# Metaprogramming

## Metaprogramming

- **Definition:**
  - Writing code that writes other code
  - Writing code that manipulates language constructs at runtime.

## Ruby is dynamic

- Classes (and Modules) can be modified as the program is running
- Metaprogramming-friendly
  - Allows one to write code that manipulates language constructs at runtime
- Magic?
- Is this a good thing?

## Open Classes

- `class` keyword creates a new class or adds to an existing class with the same name
- No distinction between code that defines a class and code of any other kind
- This is why there is no method overloading in Ruby - the latter method definition becomes the new definition of the method

## Open Classes example 1

```
class MyClass
  def test
    puts "test1"
  end
  def test
    puts "test2"
  end
end

my_class = MyClass.new
my_class.test # => test2

class MyClass
  def test
    puts "test3"
  end
end

my_class.test # => test3
```

## Open Classes (continued)

- If a class with the same name is defined multiple times - it becomes a merge (when the method names are different)
- This is useful if you want to define methods for a class / module in logical groups
- The `class` keyword behaves more like a scope operator than a class declaration

## Open Classes example 2

```
class MyClass
  def one_method
    puts "one_method"
  end
end

class MyClass
  def another_method
    puts "another_method"
  end
end

my_class = MyClass.new
my_class.one_method # => one_method
my_class.another_method # => another_method
```

## Monkey patching built-in classes

- Not only can you modify and manipulate your own classes, but you can also do the same with the built-in Ruby classes
- This practice is sometimes called “monkey patching”
- Are you serious???!!!
  - Yep. Use extreme caution!!!

## Monkey patching BAD example

```ruby
class String
  def really
    puts "YES, really"
  end
  def length
    2
  end
end

"hi, there".really # => YES, really
puts "hi,there".length # => 2 (That can't be right!!!)
```

## Monkey patching GOOD example

```
class Fixnum
  def seconds
    self
  end
  def minutes
    60 * seconds
  end
  def hours
    60 * minutes
  end
end

time1 = Time.now
time2 = time1 + 5.minutes
time3 = time1 + 10.minutes
puts time2.between?(time1, time3) # => true
```

…Rails 3 does something similar …

## Kernel Methods

- `Kernel` - module that contains well-known Ruby methods like `puts`, `gets`, `class` etc.
- `Object` mixes-in `Kernel` module, making the built-in functions globally accessible
- Possible to add your own methods to the `Kernel` module

## Adding to Kernel module example

```
module Kernel
  def log(message)
    puts "#{Time.now.strftime("%m/%d/%Y %H:%M:%S - ")} #{message}"
  end
end

puts "Regular Kernel"# => Regular Kernel
log "Our Addition to the Kernel" # => 08/12/2011 16:57:21 - Our Addition to the Kernel

p Object.ancestors# => [Object, Kernel, BasicObject]
```

## Module insertion questions

- Does the order of module inclusion matter?
- Can including 2 modules that contain the same method name cause trouble?

## Module insertion answers

```
module One
  def test; puts "test One"; end
end
module Two
  def test; puts "test Two"; end
end

class OneClass
  include One
end
class TwoClass
  include One
  include Two
end

p OneClass.ancestors # => [OneClass, One, Object, Kernel, BasicObject]
p TwoClass.ancestors # => [TwoClass, Two, One, Object, Kernel, BasicObject]
TwoClass.new.test # => test Two
```