Ruby master - Feature #9064

Add support for packages, like in Java

10/31/2013 03:16 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

Status:	Feedback	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		

Description

In Java, it's easy to define a package for a certain class:

package com.company.MyClass

We don't use that convention in Ruby but we have another way of packaging classes:

```
module MyLibrary
  module InnerNamespace
    class MyClass
    end
  end
end
```

I'd prefer to be able to use something like this instead meaning exactly the same thing:

```
package MyLibrary::InnerNamespace # or MyLibrary.InnerNamespace, I don't really care
class MyClass
end
```

Could you please consider this idea?

History

#1 - 10/31/2013 03:53 AM - david_macmahon (David MacMahon)

It seems two things are needed for your request.

1. Define nested modules with one statement
2. Add implicit "end" statements at the end of each source file

Nested modules can already be defined in one statement, but the parent modules need to be pre-defined:

```
module Foo; end
module Foo::Bar; end
```

It seems like you would like module Foo::Bar; end to create module Foo (if it doesn't already exist) and then module Bar under module Foo (i.e. Foo::Bar) all in one step (i.e. without having to declare module Foo first).

The second thing is just adding implicit "end" statements at EOF (or __END__) as needed to close out any unclosed modules or classes(?) or methods(??) or blocks(???).

With those two things, you would be able to write:

```
module Foo::Bar

class Baz
  # ...
end
```

which would create modules Foo and Foo::Bar and class Foo::Bar::Baz.

I'm not sure the implicit "end" statements are really saving a lot, but I kind of like the idea of being able to create nested modules in one statement.

Dave

#2 - 10/31/2013 04:47 AM - minad (Daniel Mendler)

I think it would be more interesting if you would also support package imports then (similar to Python) which would prevent namespace clashes. This could be introduced in a backward compatible way by putting unpackaged packages in some kind of root package. However I don't know if it is worth to add such a feature so late to a language and it is also not so nice to add another level of abstraction (package, module, class, ...).

Another suggestion - Ruby is powerful enough to implement some kind of package system directly in Ruby (See for example https://gist.github.com/minad/7238978)

#3 - 10/31/2013 04:54 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

I'm not against Python (and Node.js) package import approaches, but I think it's a completely separate subject from this ticket, so I'll focus on what I'm requesting for now...

David, I agree with you, and actually, I'd be already happy if "class" created the modules on the fly, but it can't without breaking compatibility, specially because it's not possible to know if the parents would be modules or classes... taking an analogy to "mkdir -p", let me say I'd be happy with something like:

```
class_p InexistentModule::Inner::MyClass
end
```

For this new "class_p" constructor, all parent constants should be a module and an error would be raised if any of them happen to exist as a class name. For classes, I'd prefer to do something like:

```
require 'some_module/inner'
class SomeModule::Inner::InnerClass
end
```

instead of using class_p...

#4 - 10/31/2013 05:23 AM - david_macmahon (David MacMahon)

On Oct 30, 2013, at 12:54 PM, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> David, I agree with you, and actually, I'd be already happy if "class" created the modules on the fly, but it can't without breaking compatibility, specially because it's not possible to know if the parents would be modules or classes...

Currently, using module Foo::Bar will raise NameError if Foo is not yet defined, so how would it break backwards compatibility if the behavior were changed to create all undefined parents as modules? You might still end up with as error later if something else tried to create class Foo, so some cases would not benefit from this (just get a different error at a different point), but it would simplify the syntax for cases where Foo really is a module.

On a semi-related note, I find it mildly frustrating that all openings of a class requires specifying the same superclass. Sometimes I just want to add a constant or simple method to a class, so it would be nice to be able to re-open that class without having to explicitly specify the same superclasses every time. It would be nice if the following were allowed:

```
class Foo < Bar; end # class Foo extends Bar
class Foo; end       # class Foo still extends Bar
```

...and...

```
class Foo; end       # class Foo has Object as a "stand-in" superclass
class Foo < Bar; end # class Foo replaces Bar as its "official" superclass
```

...and...

Of course, specifying conflicting explicit superclasses would still be an error.

```
class Foo < Object; end # class Foo has Object as its "official" superclass
class Foo < Bar; end    # error, inconsistent explicit superclass
```

Dave

#5 - 10/31/2013 05:53 AM - fuadksd (Fuad Saud)

To define a constant you could use Module#const_set, and you can avoid reopening the class with Module#instance_exec; maybe this isn't better, though.

--

Fuad Saud

#6 - 10/31/2013 06:55 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

When I talk about conflicts, I'm worried about an existing MyModule::MyClass in the project.

