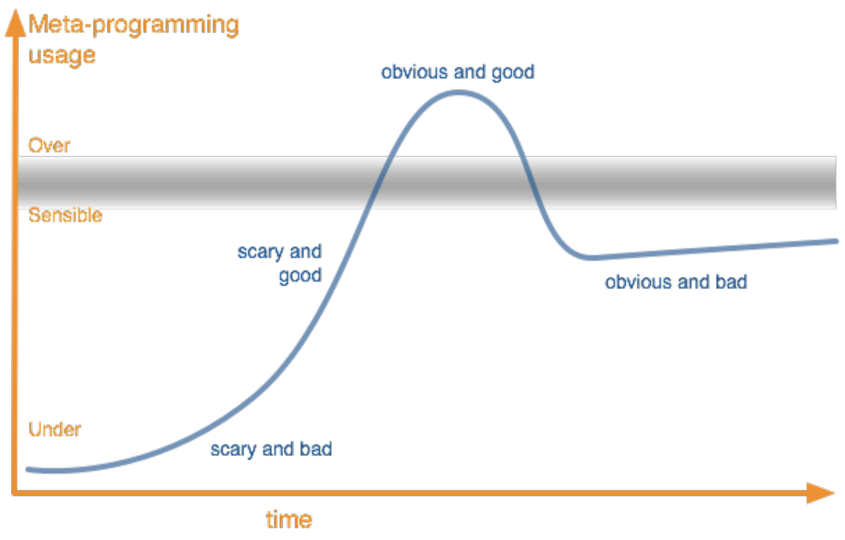
January 7, 2014

SO META: Joys and Sorrows of Metaprogramming

Hugo and James, pairing



- Scary and Bad: People are wary of meta-programming and don' use it much
- Scary and Good: people begin to see the value of meta-programming but are still uncomfortable with using it.
- Easy and Good: as people get comfortable they begin to use it t much, which can complicate the code-base.
- Easy and Bad: people are wary of meta-programming and realiz that it's very useful in small doses.

:send

- Invokes the method identified by :symbol, passing i any additional arguments specified.

```
class Trip < ActiveRecord::Base  # Crappy backport of 'where' to rails 2.3
  def self.where(attribute_map)
    keys = attribute_map.keys
    values = keys.map{|key|attribute_map[key]}
    composed_name = keys.map(&:to_s).join('_and_')
    send("find_by_#{composed_name}", values)
  end
endTrip.where(:origin => 'ORD', :destination => 'SFO')
```

:send benefits

- Allows for dynamic method invokation.

```
class Taxi
  def calculate_fare_for(distance)
    send("#{time_of_day}_rate", distance)
  end
  def rush_hour_rate
    # ...
  end
  def evening_rate
    # ...
  end
  def afternoon_rate
    # ...
  end
end
```

:send benefits

- Allows for dynamic method invokation.

```
class Image
  def execute_method_sequence methods
    methods.each{ |method| self.send(method) }
  end
  def rotate_90
    # ...
  end
  def h_flip
    # ...
  end
end

kitty1, kitty2 = Image.new, Image.new

messed_up_kitty = kitty1.execute_method_sequence ['rotate_90', 'h_flip']
different_kitty = kitty2.execute_method_sequence ['h_flip', 'rotate_90']
```

other :send benefits

- Simplifies creating your own DSLs (Domain-specific languages)
- A 'lighter' version of :eval limited to method calls within an object
- Ability to externally invoke private methods can hel in testing. (!)

