Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

9. Modules and Namespaces

A Ruby module is nothing more than a grouping of objects under a single name. The objects may be constants, methods, classes, or other modules.

Modules have two uses. You can use a module as a convenient way to bundle objects together, or you can incorporate its contents into a class with Ruby's `include` statement.

When a module is used as a container for objects, it's called a *namespace*. Ruby's `Math` module is a good example of a namespace: it provides an overarching structure for constants like `Math::PI` and methods like `Math::log`, which would otherwise clutter up the main `Kernel` namespace. We cover this most basic use of modules in Recipes 9.5 and 9.7.

Modules are also used to package functionality for inclusion in classes. The `Enumerable` module isn't supposed to be used on its own: it adds functionality to a class like `Array` or `Hash`. We cover the use of modules as packaged functionality for existing classes in Recipes 9.1 and 9.4.

`Module` is actually the superclass of `Class`, so every Ruby class is also a module. Throughout this book we talk about using methods of `Module` from within classes. The same methods will work exactly the same way within modules. The only thing you can't do with a module is instantiate an object from it:

```
Class.superclass    # => Module
Math.class          # => Module
Math.new
# NoMethodError: undefined method `new' for Math:Module
```

Recipe 9.1. Simulating Multiple Inheritance with Mixins

Problem

You want to create a class that derives from two or more sources, but Ruby doesn't support multiple inheritance.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Suppose you created a class called `Taggable` that lets you associate tags (short strings of informative metadata) with objects. Every class whose objects should be taggable could derive from `Taggable`.

This would work if you made `Taggable` the top-level class in your class structure, but that won't work in every situation. Eventually you might want to do something like make a string taggable. One class can't subclass both `Taggable` and `String`, so you'd have a problem.

Furthermore, it makes little sense to instantiate and use a `Taggable` object by itself—there is nothing there to tag! Taggability is more of a feature of a class than a fullfledged class of its own. The `Taggable` functionality only works in conjunction with some other data structure.

This makes it an ideal candidate for implementation as a Ruby module instead of a class. Once it's in a module, any class can include it and use the methods it defines.

```
require 'set' # Deals with a collection of unordered values with no duplicates

# Include this module to make your class taggable. The names of the
# instance variable and the setup method are prefixed with "taggable_"
# to reduce the risk of namespace collision. You must call
# taggable_setup before you can use any of this module's methods.
module Taggable
  attr_accessor :tags

  def taggable_setup
    @tags = Set.new
  end

  def add_tag(tag)
    @tags << tag
  end

  def remove_tag(tag)
    @tags.delete(tag)
  end
end
```

Here's a taggable string class: it subclasses `String`, but it also includes the functionality of `Taggable`.

```
class TaggableString < String
  include Taggable
  def initialize(*args)
    super
    taggable_setup
  end
end
s = TaggableString.new('It was the best of times, it was the worst of times.')
s.add_tag 'dickens'
s.add_tag 'quotation'
s.tags # =>                       #<Set: {"dickens", "quotation"}>
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

A Ruby class can only have one superclass, but it can include any number of modules. These modules are called *mixins*. If you write a chunk of code that can add functionality to classes in general, it should go into a mixin module instead of a class.

The only objects that need to be defined as classes are the ones that get instantiated and used on their own (modules can't be instantiated).

If you come from Java, you might think of a module as being the combination of an interface and its implementation. By including a module, your class implements certain methods, and announces that since it implements those methods it can be treated a certain way.

When a class includes a module with the `include` keyword, all of the module's methods and constants are made available from within that class. They're not copied, the way a method is when you alias it. Rather, the class becomes aware of the methods of the module. If a module's methods are changed later (even during runtime), so are the methods of all the classes that include that module.

Module and class definitions have an almost identical syntax. If you find out after implementing a class that you should have done it as a module, it's not difficult to translate the class into a module. The main problem areas will be methods defined both by your module and the classes that include it: especially methods like `initialize`.

Your module can define an `initialize` method, and it will be called by a class whose constructor includes a `super` call (see Recipe 9.8 for an example), but sometimes that doesn't work. For instance, `Taggable` defines a `taggable_setup` method that takes no arguments. The `String` class, the superclass of `TaggableString`, takes one and only one argument. `TaggableString` can call `super` within its constructor to trigger both `String#initialize` and a hypothetical `Taggable#initialize`, but there's no way a single super call can pass one argument to one method and zero arguments to another.

That's why `Taggable` doesn't define an `initialize` method.[1] Instead, it defines a `taggable_setup` method and (in the module documentation) asks everyone who includes the module to call `taggable_setup` within their `initialize` method. Your module can define a `<module name>_setup` method instead of `initialize`, but you need to document it, or your users will be very confused.

[1] An alternative is to define `Taggable#initialize` to take a variable number of arguments, and then just ignore all the arguments. This only works because `Taggable` can initialize itself without any outside information.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

It's okay to expect that any class that includes your module will implement some methods you can't implement yourself. For instance, all of the methods in the `Enumerable` module are defined in terms of a method called `each`, but `Enumerable` never actually defines `each`. Every class that includes `Enumerable` must define what each means within that class before it can use the `Enumerable` methods.

If you have such undefined methods, it will cut down on confusion if you provide a default implementation that raises a helpful exception:

```
module Complaint
  def gripe
    voice('In all my years I have never encountered such behavior…')
  end

  def faint_praise
    voice('I am pleased to notice some improvement, however slight…')
  end

  def voice(complaint_text)
    raise NotImplementedError,
    "#{self.class} included the Complaint module but didn't define voice!"
  end
end

class MyComplaint
  include Complaint
end

MyComplaint.new.gripe
# NotImplementedError: MyComplaint included the Complaint module
# but didn't define voice!
```

If two modules define methods with the same name, and a single class includes both modules, the class will have only one implementation of that method: the one from the module that was included last. The method of the same name from the other module will simply not be available. Here are two modules that define the same method:

```
module Ayto
  def potato
     'Pohtayto'
  end
end

module Ahto
  def potato
     'Pohtahto'
  end
end
```

One class can mix in both modules:

```
class Potato
  include Ayto
  include Ahto
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

But there can be only one `potato` method for a given class or module.[2]

[2] You could get both methods by aliasing `Potato#potato` to another method after mixing in `Ayto` but before mixing in `Ahto`. There would still only be one `Potato#potato` method, and it would still be `Ahto#potato`, but the implementation of `Ayto#potato` would survive under a different name.

```
Potato.new.potato                               # => "Pohtahto"
```

This rule sidesteps the fundamental problem of multiple inheritance by letting the programmer explicitly choose which ancestor they would like to inherit a particular method from. Nevertheless, it's good programming practice to give distinctive names to the methods in your modules. This reduces the risk of namespace collisions when a class mixes in more than one module. Collisions can occur, and the later module's method will take precedence, even if one or both methods are protected or private.

See Also

- If you want a real-life implementation of a `Taggable`-like mixin, see Recipe 13.18, "Adding Taggability with a Database Mixin"

Recipe 9.2. Extending Specific Objects with Modules

Credit: Phil Tomson

Problem

You want to add instance methods from a module (or modules) to specific objects. You don't want to mix the module into the object's class, because you want certain objects to have special abilities.

Solution

Use the `Object#extend` method.

For example, let's say we have a mild-mannered `Person` class:

```
class Person
  attr_reader :name, :age, :occupation

   def initialize(name, age, occupation)
     @name, @age, @occupation = name, age, occupation
   end

   def mild_mannered?
     true
   end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Now let's create a couple of instances of this class.

```
jimmy = Person.new('Jimmy Olsen', 21, 'cub reporter')
clark = Person.new('Clark Kent', 35, 'reporter')
jimmy.mild_mannered?                        # => true
clark.mild_mannered?                        # => true
```

But it happens that some `Person` objects are not as mild-mannered as they might appear. Some of them have super powers.