If you do "module MyModule::MyClass::InnerModule" without requiring the class first, it would create the MyClass constant as a module and then you would have different behavior depending on how you're requiring InnerModule.

Currently, it won't create MyClass as a module on demand, but will raise an exception instead and the developer will be warned about the mistake.

But if we change the behavior of "module" to make it create inexistent parent modules on demand it will then fail silently from current programmer expectation. This is what I refer to as backward incompatible.

#7 - 10/31/2013 06:59 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

fuadksd (Fuad Saud) wrote:

> To define a constant you could use Module#const_set...

Was this message target at me? If so, I didn't understand how const_set could help implementing this feature. Would you mind in explaining what you meant by using const_set?

#8 - 10/31/2013 09:29 AM - fuadksd (Fuad Saud)

Sorry, I forgot to mention I was replying Dave.

#9 - 10/31/2013 01:29 PM - nobu (Nobuyoshi Nakada)

(13/10/31 5:15), David MacMahon wrote:

> On a semi-related note, I find it mildly frustrating that all openings of a class requires specifying the same superclass. Sometimes I just want to add a constant or simple method to a class, so it would be nice to be able to re-open that class without having to explicitly specify the same superclasses every time. It would be nice if the following were allowed:
>
> ```
> class Foo < Bar; end # class Foo extends Bar
> class Foo; end       # class Foo still extends Bar
> ```

This is allowed already.

> ```
> class Foo; end       # class Foo has Object as a "stand-in" superclass
> class Foo < Bar; end # class Foo replaces Bar as its "official" superclass
> ```

If it is allowed,

```
class C; end
c = C.new
class C < String; end
c.size # will segfault
```

with the current object system.

--
Nobu Nakada

#10 - 10/31/2013 01:53 PM - david_macmahon (David MacMahon)

On Oct 30, 2013, at 2:55 PM, rosenfeld (Rodrigo Rosenfeld Rosas) wrote:

> When I talk about conflicts, I'm worried about an existing MyModule::MyClass in the project.
>
> If you do "module MyModule::MyClass::InnerModule" without requiring the class first, it would create the MyClass constant as a module and then you would have different behavior depending on how you're requiring InnerModule.
>
> Currently, it won't create MyClass as a module on demand, but will raise an exception instead and the developer will be warned about the mistake.

Yes, currently this raises NameError (uninitialized constant).

> But if we change the behavior of "module" to make it create inexistent parent modules on demand it will then fail silently from current programmer expectation. This is what I refer to as backward incompatible.

If we change the behavior, then the implicit creation of module MyModule and module MyModule::MyClass and module MyModule::MyClass::InnerModule will "work", but then any subsequent attempt to create class MyModule::MyClass will fail loudly. It would be equivalent to:

```
module MyModule; end
module MyModule::MyClass; end
module MyModule::MyClass::InnerModule; end
class MyModule::MyClass #=> raises TypeError: MyClass is not a class
```

So essentially it would be backwards compatible so long as it's not misused. If it is misused, then it's not backwards incompatible (double negative) since you still get an exception, though it would be a different exception raised at a different point.

Dave

#11 - 10/31/2013 02:23 PM - david_macmahon (David MacMahon)

On Oct 30, 2013, at 9:26 PM, Nobuyoshi Nakada wrote:

> (13/10/31 5:15), David MacMahon wrote:
>
> > It would be nice if the following were allowed:
> >
> > ```
> > class Foo < Bar; end # class Foo extends Bar
> > class Foo; end       # class Foo still extends Bar
> > ```
>
> This is allowed already.

Wow! Neat! Thanks! I had completely missed that! I guess I should re-check my examples before complaining!!! :-)

> > ```
> > class Foo; end       # class Foo has Object as a "stand-in" superclass
> > class Foo < Bar; end # class Foo replaces Bar as its "official" superclass
> > ```
>
> If it is allowed,
>
> ```
> class C; end
> c = C.new
> class C < String; end
> c.size # will segfault
> ```
>
> with the current object system.

Interesting. Why would it segfault instead of at least raising NoMethodError? It doesn't really matter now that I know the class can be reopened without specifying a superclass! Changing the class hierarchy does seem a little dubious, but then again having open classes at all is probably considered dubious by some...

Thanks again,
Dave

#12 - 10/31/2013 11:30 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

david_macmahon (David MacMahon) wrote:

> ... If we change the behavior, then the implicit creation of module MyModule and module MyModule::MyClass and module MyModule::MyClass::InnerModule will "work", but then any subsequent attempt to create class MyModule::MyClass will fail loudly...