:send catches

- Difficult to determine which methods need to exist for :send to work
- Difficult to follow the code path; you may end up debugging more often. :(
- When :send is invoked with user input, may cause security issues.
- Difficult to refactor!

when should i use/ not use :send?

ok excuse

- Only use this if you can't know which method you'd like to call at coding time. See the Timeout example

bad excuses

- Too lazy to type
- Don't want to call two methods in a row

```
args = #...
method_list = ['validate', 'send_email_with']
method_list.each{ |method| send method, args }
# :(
```

opening classes, monkey-patching

```
class Integer
  def to_roman
    # The answer
    'XLII' # Guaranteed to NOT be random
  end
end

class RSpec::Core::Example
  def passed?
    @exception.nil?
  end
  def failed?
    !passed?
  end
end
```

open classes/modules benefits

- Allows for extensions (e.g. useful methods that are lacking or for gems that are no longer maintained)
- Can save you from being stuck in older versions
- Allows the 'slow' evolution of your system with backports

open classes/modules catche

- Causes many 'WTF' moments. Where is 'insert your concern here' being modified?
- After you monkey-patch a gem, you are probably n going to want to update it.
- Ages very badly with different implementation details.
- Any dependencies you create will be very hard to clean up later.
- Unintentional side-effects. *shudder*

be especially careful when patching core classes

cache money example

```
class Array
  alias_method :count, :size

  def to_hash
    keys_and_values_without_nils = reject { |key, value| value.nil? }
    shallow_flattened_keys_and_values_without_nils = \
             keys_and_values_without_nils.inject([]) \
                   { |result, pair| result += pair }
    Hash[*shallow_flattened_keys_and_values_without_nils]
  end
end
```

- Doesn't seem like a big deal, right?

An example from rails 3.

```ruby
def find_by_attributes(match, attributes, *args)
  conditions = Hash[attributes.map {|a| [a, args[attributes.index(a)]]}]
  result = where(conditions).send(match.finder)

  if match.bang? && result.blank?
    raise RecordNotFound, "Couldn't find #{@klass.name} with " << \
         "#{conditions.to_a.collect {|p| p.join(' = ')}.join(', ')}"
  else
    result
  end
end
# from vendor/bundle/ruby/1.8/gems/activerecord-3.0.20/
# ... /lib/active_record/relation/finder_methods.rb
```

- When we upgraded to rails 3, we started seeing MissingAttribute exceptions when we used joins.
- This was because rails 3 was using Hash[] syntax to initialize models from query results.
- Also, note that :to_hash wasn't named well.

a couple of scenarios:

- If it's the ruby core, don't do it. Please. Just don't.
- If it's someone else's tool, then you should submit a pull request

it's not just your best practice

it's also a matter of those used in the gems in your projects.

hypothetical situation1:

- "We start using this gem, we're not aware that it's overriding any core ruby functionality and we end u suffering for it"

hypothetical situation2:

- "We're using this gem that we KNOW is overriding core functionality. Whether we intended to or not, v started interlacing dependencies in our projects. No we want to REMOVE this gem."

but... how do we monkeypatch the right way

- *crickets*

best practices

given you're going to do this

- Say you want to define *foo* in *obj*
- Ruby has a bunch of great methods like object#respond_to? and module#instance_method
- These can be used to suss out whether obj.foo is already defined.
- If it is you should not feel comfortable monkeypatching your own foo in.
- If it is already defined: maybe you'd like to try arour alias?
- If it's not... ?

use self-deprecating extensions

```
def self.included(klass)
  klass.class_eval do
    alias_method_chain :stub, :save_original
    alias_method_chain :unstub, :restore_original
  end
end

def should_intercept?
  RUBY_VERSION < '1.9' && mocha_will_remove_method?
end
```

- Mocha 0.13.3 has a bug with ruby 1.8.7, so we wrot an extension that will not do anything if we are running ruby 1.9 or above.

mixin your monkeypatches

What is the difference between:

```
class String
  def palindrome?
    self == self.reverse
  end
end

module JimmysPalindromeFinder
  def palindrome?
    return false unless self.respond_to? :reverse
    self == self.reverse
  end
end

class String
  include JimmysPalindromeFinder
end
```

the first case:

```
class String
  def palindrome?
    self == self.reverse
  end
end

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

the second:

```
module JimmysPalindromeFinder
  def palindrome?
    return false unless self.respond_to? :reverse
    self == self.reverse
  end
end

class String
  include JimmysPalindromeFinder
end

String.ancestors
# => [String, JimmysPalindromeFinder, Comparable, Object, Kernel, BasicObjec
```

source location

- For ruby 1.8, we have a gem: "ruby10_source_location"
- For ruby 1.9 and after, source location is included.

