the context of a given root module and all its subclasses and/or submodules. Lets call it: *namespace full tree*.

The *namespace full tree* in this context would integrate all constants of the tree defined by the namespace root and all its subnodes.

That way if we develop a new gem lets call it

gem foo

We would be able to define an **Internal Interface** that would only be used internally by gem foo classes and modules
and by any other gem foo *extension* like for instance:

gem 'foo-bar'

We have this notion of *Public* and *Internal interface* when looking at an API (Application Public Interface) in Web Services.

Web Services expose to the users a given set of methods/actions making them Public.

The remaining of the architecture is internal to the infrastructure.

In ruby libraries/packages/gems every class is available to the user.

That makes development of libraries hard because a change to the public interface of a given class may have side effects.

We may need to create a major release just because we need to change some method in one of our classes that should not be used outside of the library context.

What I aim as an architect is to allow a better defined contract between developers and users.

Having the definition of an **Internal Interface** the architect will have greater freedom to develop the required functionalities.

The development process may be like this:

1. Define Public Interface
2. Release version 1.0 once that Public Interface is well defined and will hardly change.
3. Reengineer the Internal Interface and add functionality to the Public Interface without breaking backwards compatibility.

This way dependencies between libraries will be much better understood.

Versioning of releases with this feature will be able to communicate in a greater detail to the users what nature of changes were made in each release.

We may then identify:

- Major release
  Third party libraries and/or applications might be broken.
  Library extensions might be broken.
    - Public Interface changes with broken backward compatibility

- Minor release
  Library extensions using internal interface features might be broken.
    - Public Interface changes with backward compatibility
    - Internal Interface changes with broken backward compatibility
    - Internal Interface changes with impact on overall behaviour
        - Performance improvements
        - New core engine
    - Internal Interface changes with backward compatibility
        - New features for extensions

- Patch (Tiny) release
  No side effects on extensions or third party libraries and/or applications.
    - Changes to private interface (private and protected methods)
    - Bugs
    - Refactoring
    - Unit test updates

With this feature the library development cycle will be much more linear.
Major releases will be in a much smaller number then nowadays.
We will get much more stable ecosystems.

My idea is to leverage ruby further in order to better fit the enterprise environment.

Happy to reply to any doubts and discuss with you this thoughts and what other alternatives may we have.

Cheers

Daniel

#3 - 07/01/2014 06:12 AM - zzak (zzak _)

I would suggest opening a feature request if you feel strongly about adding internal to Ruby.

See also: https://bugs.ruby-lang.org/projects/ruby/wiki/HowToRequestFeatures

#4 - 07/01/2014 06:12 AM - zzak (zzak _)

Ehh, sorry I must have missed #9992

#5 - 01/20/2016 04:41 PM - jwmittag (Jörg W Mittag)

I'm having trouble understanding what you mean by *namespace*. Classes and modules don't belong to namespaces. *Constants* belong to *modules*, but the fact that two classes which may or may not be referenced by two constants which are namespaced in the same module does not imply any sort of relationship whatsoever between either the two classes or one of the classes and the module.

Expanding your example a bit:

```
module Foo; end

Baz = Bar = Foo

class Foo::Bar
  def baz
     puts 'baz'
  end
  internal :baz
end

class Bar::Qux
  def baz
     ::Baz::Bar.new.baz
  end
end
```

Should it work? If not, why? The modules and classes involved are the *exact* same ones as in your example.

I mean, what about this:

```
foo = Class.new
bar = Class.new
baz = Module.new

baz.const_set(:Foo, foo)
baz.const_set(:Bar, bar)

quux = Module.new
quux.const_set(:Goobledigook, foo)
quux.const_set(:Blahdiblah, bar)

baz.send(:remove_const, :Bar)

module One; module Two; end end

One::Two::Three = quux::Goobledigook
```

Okay, now what namespace relations are there?

If I now do this:

```
class One::Two::Three
  internal def blub; end
end
```

Am I allowed to call foo.new.blub from a method in quux::Blahdiblah?

#6 - 05/15/2016 04:56 AM - dsferreira (Daniel Ferreira)

Hi Jörg, thank you very much for your interesting questions.

This is a subject for a wider discussion and your questions come in the right direction in my opinion.

My base view is the following example:

Fred wants to create a gem called foo.
Fred knows that he wants two methods on the interface:

```
Foo.bar
Foo.baz
```

This is all Fred aims to implement as the public interface of the foo gem for version v1.0.0.

