Uses for Modules

Practicing Ruby, September 2012

As one of Ruby's most fundamental building blocks, modules are both extremely powerful and a bit complicated. Despite being a very low level construct, they provide you with a ton of tools to make use of, including all of the following features:

- You can create nested constants using modules, which allows you to organize your code into namespaces.
- You can `include` a module into a class, mixing its functionality into all of instances of that class. It is also possible to `include` a module into other modules, to create composite mixins that can build on top of one another.
- You can also `extend` the functionality of objects on an individual basis using modules. This feature can be used to mix functionality into any object, including `Class` and `Module` objects.
- You can define methods and instance variables directly on modules, because they are first-class objects. Similarly, it is possible to dynamically generate anonymous modules using `Module.new`.
- In Ruby 2.0, you might be able to `use` modules to `refine` the functionality of other objects within a namespace, as an alternative to monkey patching.

As you can see from this list, Ruby's modules make it so that we have a lot more to think about when it comes to modeling relationships than simply whether to use class- based inheritance or object composition. They provide a whole new dimension not found in traditional object-oriented programming languages, introducing new tradeoffs for you to consider when designing your programs. When used in moderation, modules can help you create more flexible and elegant systems, but you need to understand how to work with them effectively in order to do that.

In this article, I will walk you through several practical examples of what modules can be used for, ranging from the most basic applications to the somewhat more obscure and experimental. Taken together, the scenarios I have laid out here form a decent field guide to the various applications of modules that you'll find in the wild, and will help you explore the tradeoffs of using these techniques in your own code.

Namespaces

Imagine that you are the maintainer of an XML generation library, and in it, you have a class to generate your XML documents. Perhaps uncreatively, you choose the name `Document` for your class:

```
class Document
  def generate
    # ...
  end
end
```

On its own, this seems to make a lot of sense; a user could do something simple like the following to make use of your library:

```
require "your_xml_lib"
document = Document.new
# do something with document
puts document.generate
```

But imagine that there is some other library that generates PDF documents, which happens to use similar uncreative naming for its class that does its document generation. Then, the following code would look equally valid:

```
require "their_pdf_lib"
document = Document.new
# do something with document
puts document.generate
```

As long as the two libraries were never loaded at the same time, there would be no issue. But as soon as someone loaded both libraries, some quite confusing behavior would happen. One might think that defining two different classes with the same name would lead to some sort of error being raised by Ruby, but with open classes, that is not the case. Ruby would apply the definitions of `Document` one after the other, with whatever file was required last taking precedence. The end result would almost certainly be a very broken `Document` class that could generate neither XML nor PDF.

But there is no reason for this to happen, as long as both libraries take care to wrap their classes in a namespace. The following example shows two `Document` classes that could co-exist peacefully:

```ruby
# somewhere in your_xml_lib

module XML
  class Document
    # ...
  end
end

# somewhere in their_pdf_lib

module PDF
  class Document
    # ...
  end
end
```

Using both classes in the same application is easy; you just need to explicitly include the namespace when referring to each library's `Document` class.

```ruby
require "your_xml_lib"
require "their_pdf_lib"

# this pair of calls refer to two completely different classes
pdf_document = PDF::Document.new
xml_document = XML::Document.new
```

The clash has been prevented because each library has nested its `Document` class within a module, allowing the class to be defined within that namespace rather than at the global level. Because of the way constants are resolved in Ruby, each of these libraries could even refer directly to their own `Document` object without repeating the fully qualified name within their own namespace, as shown in the following example:

```ruby
module XML
  p Document #=> XML::Document
end

module PDF
  p Document #=> PDF::Document
end
```

This behavior is convenient, but does lead to cumbersome ambiguities on occasion. The popular rack library provides an unfortunate example of this problem:

```ruby
require "rack"

module Rack
  p File      #=> Rack::File
  p ::File    #=> File
end
```

Because rack defines its own `File` class, it is placed ahead of Ruby's core class in the constant lookup order within the `Rack` module. This means that in order to access Ruby's file I/O functionality from within Rack's namespace, it is necessary to do an explicit constant lookup from the top level (i.e. `::File`), which is ugly at best, and easy to forget at worst. While in practice this kind of collision is rare, it is something to watch out for when naming your own classes and modules, even within your own namespace.

Some folks get tired of typing the fully qualified namespaces of their dependencies, and will mix top-level modules into their objects or even into the global namespace to make their code a bit more aesthetically pleasing, as shown below:

```ruby
include XML

doc = Document.new("foo.xml") #=> refers to XML::Document implicitly
doc.xpath("//h3/a").each do |link|
  #...
end
```

While in some circumstances this approach leads to more concise code, it completely negates the collision protection that namespaces provide, and can lead to some very weird and unexpected errors. The inherently unsafe nature of this technique makes it a clear anti-pattern, and hopefully the examples in this section have helped to demonstrate exactly why that is the case.