```
class String
  def palindrome?
    self == self.reverse
  end
end

"hello world!".method(:palindrome?).source_location
# => [string_extension.rb, 3]
# note this is in the format [__FILE__, __LINE__]
```

james' takeaway

- It is not really possible to be responsible with monkeypatching
- If you're going to do it, it's important to be prudent. Look before you patch!
- Transparency is another important goal. Prefer mixins to simple patches
- Core ruby classes are not a very good target
- Prefer self-deprecating patches
- Name your patches well (and specifically)
- Be careful. And fix the versions of your gems in you Gemfile

hugo's takeaway

around alias

Man-in-the-middle method calls! Mix it in and they will probabl
never know the difference. :)

```ruby
class CookieJar
  alias :old_cookie :cookie
  def cookie
    alert :mom
    old_cookie
  end
end
```

around alias

Man-in-the-middle method calls! Mix it in and they will probabl
never know the difference. :)

```
module DSL
  alias :old_on_page_with :on_page_with

  def on_page_with(*module_names)
    old_on_page_with(*module_names){ |page| @page=page; yield page }
  end

  define_method :step do |step_number, action_name, args={}|

    if step_number.is_a? Fixnum
      example.metadata[:doc_steps].store step_number, action_name
    end

    @page.perform action_name, args
  end
end
```

around alias benefits

- Great for logging method calls
- Also turning method calls into hooks
- Also useful for functionality extensions
- Handling of previously unsupported edge cases

around alias benefits

- Previously unsupported edge cases

```
# this is a patch to accommodate content-editable elements
class Capybara::Selenium::Node
  alias :old_set :set
  def set value
    if native.attribute('isContentEditable')
      #ensure we are focused on the element
      script = <<-JS
        var range = document.createRange();
        range.selectNodeContents(arguments[0]);
        window.getSelection().addRange(range);
      JS
      driver.browser.execute_script script, native
      native.send_keys(value.to_s)
    else
      old_set value
    end
  end
end
```

around alias cons

- Added dimension of misdirection can be confusing maintain.
- If used as in the previous example, you are going tc want to proceed gingerly before updating any affected libraries.

:define_method

- Private method.
- Defines an instance method in self.
- Method parameter can be a Proc or Method object
- If a block is specified, it is used as the method body
  This block is evaluated using instance_eval.

```
class Person
  define_method :say_hello do
    puts "hello"
  end
end

Person.new.say_hello # => "hello"
```

anonymous class objects

aka DIE inheritence

```
class ClasslessObject
  def self.new(attributes)
    clazz = Class.new
      attributes.keys.each do |key|
      value = attributes[key]
      callable = value
      if !value.respond_to?(:call)
        callable = lambda { || value }
      end
      clazz.send(:define_method,key,callable)
    end
    clazz.new
  end
end
```

objects like javascript!

```
object = ClasslessObject.new(
  :value_per_unit => 10,
  :quantity => 3,
  :total => lambda{||value_per_unit*quantity})
object.methods - Object.new.methods
 => ["value_per_unit", "quantity", "total"]

other = ClasslessObject.new(
  :value_per_unit => 15,
  :total => lambda {|number| value_per_unit * number})
other.methods - Object.new.methods
 => ["value_per_unit", "total"]
```

:define_method other uses

- Grow objects.
- We'll explore this later with the class factory.
- Flat scope with closures. (This isn't really within the scope of this talk)

:define_method catches

- If you have a bug, it is more difficult to locate the problem.
- Messages and stack traces are less clear.

:method_missing

- Invoked by Ruby when an object is sent a message cannot handle.
- By default, the interpreter raises an error when thi method is called.
- It is possible to override method_missing to provid more dynamic behavior.
- But if you're going to do this, you might want to...

:method_missing

Example