Both Foo.bar and Foo.baz expose complex internals which Fred would like to keep isolated from outside the gem Foo namespace.

With time these internals will have an improved architecture with different *modules*, *classes*, *methods*, etc.

By using *internal* in the internal public methods Fred is confident that there is no broken backwards compatibility since no one can use the methods outside the gem namespace.

An internal method behaves:

1. Like a *public* method inside Foo namespace.
2. Like a *private* method outside Foo namespace.

So this is the logic behind the proposed implementation.

Now if Waldo does:

```
Bar = Foo
```

Shall Bar be treated as Foo or not?

What we should not allow is something like:

```
module Foo
  class Bar
     def baz
         puts 1
     end
     internal :baz
  end
end

::Foo::Bar.new.baz    => error: internal method being called outside ::Foo namespace
```

For your presented challenges I would open the discussion to the wider community.
What are the possibilities and challenges we would face in order to implement the proposed internal access modifier?

I hope we can make it happen!
It will give us development freedom and architecture control.

Note:

I can understand we may get a degradation on performance by using internal.
Maybe we could use a flag to trigger it like we do for verbose.
With the flag off internal would behave just like public.
It would be a flag to increase levels of integrity in the code that we could use with different levels in dev, uat or production environments.
Maybe the extra flag would deserve a separate proposal by its own but makes sense to present it in this context as well in my point of view.
We have debug, verbose and warning level flags.
Why not a new (architecture integrity/performance) flag?

#7 - 05/17/2016 10:49 AM - dsferreira (Daniel Ferreira)

There is a proposal for a namespace method: Object#namespace.
The namespace method would make it viable the implementation of Module#internal and simplify a lot the code I have put in place as a proof of concept in my gem internal
If we do:

```
namespace_root = ::Foo::Bar.namespace.last
# => ::Foo
```

Then Module#internal would behave like:

1. Public method if called inside namespace_root (::Foo).
2. Private method if called outside namespace_root (::Foo).

#8 - 01/20/2017 09:47 AM - dsferreira (Daniel Ferreira)

Matthew Draper presented a feature request to extended 'protected' access modifier that comes in line with my current proposal.
Matthew presents the problem in a different angle so maybe that will make more understandable the concept of *internal interfaces*.

What is your opinion?

#9 - 12/18/2017 02:31 PM - dsferreira (Daniel Ferreira)

- Description updated

#10 - 12/18/2017 02:49 PM - dsferreira (Daniel Ferreira)

What about this use case:

```
module Foo; end

##
# Template class
#
# ==== Example
#
#          Foo::SubClass.run(:arg1)
#          #=> ...

class Foo::Base

  def initialize(arg)
    @arg = arg
  end

  def self.run(arg)
    self.new(arg).perform
  end

  public_class_method :run

  ##
  # Override abstract method

  def perform
    raise NotImplementedError
  end

  internal :perform

end

##
# Test subclass.
#
# ==== Example
#
#          Foo::Bar.run("Hello!")
#          # => My argument is Hello!

class Foo::Bar < Foo::Base

  def perform
    puts "My argument is: " + @arg
  end

  internal :perform

end
```

Currently we need to make perform method public but it would be great if we could have the public interface showing only the singleton run method.

#11 - 12/18/2017 02:52 PM - dsferreira (Daniel Ferreira)

- Description updated

#12 - 12/18/2017 03:01 PM - dsferreira (Daniel Ferreira)

- Description updated

#13 - 12/18/2017 03:04 PM - dsferreira (Daniel Ferreira)

Sorry. The previous code was with a bug. I corrected it but here it is again:

What about this use case:

```
module Foo; end

##
# Template class
#
# ==== Example
#
#          Foo::SubClass.run(:arg1)
#          #=> ...

class Foo::Base

  def initialize(arg)
    @arg = arg
  end

  def self.run(arg)
    self.new(arg).perform
  end

  public_class_method :run

  ##
  # Override abstract method

  def perform
    raise NotImplementedError
  end

  internal :perform

end

##
# Test subclass.
#
# ==== Example
#
#          Foo::Bar.run("Hello!")
#          # => My argument is Hello!

class Foo::Bar < Foo::Base

  def perform
    puts "My argument is: " + @arg
  end

  internal :perform

end
```

#14 - 12/18/2017 03:04 PM - dsferreira (Daniel Ferreira)

- Description updated