Traditional mixins

Modules are useful for namespacing, but their main purpose is to provide a convenient way to write code that be mixed into other objects, augmenting their behaviors. Because mixins facilitate code sharing in a way that is distinct from both the general OO concept of class inheritance and from things like Java's interfaces, they require you to think about your design in a way that's a bit different from most other object oriented programming languages.

While I imagine that most of our readers are comfortable with using mixins, I'll refer to some core Ruby mixins to illustrate their power. For example, consider the following bit of code which implements lazily evaluated computations:

```
class Computation

  def initialize(&block)
    @action = block
  end

  def result
    @result ||= @action.call
  end

  def <(other)
    result < other.result
  end

  def >(other)
    result > other.result
  end

  # followed by similar definitions for >=, <=, and ==
end

a = Computation.new { 1 + 1 }
b = Computation.new { 4*5 }
c = Computation.new { -3 }

p a < b  #=> true
p a <= b #=> true
p b > c  #=> true
p b >= c #=> true
p a == b #=> false
```

While Ruby makes defining custom operators easy, there is a lot more code here

than there needs to be. We can easily clean it up by mixing in Ruby's built in `Comparable` module.

```
class Computation
  include Comparable

  def initialize(&block)
    @action = block
  end

  def result
    @result ||= @action.call
  end

  def <=>(other)
    return  0 if result == other.result
    return  1 if result > other.result
    return -1 if result < other.result
  end
end

a = Computation.new { 1 + 1 }
b = Computation.new { 4*5 }
c = Computation.new { -3 }

p a < b  #=> true
p a <= b #=> true
p b > c  #=> true
p b >= c #=> true
p a == b #=> false
```

We see that our individual operator definitions have disappeared, and in its place are two new bits of code. The first new thing is just an `include` statement that tells Ruby to mix the `Comparable` functionality into the `Computation` class definition. But in order to make use of the mixin, we need to tell `Comparable` how to evaluate the sort order of our `Computation` objects, and that's where `<=>` comes in.

The `<=>` method, sometimes called the spaceship operator, essentially fills in a template method that allows `Comparable` to work. It codifies the notion of comparison in an abstract manner by expecting the method to return `-1` when the current object is considered less than the object it is being compared to, `0` when the two are considered equal, and `1` when the current object is considered greater than the object it is being compared to.

If you're still scratching your head a bit, pretend that rather than being a core Ruby object, that we've implemented `Comparable` ourselves by writing the following code:

```ruby
module Comparable
  def ==(other)
    (self <=> other) == 0
  end

  def <(other)
    (self <=> other) == -1
  end

  def <=(other)
    self < other || self == other
  end

  def >(other)
    (self <=> other) == 1
  end

  def >=(other)
    self > other || self == other
  end
end
```

Now, if you imagine these method definitions literally getting pasted into your `Computation` class when `Comparable` is included, you'll see that it would provide a behavior that is functionally equivalent to our initial example.

Of course, it wouldn't make sense for Ruby to implement such a feature for us without using it in its own structures. Because Ruby's numeric classes all implement `<=>`, we are able to simply delegate our `<=>` call to the result of the computations:

```ruby
class Computation
  include Comparable

  def initialize(&block)
    @action = block
  end

  def result
    @result ||= @action.call
  end

  def <=>(other)
    result <=> other.result
  end
end
```

The only requirement for this code to work as expected is that each `Computation`'s result must implement the `<=>` method. Since all objects that mix in `Comparable` have to implement `<=>`, any `Comparable` object returned as a result should work fine here.

While not a technically complicated example, there is surprising power in having a primitive built into your programming language which trivializes the implementation of the Template Method design pattern. For example, if you look at Ruby's `Enumerable` module and the powerful features it offers, you might think it would be a much more complicated mixin to study. But because it also hinges on a template method, simply defining `each()` gives you all sorts of complex functionality including things like `select()`, `map()`, and `reduce()`. If you haven't tried it before, you should certainly try to implement your own `Enumerable` module to get a sense of just how useful mixins can be. I have done that exercise myself many times, and even wrote a Practicing Ruby article about it.