```
module SuperPowers
  def fly
    'Flying!'
  end

  def leap(what)
     "Leaping #{what} in a single bound!"
  end

  def mild_mannered?
     false
  end

  def superhero_name
    'Superman'
  end
end
```

If we use `include` to mix the `SuperPowers` module into the `Person` class, it will give every person super powers. Some people are bound to misuse such power. Instead, we'll use `extend` to give super powers only to certain people:

```
clark.extend(SuperPowers)
clark.superhero_name                  # => "Superman"
clark.fly                             # => "Flying!"
clark.mild_mannered?                  # => false
jimmy.mild_mannered?                  # => true
```

Discussion

The `extend` method is used to mix a module's methods into an object, while `include` is used to mix a module's methods into a class.

The astute reader might point out that classes are actually objects in Ruby. Let us see what happens when we use `extend` in a class definition:

```
class Person
  extend SuperPowers
end

#which is equivalent to:
Person.extend(SuperPowers)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

What exactly are we extending here? Within the class definition, `extend` is being called on the `Person` class itself: we could have also written `self.extend(SuperPowers)`. We're extending the `Person` class with the methods defined in `SuperPowers`. This means that the methods defined in the `SuperPowers` module have now become class methods of `Person`:

```
Person.superhero_name                    # => "Superman"
Person.fly                               # => "Flying!"
```

This is not what we intended in this case. However, sometimes you do want to mix methods into a class, and `Class#extend` is an easy and powerful way to do it.

See Also

- Recipe 9.3, "Mixing in Class Methods," shows how to mix in class methods with `include`

Recipe 9.3. Mixing in Class Methods

Credit: Phil Tomson

Problem

You want to mix class methods into a class, instead of mixing in instance methods.

Solution

The simplest way to accomplish this is to call `extend` on the class object, as seen in the Discussion of Recipe 9.2. Just as you can use `extend` to add singleton methods to an object, you can use it to add class methods to a class. But that's not always the best option. Your users may not know that your module provides or even requires some class methods, so they might not `extend` their class when they should. How can you make an `include` statement mix in class methods as well?

To begin, within your module, define a submodule called `ClassMethods`,[3] which contains the methods you want to mix into the class:

[3] The name `ClassMethods` has no special meaning within Ruby: technically, you can call your submodule whatever you want. But the Ruby community has standardized on `ClassMethods` as the name of this submodule, and it's used in many Ruby libraries, so you should use it too.

```
module MyLib
  module ClassMethods
    def class_method
      puts "This method was first defined in MyLib::ClassMethods"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    end
  end
end
```

To make this code work, we must also define the `included` callback method within the `MyLib` module. This method is called every time a module is included in the class, and it's passed the class object in which our module is being included. Within the callback method, we extend that class object with our `ClassMethods` module, making all of its instance methods into class methods. Continuing the example:

```
module MyLib
  def self.included(receiver)
    puts "MyLib is being included in #{receiver}!"
    receiver.extend(ClassMethods)
  end
end
```

Now we can include our `MyLib` module in a class, and get the contents of `ClassMethods` mixed in as genuine class methods:

```
class MyClass
  include MyLib
end
# MyLib is being included in MyClass!

MyClass.class_method
# This method was first defined in MyLib::ClassMethods
```

Discussion

`Module#included` is a callback method that is automatically called during the inclusion of a module into a class. The default `included` implementation is an empty method. In the example, `MyLib` overrides it to extend the class that's including the `MyLib` module with the contents of the `MyLib::ClassMethods` submodule.

The `Object#extend` method takes a `Module` object as a parameter. It mixes all the methods defined in the module into the receiving object. Since classes are themselves objects, and the singleton methods of a `Class` object are just its class methods, calling `extend` on a class object fills it up with new class methods.

See Also

- Recipe 7.11, "Coupling Systems Loosely with Callbacks," covers callbacks in general and shows how to write your own
- Recipe 10.6, "Listening for Changes to a Class," covers Ruby's other class and module callback methods

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 9.4. Implementing Enumerable: Write One Method, Get 22 Free

Problem

You want to give a class all the useful iterator and iteration-related features of Ruby's arrays (`sort, detect, inject`, and so on), but your class can't be a subclass of `Array`. You don't want to define all those methods yourself.

