

CSc 372

Comparative Programming Languages

34 : Ruby — Modules

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2011 Christian Collberg

Christian Collberg

- Modules define *namespaces*. This allows you to have several methods or constants with the same name.

```
module Duck
  def Duck.speak()
    return "quack"
  end
end

module Goose
  def Goose.speak()
    return "honk"
  end
end
```

Methods in Modules

- Outside of the module M, you refer to one of its methods meth as M.meth:

```
module Duck
  def Duck.speak()
    return "quack"
  end
end
```

```
puts Duck.speak()
puts Goose.speak()
```

```
module Goose
  def Goose.speak()
    return "honk"
  end
end
```

Constants in Modules

- Outside of the module M you refer to one of its constants con as M::con.

```
module Duck
  IS_CUTE = true
end
```

```
module Goose
  IS_CUTE = false
end
```

```
puts Duck::IS_CUTE
```

Classes in Modules

- You can define a class within a module. Since the class name is essentially a constant, you reference the class using `::`.

```
module Fowl
  class Duck
    def speak()
      puts "quack!"
    end
  end
  class Goose
    def speak()
      puts "honk!"
    end
  end
end

d = Fowl::Duck.new()
d.speak()
g = Fowl::Goose.new()
g.speak()
```

Modules in Modules

- You can even have modules inside modules!

```
module Birdies
  module Duckie
    def Duckie.speak()
      puts "quack!"
    end
  end
  module Goosie
    def Goosie.speak()
      puts "honk!"
    end
  end
end

Birdies::Duckie.speak()
Birdies::Goosie.speak()
```

Including Modules

- You can put several definitions in one file:

duckies.rb	goosies.rb
<pre>module Duck IS_CUTE = true def Duck.speak() return "quack" end end</pre>	<pre>module Goose IS_CUTE = false def Goose.speak() return "honk" end end</pre>

Including Modules...

- You include the file in by saying `"require 'file'"` (or `"load 'file'"` but this will load the definitions multiple times if you load more than once):

main.rb
<pre>require 'duckies' require 'goosies' puts Duck.speak() puts Goose.speak() puts Duck::IS_CUTE</pre>

Mixins

- Create a module with instance methods which may be useful in many different kinds of classes:

```
module Debug
  def printme()
    puts "#{self.class.name}" +
      "(\##{self.object_id})"
  end
end
```

Mixins...

- *Include* a module within a class and its instance methods automatically become available in the class:

```
class Ducktape
  include Debug
  def color()
    puts "silver"
  end
end

d = Ducktape.new()
d.printme()
```

- You're including a *reference* to the module: any change to it will affect all classes in which it is included.

Mixing in module Comparable

- Include Comparable in your class and define your own <=> method (returning 1, -1, or 0, for greater-than, less-than, or equal, respectively).

```
module Comparable
  def ==(arg)
  end
  def >=(arg)
  end
  def <(arg)
  end
  ...
end
```

Mixing in module Comparable...

```
class Ducktape
  include Comparable
  attr_reader :size
  def initialize(size)
    @size = size
  end
  def <=>(other)
    if self.size > other.size then return 1
    elsif other.size > self.size then return -1
    else return 0
    end
  end
end
```

Mixing in module Comparable...

- Your class now gets immediate access to the methods that Comparable defines (<, <=, >, >=):

```
small = Ducktape.new(100)
large = Ducktape.new(200)
```

```
puts small < large
puts small > large
puts small == large
puts small <= large
puts small >= large
```

Mixing in Enumerable

- Include the Enumerable module and define an each() method.

```
module Enumerable
  def each_with_index
  end
  def collect
  end
  def sort
  end
  def member?(arg0)
  end
  def inject(arg0, arg1, *rest)
  end
  ...
end
```

Mixing in Enumerable...

```
class Flock
  include Enumerable

  def initialize(mum, dad, babies)
    @mum = mum
    @dad = dad
    @babies = babies
  end

  def each()
    yield @mum
    yield @dad
    @babies.each() {|b| yield b}
  end
end
```

Mixing in Enumerable...

- You now get access to methods such as collect(), sort(), and inject():

```
f = Flock.new("daisy","donald",
             ["huey","dewey","louie"])

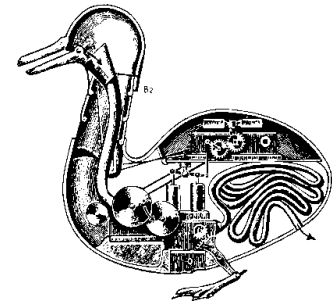
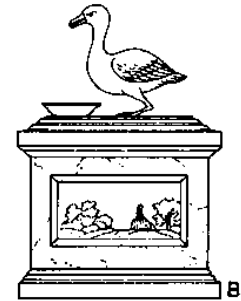
f.each() {|x| puts x}

puts f.collect {|x| x.length()}

puts f.sort()

puts f.inject() {|v,x| (v=="")?x:v+" "+x}
```

- Read Chapter 9, page 117–125, in *Programming Ruby — The Pragmatic Programmers Guide*, by Dave Thomas.



... [French engineer Jacques] de Vaucanson [1709-82] built ... a mechanical duck which could move in the typical, wagging way of a duck, eat and digest fish, and excrete the remains in a “natural” way. The mechanism was driven by a weight and had more than a thousand moving parts. . .

From: <http://music.calarts.edu/~sroberts/articles/DeVaucanson.duck.html>