As you may already know, it is similarly convenient to use mixins at the class level. The `Forwardable` module from Ruby's standard library provides a nice demonstration of why that can be quite useful:

```
require "forwardable"

class Stack
  extend Forwardable

  def_delegators :@data, :push, :pop, :size, :first, :empty?

  def initialize
    @data = []
  end
end
```

In this example, we can see that after we extend our `Stack` class with the `Forwardable` module, we are provided with a class level method called `def_delegators` which allows us to easily define methods which delegate to an object stored in the specified instance variable. Playing around with the `Stack` object a bit should illustrate what this code has done for us.

```
>> stack = Stack.new
=> #<Stack:0x4f09c @data=[]>
>> stack.push 1
=> [1]
>> stack.push 2
=> [1, 2]
>> stack.push 3
```

```
=> [1, 2, 3]
>> stack.size
=> 3
>> until stack.empty?
>>   p stack.pop
>> end
3
2
1
```

As before, it may be helpful to think about how we might implement `Forwardable` ourselves. The following bit of code shows one way to approach the problem.

```
module Forwardable
  def def_delegators(ivar, *delegated_methods)
    delegated_methods.each do |m|
      define_method(m) do |*a, &b|
        obj = instance_variable_get(ivar)
        obj.send(m,*a, &b)
      end
    end
  end
end
```

While the metaprogramming techniques used here may look a bit opaque if you're not familiar with them, this is fairly vanilla dynamic Ruby code. If you're curious about how it works, go ahead and try it out on your own machine to verify that it does work as expected.

An interesting thing about mixins is that they don't explicitly distinguish between methods which are designed to be mixed in at the instance level and methods which ought to be mixed in at the class level. The decision of *where* a module's functionality gets attached depends on whether `include` or `extend` is used, and what the target object is.

The somewhat indirect relationship between modules and the objects they get mixed into facilitates some very powerful customizations at the per-object level, and we will talk a bit more about that towards the end of this article. However, before we do that I'd like to cover a couple uses of modules that leverage the fact that they are not simply a low-level mechanism for implementation sharing between objects, but also first class objects themselves.

Namespaced functions

While modules are most commonly used as a mechanism for mixing functionality into other objects, it is also possible to use them directly. This technique is most commonly used for creating namespaced functions, such as in Ruby's `Math` module:

```
p Math.sin(Math::PI/2.0) #=> 1.0
p Math.log(Math::E)      #=> 1.0
```

The functions provided by the `Math` module do not depend on the kind of state and identity that objects typically do, and so it would not make sense to create instances of `Math` objects. However, because Ruby is a general purpose language, defining methods like `sin()` and `log()` in the top-level namespace would be pretty sloppy. Implementing `Math` as a module that responds to direct function calls balances the tensions between these two competing design concerns.

Ruby also uses this pragmatic approach in a few other places, including the fileutils standard library:

```
require "fileutils"

FileUtils.mkdir_p("temp/long/path/name")

# do something interesting here...
```

Emulating this style in your own code is easy, but there are several ways to do it, all of which have their own quirks to them:

1. Defining singleton methods on modules (i.e. using `def self.foo` or `class << self`) prevents those methods from being mixed into other objects.
2. The `module_function` keyword prohibits the use of private methods, and has fairly complicated semantics.
3. Using `extend self` solves these problems, but is somewhat unintuitive and tends to surprise those who have never seen it used before.

In all fairness, none of these approaches feel pure from a design perspective, but because it has the least complications, the `extend self` approach has become somewhat idiomatic. In a nutshell, `extend self` works by adding the methods defined by the module to its own lookup path, allowing those methods to be called directly. You can check out an earlier version of this article and an addendum to it for a much more nuanced discussion on this topic; I have only omitted it here to allow us to focus on practical applications rather than implementation details.

While namespaced functions are not something you will use every single day, it is not especially rare to stumble upon scenarios that can benefit from this style of design. For example, when I was working on building out the backend for a trivia website, I was given some logic for normalizing user input so that it could be compared against a predetermined pattern. While I could have stuck this logic in a number of different places, I decided I wanted to put it within a module of its own, because it did not rely on any persistent state and could be defined independently of the way our questions and quizzes were modeled. The following code is what I came up with:

```
module MinimalAnswer
  extend self

  def match?(pattern, input)
    pattern.split(/,/).any? do |e|
      normalize(input) =~ /\b#{normalize(e)}/i
    end
  end

  private

  def normalize(input)
    input.downcase.strip.gsub(/\s+/," ").gsub(/[?.!\-,:'"]/, '')
  end
end
```

The following example shows how `MinimalAnswer` is meant to be used:

```
>> MinimalAnswer.match?("Cop,Police Officer", "COP")
=> true
>> MinimalAnswer.match?("Cop,Police Officer", "police officer")
=> true
>> MinimalAnswer.match?("Cop,Police Officer", "police office")
=> false
>> MinimalAnswer.match?("Cop,Police Officer", "police officer.")
=> true
```

As I said before, this is a minor bit of functionality and could probably be shelved onto something like a `Question` object or somewhere else within the system. But the downside of that approach would be that as this `MinimalAnswer` logic began to get more complex, it would begin to stretch the scope of whatever object you attached this logic to. By breaking it out into a module right away, we give this code its own namespace to grow in, and also make it possible to test the logic in isolation, rather than trying to bootstrap a potentially much more complex object in order to test it.