Solution

Implement an `each` method, then include the `Enumerable` module. It defines 22 of the most useful iteration methods in terms of the `each` implementation you provide.

Here's a class that keeps multiple arrays under the covers. By defining `each`, it can expose a large interface that lets the user treat it like a single array:

```
class MultiArray
  include Enumerable

  def initialize(*arrays)
    @arrays = arrays
  end

  def each
    @arrays.each { |a| a.each { |x| yield x } }
  end
end

ma = MultiArray.new([1, 2], [3], [4])
ma.collect                                      # => [1, 2, 3, 4]
ma.detect { |x| x > 3 }                        # => 4
ma.map { |x| x ** 2 }                           # => [1, 4, 9, 16]
ma.each_with_index { |x, i| puts "Element #{i} is #{x}" }
# Element 0 is 1
# Element 1 is 2
# Element 2 is 3
# Element 3 is 4
```

Discussion

The `Enumerable` module is the most common mixin module. It lets you add a lot of behavior to your class for a little investment. Since Ruby relies so heavily on iterator methods, and almost every data structure can be iterated over in some way, it's no wonder that so many of the classes in Ruby's standard library include `Enumerable: Dir, Hash, Range`, and `String`, just to name a few.

Here's the complete list of methods you can get by including `Enumerable`. Many of them are described elsewhere in this book, especially in Chapter 4. Perhaps the most useful are `collect, inject, find_all`, and `sort_by`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Enumerable.instance_methods.sort
# => ["all?", "any?", "collect", "detect", "each_with_index", "entries",
# => "find", "find_all", "grep", "include?", "inject", "map", "max",
# => "member?", "min", "partition", "reject", "select", "sort", "sort_by",
# => "to_a", "zip"]
```

Although you can get all these methods simply by implementing an `each` method, some of the methods won't work unless your `each` implementation returns objects that can be compared to each other. For example, a data structure that contains both numbers and strings can't be sorted, since it makes no sense to compare a number to a string:

```
ma.sort                                  # => [1, 2, 3, 4]
mixed_type_ma = MultiArray.new([1, 2, 3], ["a", "b", "c"])
mixed_type_ma.sort
# ArgumentError: comparison of Fixnum with String failed
```

The methods subject to this restriction are `max, min, sort`, and `sort_by`. Since you probably don't have complete control over the types of the data stored in your data structure, the best strategy is probably to just let a method fail if the data is incompatible. This is what `Array` does:

```
[1, 2, 3, "a", "b", "c"].sort
# ArgumentError: comparison of Fixnum with String failed
```

One more example: in this one, I'll make `Module` itself include `Enumerable`. My `each` implementation will iterate over the instance methods defined by a class or module. This makes it easy to find methods of a class that meet certain criteria.

```
class Module
  include Enumerable
  def each
    instance_methods.each { |x| yield x }
  end
end

# Find all instance methods of String that modify the string in place.
String.find_all { |method_name| method_name[-1] == ?! }
# => ["sub!", "upcase!", "delete!", "lstrip!", "succ!", "gsub!",
# => "squeeze!", "downcase!", "rstrip!", "slice!", "chop!", "capitalize!",
# => "tr!", "chomp!", "next!", "swapcase!", "reverse!", "tr_s!", "strip!"]

# Find all instance methods of Fixnum that take 2 arguments.
sample = 0
sample.class.find_all { |method_name| sample.method(method_name).arity == 2 }
# => ["instance_variable_set", "between?"]
```

See Also

- Many of the recipes in Chapter 4 actually cover methods of `Enumerable`; see especially Recipe 4.12, "Building Up a Hash Using Injection"
- Recipe 9.1, "Simulating Multiple Inheritance with Mixins"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 9.5. Avoiding Naming Collisions with Namespaces

Problem

You want to define a class or module whose name conflicts with an existing class or module, or you want to prevent someone else from coming along later and defining a class whose name conflicts with yours.

Solution

A Ruby module can contain classes and other modules, which means you can use it as a namespace.

Here's some code from a physics library that defines a class called `String` within the `StringTheory` module. The real name of this class is its fully-qualified name: `StringTheory::String`. It's a totally different class from Ruby's built-in `String` class.