```
require 'timeout'
class TimeoutWrapper
  def initialize(timeout_in_seconds, target)
    @timeout = timeout_in_seconds
    @target = target
  end
  def method_missing(method_name, *args)
    status = Timeout::timeout(@timeout) {
      @target.send(method_name, args)
    }
  end
end
class ExtraTerrestrial < BasicObject; end
et = ExtraTerrestrial.new
patient_et = TimeoutWrapper.new(5, et)
et.methods # => <Error: undefined method "methods">
patient_et.methods # => [:method_missing, :nil?, :methods, ... ]
```

clean your room first?

- Since ruby 1.9, the easiest way to do this is to defin your class as a subclass of BasicObject.
- BasicObject is at the top of the ruby inheritence tre
- It doesn't even have kernel methods, like puts or sleep in scope.

for all of you fans of ruby <1.

```
# another option
class MyCleanRoom
  instance_methods.each do |m|
     undef_method m unless m.to_s =~ /^__|method_missing|respond_to?/
  end
end
```

:method_missing

Fixed example

```
require 'timeout'
class TimeoutWrapper < BasicObject
  def initialize(timeout_in_seconds, target)
    @timeout = timeout_in_seconds
    @target = target
  end
  def method_missing(method_name, *args)
    status = Timeout::timeout(@timeout) {
      @target.send(method_name, args)
    }
  end
end

et = ExtraTerrestrial.new
patient_et = TimeoutWrapper.new(5, et)
patient_et.phone_home # will invoke et#phone_home method for 5 seconds
```

overrided method_missing

```
class Product < ActiveRecord::Base
 def method_missing method_name, *args
   if(method_name.to_s =~ /^print_/)
     puts send(method_name.to_s[6..-1])
   else
     super
   end
 end
end
p=Product.new(:name => "My Awesome Product")
p.print_name # => "My Awesome Product"
p.send(:print_name) # => "My Awesome Product"
# good, but....
p.respond_to?(:print_name) # => false  (!?)
p.method(:print_name) # => <throws NameError: undefined method> (!?)
# how to fix this!?!?
```

override method_missing, define responds_to_missing

```
class Product < ActiveRecord::Base
 def method_missing method_name, *args
   if(method_name.to_s =~ /^print_/)
     puts send(method_name.to_s[6..-1])
   else
     super
   end
 end
 def respond_to_missing?(method_name, include_private=false)
   !(method_name.to_s =~ /^print_/).nil? || super
 end
end
p=Product.new(:name => "My Awesome Product")
p.print_name # => "My Awesome Product"
p.send(:print_name) # => "My Awesome Product"
p.respond_to?(:print_name) # => true
p.method(:print_name) # => <Method: ...>
```

:method_missing benefits

- Allows an object to "respond" to many methods cheaply.
- Makes creating DSLs a dream.
- Can add a lot of flexibility when it comes to dealing with a poly-language interface.

:method_missing catches

- There is usually a better solution without reflection
- It is VERY easy to overuse.
- Causes insane amounts of 'WTF?!?' moments. :(

eval, instance_eval, module_eval, class_eval

- Evaluate Strings or blocks of Ruby code in the conte of the receiving object.
- You can do just about whatever you want with this.
- Downsides?
- Many.

reasons to be wary of the :eval_family

- Strings of code are not syntax-highlighted.
- Syntax errors within the Strings will not register unt they are evaluated.
- These are "potentially dangerous operations."
- What do we mean by "potentially dangerous operations"?

the ruby security model

$SAFE == 0

- No checking of the use of externally supplied (taint data is performed. This is Ruby's default mode.

$SAFE >= 1

- Ruby disallows the use of tainted data by potentiall dangerous operations.

$SAFE === (2...4)

- Out of scope of this presentation. :P

What can we do to manage security?

- Manage safety! $SAFE= 1 in production is preferabl though some great tools need $SAFE=0 to work. Mostly pry comes to mind.
- Make sure external inputs are marked tainted, ther sanitized before they are trusted.
- Object#taint, Object#tainted?, Object#untaint are methods built just for this purpose.
- Avoid :evaling code that you did not write. It's just prudent.
- But if you must use the eval family...

eval_family best practice:

Use the eval family with positioning information :)

You will get a better stack trace in the case of an exception.