Whenever you have a bit of logic that seems to not have many state dependencies between its functions, you might consider this approach. But since stateless code is rare in Ruby, you may wonder if learning about self-mixins really bought us that much. As it turns out, the technique can also be used in more stateful scenarios when you recognize that Ruby modules are objects themselves, and like any object, they can contain instance data. We'll talk about how to take advantage of that fact in the next section.

Singleton objects

> **NOTE:** Ruby overloads the term 'singleton object', so we need to be careful about terminology. What I'm about to show you is how to use self-mixed modules to implement the Singleton design pattern. Although this pattern can be awkward to implement in Ruby, modules offer one of the more reasonable ways of doing so.

I've found in object design that objects typically need zero, one, or many instances. When an object doesn't really need to be instantiated at all because it has no data in common between its behaviors, the functional approach we just reviewed often works best. The vast majority of the remaining cases fall into ordinary class definitions which facilitate many instances. Virtually everything we model fits into this category, so it's not worth discussing in detail. However, there are some rare cases in which a single object is really all we need. In particular, configuration systems come to mind. For example, take a look at the configuration object from the trivia website I mentioned earlier:

```
AccessControl.configure do
  role "basic",
    :permissions => [:read_answers, :answer_questions]

  role "premium",
    :parent      => "basic",
    :permissions => [:hide_advertisements]

  role "manager",
    :parent      => "premium",
    :permissions => [:create_quizzes, :edit_quizzes]

  role "owner",
    :parent      => "manager",
    :permissions => [:edit_users, :deactivate_users]
end
```

To guess at how this kind of object might be implemented, we need to consider how it will be used. While it is easy to imagine roles shifting over time, getting added and removed as needed, it's hard to imagine what the utility of having more than one `AccessControl` object would be. For this reason, it's safe to say that `AccessControl` configuration data is global information, and so does not need the data segregation that creating instances of a class provides.

By modeling `AccessControl` as a module rather than class, it becomes impossible to create new instances of the object, and so all the state needs to be stored within the module itself:

```ruby
module AccessControl
  extend self

  def configure(&block)
    instance_eval(&block)
  end

  def definitions
    @definitions ||= {}
  end

  # Role definition omitted, replace with a stub if you want to test
  # or refer to Practicing Ruby Issue #4
  def role(level, options={})
    definitions[level] = Role.new(level, options)
  end

  def roles_with_permission(permission)
    definitions.select { |k,v| v.allows?(permission) }.map { |k,_| k }
  end

  def [](level)
    definitions[level]
  end
end
```

There are two minor points of potential confusion in this code worth discussing, the first is the use of `instance_eval` in `configure()`, and the second is that the `definitions()` method refers to instance variables. This is where you need to remind yourself that methods are executed within the context of whatever object they get mixed into, even if that object is the module itself. Once you recognize those key points, a bit of introspection reveals what is really going on:

```
>> AccessControl.configure { "I am #{self.inspect}" }
=> "I am AccessControl"
>> AccessControl.instance_eval { "I am #{self.inspect}" }
=> "I am AccessControl"
>> AccessControl.instance_variables
=> ["@definitions"]
```

Because `AccessControl` is an ordinary Ruby object, it has ordinary instance variables and can make use of `instance_eval` just like any other object. The key difference here is that `AccessControl` is a module, not a class, and so cannot be used as a factory for creating more instances. In fact, calling `AccessControl.new` raises a `NoMethodError`.

In a traditional implementation of Singleton Pattern, you have a class which disables instantiation through the ordinary means, and creates a single instance that is accessible through the class method `instance()`. However, this seems a bit superfluous in a language in which classes are full blown objects, and so isn't necessary in Ruby.

For cases like the configuration system we've shown here, choosing to use this approach is reasonable. That having been said, the reason why I don't have another example that I can easily show you is that with the exception of this narrow application for configuration objects, I find it relatively rare to have a legitimate need for the Singleton Pattern. I'm sure if I thought long and hard on it, I could dig some other examples up, but upon looking at recent projects I find that variants of the above are all I use this technique for.

