Modules and Classes

Modules

Modules serve two purposes in Ruby, namespacing and mix-in functionality.

A namespace can be used to organize code by package or functionality that separates common names from interference by other packages. For example, the IRB namespace provides functionality for irb that prevents a collision for the common name “Context”.

Mix-in functionality allows sharing common methods across multiple classes or modules. Ruby comes with the Enumerable mix-in module which provides many enumeration methods based on the each method and Comparable allows comparison of objects based on the <=> comparison method.

Note that there are many similarities between modules and classes. Besides the ability to mix-in a module, the description of modules below also applies to classes.

Module Definition

A module is created using the module keyword:

```ruby
module MyModule
  # ...
end
```

A module may be reopened any number of times to add, change or remove functionality:

```ruby
module MyModule
  def my_method
  end
end

module MyModule
  alias my_alias my_method
end

module MyModule
  remove_method :my_method
end
```

Reopening classes is a very powerful feature of Ruby, but it is best to only reopen classes you own. Reopening classes you do not own may lead to naming conflicts or difficult to diagnose bugs.

Nesting

Modules may be nested:

```
module Outer
  module Inner
  end
end
```

Many packages create a single outermost module (or class) to provide a namespace for their functionality.

You may also define inner modules using :: provided the outer modules (or classes) are already defined:

```
module Outer::Inner::GrandChild
end
```

Note that this will raise a NameError if Outer and Outer::Inner are not already defined.

This style has the benefit of allowing the author to reduce the amount of indentation. Instead of 3 levels of indentation only one is necessary. However, the scope of constant lookup is different for creating a namespace using this syntax instead of the more verbose syntax.

Scope

self

self refers to the object that defines the current scope. self will change when entering a different method or when defining a new module.

Constants

Accessible constants are different depending on the module nesting (which syntax was used to define the module). In the following example the constant A::Z is accessible from B as A is part of the nesting:

```
module A
  Z = 1

  module B
    p Module.nesting #=> [A::B, A]
    p Z #=> 1
  end
end
```

However, if you use :: to define A::B without nesting it inside A a NameError exception will be raised because the nesting does not include A:

```
module A
  Z = 1
end

module A::B
  p Module.nesting #=> [A::B]
  p Z #=> raises NameError
end
```

If a constant is defined at the top-level you may preceded it with :: to reference it:

```
Z = 0

module A
  Z = 1

  module B
    p ::Z #=> 0
  end
end
```

Methods

For method definition documentation see the syntax documentation for methods.

Class methods may be called directly. (This is slightly confusing, but a method on a module is often called a "class method" instead of a "module method". See also Module#module_function which can convert an instance method into a class method.)

When a class method references a constant it uses the same rules as referencing it outside the method as the scope is the same.

Instance methods defined in a module are only callable when included. These methods have access to the constants defined when they were included through the ancestors list:

```
module A
  Z = 1

  def z
    Z
  end
end

include A

p self.class.ancestors #=> [Object, A, Kernel, BasicObject]
p z #=> 1
```

Visibility

Ruby has three types of visibility. The default is public. A public method may be called from any other object.

The second visibility is protected. When calling a protected method the sender must be a subclass of the receiver or the receiver must be a subclass of the sender. Otherwise a NoMethodError will be raised.

Protected visibility is most frequently used to define == and other comparison methods where the author does not wish to expose an object's state to any caller and would like to restrict it only to inherited classes.

Here is an example:

```
class A
  def n(other)
    other.m
  end
end

class B < A
  def m
    1
  end

  protected :m

end

class C < B
end

a = A.new
b = B.new
c = C.new

c.n b #=> 1 -- C is a subclass of B
b.n b #=> 1 -- m called on defining class
a.n b # raises NoMethodError A is not a subclass of B
```

The third visibility is private. A private method may not be called with a receiver, not even self. If a private method is called with a receiver a NoMethodError will be raised.

alias and undef

You may also alias or undefine methods, but these operations are not restricted to modules or classes. See the miscellaneous syntax section for documentation.

Classes

Every class is also a module, but unlike modules a class may not be mixed-in to another module (or class). Like a module, a class can be used as a namespace. A class also inherits methods and constants from its superclass.

Defining a class

Use the class keyword to create a class:

```
class MyClass
  # ...
end
```

If you do not supply a superclass your new class will inherit from Object. You may inherit from a different class using < followed by a class name:

```
class MySubclass < MyClass
  # ...
end
```

There is a special class BasicObject which is designed as a blank class and includes a minimum of built-in methods. You can use BasicObject to create an independent inheritance structure. See the BasicObject documentation for further details.

Inheritance

Any method defined on a class is callable from its subclass:

```
class A
  Z = 1

  def z
    Z
  end
end

class B < A
end

p B.new.z #=> 1
```

The same is true for constants:

```
class A
  Z = 1
```

```ruby
end

class B < A
  def z
    Z
  end
end

p B.new.z #=> 1
```

You can override the functionality of a superclass method by redefining the method:

```ruby
class A
  def m
    1
  end
end

class B < A
  def m
    2
  end
end

p B.new.m #=> 2
```

If you wish to invoke the superclass functionality from a method use super:

```ruby
class A
  def m
    1
  end
end

class B < A
  def m
    2 + super
  end
end

p B.new.m #=> 3
```

When used without any arguments super uses the arguments given to the subclass method. To send no arguments to the superclass method use super(). To send specific arguments to the superclass method provide them manually like super(2).

super may be called as many times as you like in the subclass method.

Singleton Classes

The singleton class (also known as the metaclass or eigenclass) of an object is a class that holds methods for only that instance. You can access the singleton class of an object using class << object like this:

```
class C
end

class << C
  # self is the singleton class here
end
```

Most frequently you'll see the singleton class accessed like this:

```
class C
  class << self
    # ...
  end
end
```

This allows definition of methods and attributes on a class (or module) without needing to write def self.my_method.

Since you can open the singleton class of any object this means that this code block:

```
o = Object.new

def o.my_method
  1 + 1
end
```

is equivalent to this code block:

```
o = Object.new

class << o
  def my_method
    1 + 1
  end
end
```

Both objects will have a my_method that returns 2.