What I was trying to say is that MyClass will be a different object depending on the order you require 'my_module/my_class/inner_module' unless this file explicitly requires 'my_module/my_class' before defining InnerModule. If this is not the case, you would get no error in your application if it first requires MyClass and then InnerModule. But if for some reason inner_module is loaded first at some point, things will break later when you require MyClass. Currently it will break as soon as you require InnerModule without requiring MyClass first...

I'm particularly not worried about this behavior since I always opt for explicit in my applications and it doesn't matter at all the order in which my files are required since they would always behave the same way. But since this is not a common practice in the Ruby community I think the Ruby core team might be worried about this behavior change...

#13 - 06/26/2014 12:32 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

- File feature-9064.pdf added

Attached proposal slide

#14 - 06/30/2014 08:07 AM - naruse (Yui NARUSE)

received, thanks!

#15 - 06/30/2014 05:17 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

- File feature-9064.pdf added

Reattaching using Firefox

#16 - 07/05/2014 03:13 PM - dsferreira (Daniel Ferreira)

Hi,

I only have now seen this feature request.
Luckily it came almost at the same time has my feature request for an **internal interface** (https://bugs.ruby-lang.org/issues/9992).
It seems that we are now starting to think more on how to optimize ruby for the enterprise environment and that is very good.
I support 100% this feature.

I would change the name of the command though.

For me and following the ruby way
(AFAIK: packages in ruby are gems and each gem defines its own namespace tree)

instead of:

```
package MyLibrary::InnerNamespace
```

I would sugest:

```
namespace MyLibrary::InnerNamespace
```

```
class MyClass
end
```

As an helper for wrapping the defined class inside the specified namespace.

Using namespace new command we would still rely on modules and classes for the definition of the namespace.

By using namespace
we would use the already defined namespace in the required code
or
create a namespace based on modules by default.

What do you think?

I think we are heading in the right direction.

With this **namespace** definition I think the new feature will be totally backwards compatible.

Glad Matz is already assigned to this one.

Cheers,

Daniel

#17 - 07/26/2014 05:21 AM - naruse (Yui NARUSE)

- File deleted (feature-9064.pdf)

#18 - 07/26/2014 05:30 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Feedback

I am not sure about the motivation behind this proposal.
Are the following all reasons you have:

- Reduce module .. end lines
- Reduce indentations

If so, the change is too big for small gain.

I am not (yet) against the packaging system, but I think Node.js/Python one is better than this.

Matz.

#19 - 07/28/2014 12:48 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Not all reasons, but important ones in my opinion specially as you get big namespaces and nesting level.

I didn't want to mix everything in a single request, but since you're asking for more reasons/motivation let me also state what else I'd like to be added in case packages are approved.

The first immediate benefit is that we could have something like this:

```
# a/b/my_class.rb
package A::B

class MyClass
end

# a/b/c/specialized_class.rb
require_relative '../my_class'
package A::B::C

class SpecializedClass
  def initialize
    @my_class = MyClass.new # rather than A::B::MyClass.new
  end
end

# Another feature I'd like added but I haven't included in this ticket when I created waiting for a future tic
ket as it's an enhancement over the original packaging system: "use"

# j/k/other_class.rb
package J::K
require 'a/b/my_class'
use A::B::MyClass
# or A::B or A::B::* or import rather than use, that's why I haven't proposed it yet, I don't have this part c
lear yet.
class OtherClass < MyClass
end
```

I can see lots of good things coming from some packaging system and it would also encourage more namespaces usage if people wouldn't have to type a lot to use inner classes.

But specially the search path would always be the same when using namespaces/packages.

For instance, consider this example:

```
# a/b/namespace.rb
module A
  module B
  end
end

# my_class.rb
require 'a/b/namespace'

class A::B::MyClass
end

# This is different from below with regards to the search path which I find to be a (non-obvious) problem.

module A
  module B
    class MyClass
    end
  end
end
```

Now, you got me curious about what you have in mind when you talk about integrating something like Node's packaging system into Ruby. I can't really understand how that would play well with the way Ruby works... Could you please provide an example of how you envision such packaging system in Ruby? Maybe I could like your idea better than mine...

#20 - 07/28/2014 10:27 PM - nobu (Nobuyoshi Nakada)

- Description updated

It doesn't seem that package A::B::C searches constants from A and A::B.

#21 - 07/29/2014 02:07 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

In Java it doesn't but I only borrowed Java's name, while keeping the concept of Ruby modules. Ruby modules do search classes in the parent modules and I wanted package to simply make it easier to use nested modules in Ruby.

Files

feature-9064.pdf	16.7 KB	06/30/2014	rosenfeld (Rodrigo Rosenfeld Rosas)