```
module StringTheory
  class String
    def initialize(length=10**-33)
      @length = length
    end
  end
end

String.new                                    # => ""

StringTheory::String.new
# => #<StringTheory::String:0xb7c343b8 @length=1.0e-33>
```

Discussion

If you've read Recipe 8.17, you've already seen namespaces in action. The constants defined in a module are qualified with the module's name. This lets `Math::PI` have a different value from `Greek::PI`.

You can qualify the name of any Ruby object this way: a variable, a class, or even another module. Namespaces let you organize your libraries, and make it possible for them to coexist alongside others.

Ruby's standard library uses namespaces heavily as an organizing principle. An excellent example is REXML, the standard XML library. It defines a `REXML` namespace that includes lots of XML-related classes like `REXML::Comment` and `REXML::Instruction`. Naming those classes `Comment` and `Instruction` would be a disaster: they'd get overwritten by other librarys' `Comment` and `Instruction` classes. Since nothing about the

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

genericsounding names relates them to the REXML library, you might look at someone else's code for a long time before realizing that the `Comment` objects have to do with XML.

Namespaces can be nested: see for instance `rexml's REXML::Parsers` module, which contains classes like `REXML::Parsers::StreamParser`. Namespaces group similar classes in one place so you can find what you're looking for; nested namespaces do the same for namespaces.

In Ruby, you should name your top-level module after your software project (`SAX`), or after the task it performs (`XML::Parser`). If you're writing Yet Another implementation of something that already exists, you should make sure your namespace includes your project name (`XML::Parser::SAX`). This is in contrast to Java's namespaces: they exist in its package structure, which follows a naming convention that includes a domain name, like `org.xml.sax`.

All code within a module is implicitly qualified with the name of the module. This can cause problems for a module like `StringTheory`, if it needs to use Ruby's built-in `String` class for something. This should be fixed in Ruby 2.0, but you can also fix it by setting the built-in `String` class to a variable before defining your `StringTheory::String` class. Here's a version of the `StringTheory` module that can use Ruby's builtin `String` class:

```
module StringTheory2
 RubyString = String
 class String
   def initialize(length=10**-33)
     @length = length
   end
  end

  RubyString.new("This is a built-in string, not a StringTheory2::String")
end
# => "This is a built-in string, not a StringTheory2::String"
```

See Also

- qRecipe 8.17, "Declaring Constants"
- Recipe 9.7, "Including Namespaces"

Recipe 9.6. Automatically Loading Libraries as Needed

Problem

You've written a big library with multiple components. You'd like to split it up so that users don't have to load the entire library into memory just to use part of it. But you don't want to make your users explicitly `require` each part of the library they plan to use.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Split the big library into multiple files, and set up autoloading for the individual files by calling `Kernel#autoload`. The individual files will be loaded as they're referenced.

Suppose you have a library, `functions.rb`, that provides two very large modules:

```
# functions.rb
module Decidable
  # … Many, many methods go here.
end

module Semidecidable
  # … Many, many methods go here.
end
```

You can provide the same interface, but possibly save your users some memory, by splitting `functions.rb` into three files. The `functions.rb` file itself becomes a stub full of `autoload` calls:

```
# functions.rb
autoload :Decidable, "decidable.rb"
autoload :Semidecidable, "semidecidable.rb"
```

The modules themselves go into the files mentioned in the new `functions.rb`:

```
# decidable.rb
module Decidable
  # … Many, many methods go here.
end
# semidecidable.rb
module Semidecidable
  # … Many, many methods go here.
end
```

The following code will work if all the modules are in `functions.rb`, but it will also work if `functions.rb` only contains calls to `autoload`:

```
require 'functions'
Decidable.class                                    # => Module
# More use of the Decidable module follows…
```

When `Decidable` and `Semidecidable` have been split into autoloaded modules, that code only loads the `Decidable` module. Memory is saved that would otherwise be used to contain the unsed `Semidecidable` module.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

Refactoring a library to consist of autoloadable components takes a little extra planning, but it's often worth it to improve performance for the people who use your library.

Each call to `Kernel#autoload` binds a symbol to the path of the Ruby file that's supposed to define that symbol. If the symbol is referenced, that file is loaded exactly as though it had been passed as an argument into `require`. If the symbol is never referenced, the user saves some memory.

Since you can use `autoload` wherever you might use `require`, you can autoload builtin libraries when the user triggers some code that needs them. For instance, here's some code that loads Ruby's built-in `set` library as needed:

```
autoload :Set, "set.rb"

def random_set(size)
  max = size * 10
  set = Set.new
  set << rand(max) until set.size == size
  return set
end

# More code goes here…
```

If `random_set` is never called, the `set` library will never be loaded, and memory will be saved. As soon as `random_set` gets called, the `set` library is autoloaded, and the code works even though we never explicitly `require 'set'`:

```
random_set(10)
# => #<Set: {39, 83, 73, 40, 90, 25, 91, 31, 76, 54}>

require 'set'                                    # => false
```

Recipe 9.7. Including Namespaces

Problem

You want to use the objects within a module without constantly qualifying the object names with the name of their module.

Solution

Use `include` to copy a module's objects into the current namespace. You can then use them from the current namespace, without qualifying their names.

Instead of this:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'rexml/document'

REXML::Document.new(xml)
```

You might write this:

```
require 'rexml/document'
include REXML

Document.new(xml)
```

Discussion

This is the exact same `include` statement you use to incorporate a mixin module into a class you're writing. It does the same thing here as when it includes a mixin: it copies the contents of a module into the current namespace.

Here, though, the point isn't to add new functionality to a class or module: it's to save you from having to do so much typing. This technique is especially useful with large library modules like Curses and the Rails libraries.

This use of `include` comes with the same caveats as any other: if you already have variables with the same names as the objects being included, the included objects will be copied in over them and clobber them.

You can, of course, import a namespace that's nested within a namespace of its own. Instead of this:

```
require 'rexml/parsers/pullparser'

REXML::Parsers::PullParser.new("Some XML")
```

You might write this:

```
require 'rexml/parsers/pullparser'
include REXML::Parsers

PullParser.new("Some XML")
```

See Also

- Recipe 11.3, "Extracting Data While Parsing a Document"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 9.8. Initializing Instance Variables Defined by a Module

Credit: Phil Tomson

Problem

You have a mixin module that defines some instance variables. Given a class that mixes in the module, you want to initialize the instance variables whenever an instance of the class is created.

Solution

Define an `initialize` method in the module, and call `super` in your class's constructor. Here's a `Timeable` module that tracks when objects are created and how old they are:

```
module Timeable
 attr_reader :time_created

 def initialize
   @time_created = Time.now
 end

 def age #in seconds
   Time.now - @time_created
 end
end
```

`Timeable` has an instance variable `time_created`, and an `initialize` method that assigns `Time.now` (the current time) to the instance variable. Now let's mix `Timeable` into another class that also defines an `initialize` method:

```
class Character
  include Timeable
  attr_reader :name
  def initialize( name )
    @name = name
    super() #calls Timeable's initialize
  end
end
c = Character.new "Fred"

c.time_created
# => Mon Mar 27 18:34:31 EST 2006
```

Discussion

You can define and access instance variables within a module's instance methods, but you can't actually instantiate a module. A module's instance variables only exist within objects of a class that includes the module. However, classes don't usually need to know about the

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

instance variables defined by the modules they include. That sort of information should be initialized and maintained by the module itself.

The `Character#initialize` method overrides the `Timeable#initialize` method, but you can use `super` to call the `Timeable` constructor from within the `Character` constructor. When a module is included in a class, that module becomes an *ancestor* of the class. We can test this in the context of the example above by calling the `Module#ancestors` on the `Character` class:

```
Character.ancestors                  # => [Character, Timeable, Object, Kernel]
```

When you call `super` from within a method (such as `initialize`), Ruby finds every ancestor that defines a method with the same name, and calls it too.