However, if you work with other people's code, it is likely that you'll run into someone implementing Singleton Pattern this way. Now, rather than scratching your head, you will have a solid understanding of how this technique works, and why someone might want to use it.

Localized monkey patches

Back in the bad old days before Prawn, I was working on a reporting framework called Ruby Reports (Ruport), which generated PDF reports via `PDF::Writer`. At the time, `PDF::Writer` was quite buggy, and essentially abandoned, but was the only game in town when it came to PDF generation.

One of the bugs was something fairly critical: Memory consumption for outputting simple PDF tables would balloon like crazy, causing a document with more than a few pages to take anywhere from several minutes to several *hours* to run.

The original author of the library had a patch laying around that inserted a hook which did some caching that greatly reduced the memory consumption, but he had not tested it extensively and did not want to cut a release. I had talked to him about possibly monkey patching `PDF::Document` in Ruport's code to add this patch, but together, we came up with a better solution: wrap the patch in a module.

```
module PDFWriterMemoryPatch
  def _post_transaction_rewind
    @objects.each { |e| e.instance_variable_set(:@parent,self) }
  end
end
```

In Ruport's PDF formatter code, we used `extend` to apply the patch:

```
@document = PDF::Document.new
@document.extend(Ruport::PDFWriterMemoryPatch)
```

Throughout our application, whenever someone interacted with a `PDF::Document` instance we created, they had a patched instance that fixed the memory leak. This meant from the Ruport user's perspective, the bug was fixed. So what makes this different from monkey patching?

Because we were only manipulating the individual objects that we created in our library, we were not making a global change that might surprise people. For example if someone was building an application that only implicitly loaded Ruport as a dependency, and they created a `PDF::Document` instance, our patch would not be loaded. This prevented us from causing unexpected behavior in any code that lived outside of Ruport itself. While this approach didn't shield us from the risks that a future change to `PDF::Writer` could potentially break our patch in Ruport, it did prevent any risk of global consequences. Anyone who's ever spent a day scratching their head because of some sloppy monkey patch in a third party dependency will immediately be able to see the value of this sort of isolation.

Ruby 2.0 may help make modules even more useful for this kind of thing via refinements. While refinements are currently considered experimental, they have

been implemented in Ruby's development branch to give people a chance to play around with them. Had this feature been available when we were working on Ruport, we could have written the previous patch in the following way:

```
module Ruport
  module PDFWriterMemoryPatch
    refine PDF::Document do
      def _post_transaction_rewind
        @objects.each { |e| e.instance_variable_set(:@parent,self) }
      end
    end
  end

  class Formatter::PDF
    use PDFWriterMemoryPatch

    def initialize
      @document = PDF::Document.new
    end
  end
end
```

The new `refine` keyword gives modules a way to define methods that will automatically override methods on other objects in a localized scope. The `use` keyword applies these refinements within the context of a particular class, module, or file. So that means that in `Ruport::Formatter::PDF`, `PDF::Document` objects will have `_post_transaction_rewind` defined on it, but that outside of that class the refinements will not be applied. This somewhat more formalized approach has various advantages and disadvantages to it, many of which are still being actively discussed. Nonetheless, refinements provide an explicit implementation of localized monkey patches that per-object mixins can only approximate.

Further reading

The very existence of modules represents a comprimise between design purity and pragmatism, and so an article on the "misuses of modules" could be just as long as this one, if not longer. However, the cost of using modules is often much harder to quantify than the benefits of doing so, and so those discussions often get very academic very quickly.

I have attempted to cover the practical downsides of using modules in various Practicing Ruby articles over the years, and if that topic interests you, I'd encourage you to check out the following articles:

- Ruby and the Singleton Pattern don't get along (Issue 2.8)
- Criteria for disciplined inheritance (Issue 3.7, Issue 3.8)
- The hidden costs of inheritance (Issue 4.9)
- Implementing the Active Record pattern (Issue 4.8, Issue 4.10)

You'll notice that most of those articles are more about inheritance than they are about modules directly. However, that is one of the most important things to note: mixing modules into objects *is* a form of inheritance. A huge amount of the problems that you'll find with class-based inheritance also apply to mixins, and it seems that this is an often neglected point.

In a language like Ruby which tries to bridge the gaps between various programming paradigms and design styles, finding the right balance and making the right choices is left up to the programmer. I encourage you to experiment with modules in your own code as you see fit, and find the right balance that works for you. However, it does not hurt to take into account the hard lessons learned by others. ;-)