Ruby: Symbols and Object-Oriented Concepts

Lecture 9

Symbols

- Roughly: *unique* & *immutable* strings
- Syntax: prefix with ":"

```
:height
:'some symbol'
:"#{name}'s crazy idea"
```

- Easy (too easy?) to convert between symbols and strings

```
:name.to_s          #=> "name"
'name'.to_sym       #=> :name
```

- But symbols are *not* strings

```
:name == 'name' #=> false
```

Operational View

- A symbol is created once, and all uses refer to that same object (aliases)
- Symbols are immutable
- Example

```
[].object_id #=> 200
[].object_id #=> 220
[].equal? [] #=> false
:world.object_id #=> 459528
:world.object_id #=> 459528
:world.equal? :world #=> true
```

Symbols as Hash Keys

- Literal notation, but note colon location!

```
colors = {red: 0xf00,
          green: 0x0f0,
          blue: 0x00f}
```

- This is just syntactic sugar
  - `{name: value}` same as `{:name => value}`
  - The key is a *symbol* (eg `:red`)
- Pitfalls

```
colors.red     #=> NoMethodError
colors["red"]  #=> nil
colors[:red]   #=> 3840 (ie 0xf00)
```

Keyword Arguments

- Alternative to positional matching of arguments with formal parameters

```
def display(first:, last:)
  puts "Hello #{first} #{last}"
end
display first: 'Mork', last: 'Ork'
display last: 'Hawking', first: 'Steven'
```

- Providing a default value makes that argument optional

```
def greet(title: 'Dr.', name:)
  puts "Hello #{title} #{name}"
end
```

- Benefits: Client code is easier to read, and flexibility in optional arguments

Classes

- Classes have methods and variables

```
class LightBulb     # name with CamelCase
  def initialize    # special method name
    @state = false # @ means "instance variable"
  end
  def on?
    @state          # implicit return
  end
  def flip_switch!  # name with snake_case
    @state = !@state
  end
end
```

- Instantiation calls *initialize* method

```
f = LightBulb.new #=> <LightBulb:0x0000e71c2322
                           @state=false>
f.on? #=> false
```

Visibility

- Instance variables are always private
  - Private to *object*, not class
- Methods can be private, protected, or public (default)

```
class LightBulb
  private def inside
    …
  end

  def access_internals(other_bulb)
    inside # ok
    other_bulb.inside # no! inside is private
    self.inside  # no explicit recv'r allowed
  end
end
```

Getters/Setters


```
class LightBulb
  def initialize(color, state: false)
    @color = color # not visible outside object
    @state = state # not visible outside object
  end
  def color
    @color
  end
  def state
    @state
  end
  def state=(value)
    @state = value
  end
end
```

Attributes


```
class LightBulb
  def initialize(color, state: false)
    @color = color
    @state = state
  end
  def color
    @color
  end

  attr_accessor :state # name is a symbol



end
```

Attributes


```
class LightBulb
  def initialize(color, state: false)
    @color = color
    @state = state
  end

  attr_reader :color


  attr_accessor :state




end
```

Attributes


```
class LightBulb
  attr_reader :color
  attr_accessor :state
  attr_writer :size

  def initialize(color, state: false)
    @color = color
    @state = state
    @size = 0
  end
end
```

Classes Are Always Open

- A class can always be extended

```
class Street
  def construction … end
end
…
class Street
  def repave … end # Street now has 2 methods
end
```

- Applies to core classes too

```
class Integer
  def log2_of_cube # lg(self^3)
    (self**3).to_s(2).length - 1
  end
end
500.log2_of_cube #=> 26
```

Classes are Always Open (!)

- Existing methods can be redefined!

- When done with system code (libraries, core …) called "monkey patching"
- Tempting, but… Just Don't Do It

No Overloading

- Method identified by (symbol) name
  - No distinction based on number of arguments
- Approximation: default arguments

```
def initialize(width, height = 10)
  @width = width
  @height = height
end
```

- Old alternative: trailing options hash

```
def initialize(width, options)
```

- Modern style: default keyword arguments

```
def initialize(height: 10, width:)
```

A Class is an Object Instance too

- Even classes are objects, created by :new

```
LightBulb = Class.new do #class LightBulb
  def initialize
    @state = false
  end
  def on?
    @state
  end
  def flip_switch!
    @state = !@state
  end
end
```

Instance, Class, Class Instance


```
class LightBulb
  @state1           # class instance var
  def initialize
    @state2 = …     # instance variable
    @@state3 = …    # class variable
  end
  def bar           # instance method
    …               # sees @state2, @@state3
  end
  def self.foo      # class method
    …               # sees @state1, @@state3
  end
end
```

Inheritance

- Single inheritance between classes

```
class LightBulb < Device
  …
end
```

  - Default superclass is Object (which inherits from BasicObject)
- Keyword **super** to call parent's method
  - No args means forward all args

```
class LightBulb < Device
  def electrify(current, voltage)
    do_work
    super # with current and voltage
  end
end
```

Modules

- Another container for definitions

```
module Stockable
  MAX = 1000
  class Item … end
  def self.inventory … end # utility fn
  def order … end
end
```

- Cannot, themselves, be instantiated

```
s = Stockable.new          # NoMethodError
i = Stockable::Item.new    # ok
Stockable.inventory        # ok
Stockable.order            # NoMethodError
```

Modules as Namespaces

- Modules create independent namespaces
  - cf. packages in Java
- Access contents via scoping (::)

```
Math::PI      #=> 3.141592653589793
Math::cos 0 #=> 1.0
widget = Stockable::Item.new
x = Stockable::inventory
Post < ActiveRecord::Base
BookController < ActionController::Base
```

- Style: use dot to invoke utility functions (ie module methods)

```
Math.cos 0   #=> 1.0
Stockable.inventory
```

Modules are Always Open

- Module contains several related classes
- Style: Each class should be in its own file
- So split module definition

```
# game.rb
module Game
end

# game/card.rb
module Game
  class Card … end
end

# game/player.rb
module Game
  class Player … end
end
```

Modules as "Mixins"

- Another container for method definitions

```
module Stockable
  def order … end
end
```

- A module can be *included* in a class

```
class LightBulb < Device
  include Stockable, Comparable …
end
```

- Module's (instance) methods become (instance) methods of the class

```
bulb = LightBulb.new
bulb.order              # from Stockable
if bulb <= old_bulb # from Comparable
```

Requirements for Mixins

- Mixins often rely on certain aspects of classes into which they are included
- Example: Comparable methods use **#<=>**

```
module Comparable
  def <(other) … end
  def <=(other) … end
end
```

- Enumerable methods use **#each**
- Recall *layering* in SW I/II? Roughly:
  - Class implements kernel methods
  - Module implements secondary methods

Software Engineering

- All the good principles of SW I/II apply
- Single point of control over change
  - Avoid magic numbers
- Client view: abstract state, contracts, invariants
- Implementer view: concrete rep, correspondence, invariants
- Checkstyle tool: rubocop
- Documentation: YARD
  - Notation for types: yardoc.org/types.html

```
@param words Array<String> the lexicon
```

Summary

- Classes as blueprints for objects
  - Contain methods and variables
  - Public vs private visibility of methods
  - Attributes for automatic getters/setters
- Metaprogramming
  - Classes are objects too
  - “Class instance” variables
- Single inheritance
- Modules are namespaces and mixins