See Also

- Recipe 8.13, "Calling a Superclass's Method"
- Sometimes an `initialize` method won't work; see Recipe 9.3, "Mixing in Class Methods," for when it won't work, and how to manage without one
- Recipe 9.9, "Automatically Initializing Mixed-In Modules," covers an even more complex case, when you want a module to perform some initialization, without making the class that includes do anything at all beyond the initial `include`

Recipe 9.9. Automatically Initializing Mixed-In Modules

Credit: Phil Tomson

Problem

You've written a module that gets mixed into classes. Your module has some initialization code that needs to run whenever the mixed-into class is initialized. You do not want users of your module to have to call `super` in their `initialize` methods.

Solution

First, we need a way for classes to keep track of which modules they've included. We also need to redefine `Class#new` to call a module-level `initialize` method for each included module. Fortunately, Ruby's flexibility lets us makes changes to the built-in `Class` class (though this should never be done lightly):

```
class Class
  def included_modules
    @included_modules ||= []
  end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  alias_method :old_new, :new
  def new(*args, &block)
    obj = old_new(*args, &block)
    self.included_modules.each do |mod|
      mod.initialize if mod.respond_to?(:initialize)
    end
    obj
  end
end
```

Now every class has a list of included modules, accessable from the `included_modules` class method. We've also redefined the `Class#new` method so that it iterates through all the modules in `included_modules`, and calls the module-level `initialize` method of each.

All that's missing is a way to add included modules to `included_modules`. We'll put this code into an `Initializable` module. A module that wants to be initializable can mix this module into itself and define an `initialize` method:

```
module Initializable

  def self.included(mod)
    mod.extend ClassMethods
  end

  module ClassMethods
    def included(mod)
      if mod.class != Module #in case Initializeable is mixed-into a class
        puts "Adding #{self} to #{mod}'s included_modules" if $DEBUG
        mod.included_modules << self
      end
    end
  end
end
```

The `included` callback method is called whenever this module is included in another module. We're using the pattern shown in Recipe 9.3 to add an `included` callback method into the receiving module. If we didn't do this, you'd have to use that pattern yourself for every module you wanted to be `Initializable`.

Discussion

That's a lot of code, but here's the payoff. Let's define a couple of modules which include `Initializeable` and define `initialize` module methods:

```
module A
  include Initializable
  def self.initialize
    puts "A's initialized."
  end
end

module B
  include Initializable
  def self.initialize
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    puts "B's initialized."
  end
end
```

We can now define a class that mixes in both modules. Instantiating the class instantiates the modules, with not a single `super` call in sight!

```
class BothAAndB
  include A
  include B
end

both = BothAAndB.new
# A's initialized.
# B's initialized.
```

The goal of this recipe is very similar to Recipe 9.8. In that recipe, you call `super` in a class's `initialize` method to call a mixed-in module's `initialize` method. That recipe is a lot simpler than this one and doesn't require any changes to built-in classes, so it's often preferable to this one.

Consider a case like the `BothAAndB` class above. Using the techniques from Recipe 9.8, you'd need to make sure that both `A` and `B` had calls to `super` in their `initialize` methods, so that each module would get initialized. This solution moves all of that work into the `Initializable` module and the built-in `Class` class. The other drawback of the previous technique is that the user of your module needs to know to call `super` somewhere in their `initialize` method. Here, everything happens automatically.

This technique is not without its pitfalls. Anytime you redefine critical built-in methods like `Class#new`, you need to be careful: someone else may have already redefined it elsewhere in your program. Also, you won't be able to define your own `included` method callback in a module which includes `Initializeable`: doing so will override the callback defined by `Initializable` itself.

See Also

- Recipe 9.3, "Mixing in Class Methods"
- Recipe 9.8, "Initializing Instance Variables Defined by a Module"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01