```
eval("puts 'hello world'")
# becomes
eval("puts 'hello world'", binding, __FILE__, __LINE__)

String.module_eval("A=1")
# becomes
String.module_eval("A=1", binding, __FILE__, __LINE__)

"str".instance_eval("puts self")
# becomes
"str".instance_eval("puts self", binding, __FILE__, __LINE__)
```

Credit: Ola Bini https://olabini.com/blog/2008/01/ruby-antipattern-using-eval-without-positioning-information/

eval family summary

- Even then, many programmers avoid the :eval family.
- The :exec family and :send offer more reliable ways of achieving similar ends.
- But if you're going to ... (1) do it safely and (2) add positioning information to your :eval calls!

class_exec, module_exec, instance_exec

- This family allows you to execute block of code within the receiver.
- In many cases, the *_exec family is a great alternative to the eval family.
- While Kernel.eval is a member of the eval family, Kernel.exec is *not* a membe of the exec family.

difference between :send, the :*evals, and the :*_execs

:*_execs	:*evals	:send
Executes a block of ruby code within the reciever	Executes a String of ruby code *or* a block of ruby code within the receiver	Executes a String or symbol as a method call within the receiver
Has access to private attributes and methods	Has access to private attributes and methods	Has access to private methods, but not attributes
Not "potentially dangerous" (will process tainted input at $SAFE>0)	"Potentially dangerous" (will not be able to process tainted input at $SAFE>0)	Not "potentially dangerous". This surprised me, sort of

class factories

- There are many methods in ruby that have significa value when metaprogramming, but whose use can also be used in normal, non-meta ways.
- This isn't a talk about their normal, non-meta ways.
- So let's delve into one final example that's about as big, bad, and meta as it gets.
- A class factory!

backstory

- One day, I wanted to create a game.
- The game was going to have a bunch of baddies tha
  were all fairly generic.
- But I wanted subclasses of baddies that attacked in
  the same way and had similar attributes.
- BUT I didn't want to write classes for each of them.
  The difference between these subclasses was a triv
  matter of attribute settings! How boring!

```
class Baddies < BasicObject
  def self.method_missing(name, args={})
    return Baddy.new_baddy_with_attr(args.merge(name: name.to_s) )
  end

  class Baddy
    def self.new_baddy_subclass_with_attr args
      new_class = Class.new(self) do
        args.each do |attribute, value|
          define_method("#{attribute}" ) { || instance_variable_get "@#{attr
          define_method("#{attribute}=") { |new_value| instance_variable_set
        end
        define_method("initialize") do
          args.each do |attribute, value|
            instance_variable_set("@#{attribute}", value)
          end
        end
      end
      Object.const_set args[:name].classify, new_class
      new_class
    end
  end
end
```

the result

```
Baddies.thugs hp:20, display_name:"THG", attack: crowbar
Baddies.artists hp:15, display_name:"ART", attack: paint_toss
Baddies.meanies hp:28, display_name:"MEA", attack: insult
Baddies.luddites hp:26, display_name:"LUD", attack: primitive_means

level.fill_with([Thug, Artist, Meanie, Luddite])

# now we can fill our level with instances of thugs, artists,
# meanies, and luddites
```

the ultimate best practice \o

- UNIT TESTING!
- When metaprogramming, we should test both
- Which pieces get changed
- How they get changed
- This is very important!

why test?

- If we modify it, we know it's still working.
- Tests are really important in terms of documentatic
- Metaprogramming is often hard to read.

the ultimate best practice \o

- In the previous class factory example, I can test:
- That my class factory is capable of making 'arbitrar classes with 'arbitrary' attributes.
- That my class factory will not inadvertantly monkeypatch or overwrite a pre-existing class (and hopefully it notifies me if I try to do so)
- I can also test that the classes that I make using the class factory meet my expectations.
- Metaprogramming: saves on code, loads up on test

questions?

thank you for coming out!