Module

Outline

1. Getting Modular
2. Module as namespace
3. “Include” and “extend” module

1. Getting modular

Mixing it up:

- A **module** is a named group of methods, constants, and class variables
- Modules only hold *behaviour*
- A <u>class object</u> is an instance of the ***Class*** class, a <u>module object</u> is an instance of the ***Module*** class

  => "All classes are modules, but not all modules are classes"
- Using ***module*** keyword to define a modules
- A modules can’t be instantiated, can't be subclassed, no "module hierarchy" of inheritance

  => Ruby modules allow create groups of methods that can then ***include*** or ***mix*** into any number of classes

1. Getting modular (cont.)

Example code:

```
module WarmUp
  def push_ups
    "Phew, I need a break!"
  end
end

class Gym
  include WarmUp

  def preacher_curls
    "I'm building my biceps."
  end
end
```

```
class Dojo
  include WarmUp

  def tai_kyo_kyu
    "Look at my stance!"
  end
end

puts Gym.new.push_ups   #=> Phew, I need
a break!
puts Dojo.new.push_ups  #=> Phew, I need
a break!
```

1. Getting modular (cont.)

Some hierarchy:

- All classes are instances of Ruby’s **Class**, all modules in Ruby are instances of **Module**
- **Module** is the superclass of **Class**

```
module WarmUp
end

puts WarmUp.class        # Module
puts Class.superclass    # Module
puts Module.superclass   # Object
```

1. Getting modular(cont.)

Mixins in Ruby:

- Class can inherit features from multiple parent class, the class is supposed to show multiple inheritance
- Ruby does not support multiple inheritance directly but Ruby Modules have another wonderful use

  => **mixin**

```
module A
  def a1; end
  def a2; end
end

module B
  def b1; end
  def b2; end
end
```

```
class Sample
  include A
  include B
  def some_thing
  end
end
sample = Sample.new
sample.a1
sample.b1
sample.some_thing
```

2. Module as Namespace

Define module with namespace:

- Namespacing is a way of building logically related objects together
- This is allow classes or modules with conficting name to co-exist while avoiding collision
- Modules are a good way to group *related methods* when object-oriented programming is not necessary
- Modules can also hold classes

```
module Perimeter
  class Array
    def initialize
      @size = 400
    end
  end
end

our_array = Perimeter::Array.new
ruby_array = Array.new

p our_array.class         #=> Perimeter::Array
p ruby_array.class        #=> Array
```

2. Module as Namespace

Modules without namespace:

```
class Push
  def up
    40
  end
end

require "gym"      #=> up returns 40
gym_push = Push.new
p gym_push.up
```

```
class Push
  def up
    30
  end
end

require "dojo"      #=> up returns 30
dojo_push = Push.new
p dojo_push.up
```

2. Module as Namespace

Using namespace:

```
module Gym
  class Push
    def up
      puts 40
    end
  end
end
require "gym"
```

```
module Dojo
  class Push
    def up
      puts 30
    end
  end
end
require "dojo"
```

```
dojo_push = Dojo::Push.new
p dojo_push.up     #=> 30

gym_push = Gym::Push.new
p gym_push.up    #=> 40
```

2. Module as Namespace

```
module Dojo
 A = 4
 module Kata
      B = 8
  module Roulette
   class ScopeIn
    def push
     15
    end
   enda
  end
 end
end

A = 16
B = 23
C = 42
```

```
puts "A - #{A}"                #=> A - 16
puts "Dojo::A - #{Dojo::A}"        #=> Dojo::A - 4

puts "B - #{B}"                #=> B - 23
puts "Dojo::Kata::B - #{Dojo::Kata::B}"   #=>
Dojo::Kata::B - 8

puts "C - #{C}"
puts "Dojo::Kata::Roulette::ScopeIn.new.push -
#{Dojo::Kata::Roulette::ScopeIn.new.push}"
```

=> :: operator: **constant lookup**

3. "include" and "extend" Modules

"include" Modules: ***include*** is only add instance level methods - not class level methods

```
module Foo
  def foo_name
    puts "My name is Boo!!!"
  end
end

class Bar
  include Foo
end

Bar.new.foo_name       #=> My name is Boo!!!
```

3. "include" and "extend" Modules

"included"callback: "included" method callback that Ruby invokes whenever the module is included into another module/class

```
module Foo
  def self.included klass
    puts "Foo has been included
              in  class #{klass}"
  end
end

class Bar
  include Foo
end

#=> Foo has been included in  class Bar
```

```
module Sample
  module ClassMethods
  end

  module InstanceMethods
  end

  def self.included receiver
    receiver.extend ClassMethods
    receiver.send :include, InstanceMethods
  end
end
```

3. "include" and "extend" Modules

"extend" Modules: ***extend*** method works similar to ***include,*** can use it to extend any object by including methods and constants from a module

```
module Foo
  def module_method
    puts "Module Method invoked"
  end
end

class Bar
  # extend Foo
end

bar = Bar.new
bar.extend Foo
bar.module_method  #=> Module Method invoked
```

3. "include" and "extend" Modules

"extended" callbacks:

```
module Foo
  def self.extended base
    puts "Class #{base} has been extended with module #{self} !"
  end
end

class Bar
  extend Foo
end

#=> Class Bar has been extended with module Foo !
```

References

- http://ruby-doc.org/
- http://rubylearning.com/satishtalim/modules_mixins.html
- https://learnrubythehardway.org/book/ex40.html
- http://www.rubyfleebie.com/an-introduction-to-modules-part-1/
- http://www.rubyfleebie.com/an-introduction-to-modules-part-2/

Thank you for listening!