



Extending Ruby with Class

RailsConf Europe 2008
Tammo Freese
September 4th, 2008



Introduction

- Extending Ruby is very easy, but it gets harder if you care about other extenders.
- Four examples to show problems and fixes:
 - `String#camelize` (new method)
 - `Array#sort!` (extended with logging)
 - `Array#find_by_...` (via `method_missing`)
 - `find_by_...` (reusable, via `method_missing`)



Main Rule

Care About Others.



Example 1: String#camelize

```
class String
  def camelize
    gsub(/(^|_)(.)/) { $2.upcase }
  end
end
```



Is There a Problem?

```
class String
  def camelize
    gsub(/(^|_)(.)/) { $2.upcase }
  end
end
```



Is There a Problem?

- Care about others:
String is used everywhere. What if another extension defines `String#camelize` differently?
 - Strange errors
 - Maybe no error messages, but wrong behaviour
- Fix:
 - Fail (or at least warn) if the method is already defined.
 - Apply this for commonly used classes/methods only!



Fix

```
class Class
  def all_instance_methods
    instance_methods | private_instance_methods
  end
  def assert_not_defined(method)
    return if !all_instance_methods.include?(method.to_s)
    raise "Name Collision for #{self}##{method}"
  end
end
```



Fix

```
class String
  assert_not_defined :camelize
  def camelize
    gsub(/(^|_)(.)/) { $2.upcase }
  end
end
```



Real World Examples

- Violations:
 - `Array#find_all` (Ruby vs Rails 1.? associations)
 - `Array#sum` (Rails vs. Classifier)
 - `Object#metaclass` (RSpec vs Facets)
- Fix:
 - `Builder::XChar` in the Builder library



Workarounds

- If one method handles more cases than the other:
 - Include its library last.
- If the behaviour of the methods can be merged into one:
 - Write your own method that combines both implementations.
- Otherwise:
 - No simple workaround



Example 2

```
class Array
  alias_method :sort_without_time_logging!, :sort!
  def sort!
    time_now = Time.now
    sort_without_time_logging!
    puts "sort!: #{Time.now - time_now} seconds."
  end
end
```



Is There a Problem?

```
class Array
  alias_method :sort_without_time_logging!, :sort!
  def sort!
    time_now = Time.now
    sort_without_time_logging!
    puts "sort!: #{Time.now - time_now} seconds."
  end
end
```



Is There a Problem?

- Care about others:
 - `Array#sort!` normally returns the `Array` itself.
 - `Array#sort!` accepts a block parameter.
- Solution:
 - Let the method return the original return value.
 - Pass the block to the original method.



Fix

```
class Array
  alias_method :sort_without_time_logging!, :sort!
  def sort!(&block)
    time_now = Time.now
    sort_without_time_logging!(&block)
  ensure
    puts "sort!: #{Time.now - time_now} seconds."
  end
end
```



Fix

```
class Array
  alias_method :sort_without_time_logging!, :sort!
  def sort!(*args, &block)
    time_now = Time.now
    sort_without_time_logging!(*args, &block)
  ensure
    puts "sort!: #{Time.now - time_now} seconds."
  end
end
```



Workarounds

- If another library does not pass arguments or return values:
 - Submit a patch.
 - Fix the problems in a local copy of the library.



Example 3: Array#find_by_...

```
class Array
  def method_missing(id, *args)
    raise NameError unless id.to_s =~ /^find_by_(.+)$/

    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```



Is There a Problem?

```
class Array
  def method_missing(id, *args)
    raise NameError unless id.to_s =~ /^find_by_(.+)$/

    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```



Is There a Problem?

- Care about others:
 - Others may rely on the original `method_missing` implementation's behaviour (returning `NameError` or `NoMethodError`)
 - Others may have extended `Object#method_missing`.
- Fix:
 - use `super` to call the original method.



Example 3: Array#find_by_...

```
class Array
  def method_missing(id, *args)
    raise NameError
    return super unless id.to_s =~ /^find_by_(.+)$/

    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```



Is There a Problem?

```
class Array
  def method_missing(id, *args)
    return super unless id.to_s =~ /^find_by_(.+)$/

    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```



Is There a Problem?

- Care about others:
 - Others may have already defined/extended `Array#method_missing`.
 - Our implementation would overwrite it.
- Fix:
 - Instead of calling `super`, use `alias_method`.



Fix

```
class Array
  alias_method :mm_without_find_by, :method_missing
  def method_missing(id, *args, &block)
    return super mm_without_find_by(id, *args, &block)
      unless id.to_s =~ /^find_by_(.+)$/

    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```



Is There a Problem?

```
class Array
  alias _method :mm_without_find_by, :method_missing
  def method_missing(id, *args, &block)
    return mm_without_find_by(id, *args, &block)
      unless id.to_s =~ /^find_by_(.+)$/
    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```



Is There a Problem?

- Care about others:
 - Even if another extension which was loaded first could handle `find_by_...`, our extension would overwrite it.
 - Which should come first?
My opinion: Extend means First Come, First Serve
- Fix:
 - Call the original method first.



Is There a Problem?

```
class Array
  alias_method :mm_without_find_by, :method_missing
  def method_missing(id, *args, &block)
    return mm_without_find_by(id, *args, &block)
  rescue NameError => e
    raise e unless id.to_s =~ /^find_by_(.+)$/
    attr = $1
    find { |elem| elem.__send__(attr) == args.first }
  end
end
```



Is There a Problem?

- There is at least one left, but I'll show the fix in Example 4.



Example 4: Reusable find_by_...

- Make find_by_... reusable for all find methods.
 - Move the method into a module.
 - Override the module's included hook to define method_missing in each class where the module is included.



Example 4

```
class Array module FindBySupport
  alias _method :mm_without_find_by, method_missing
  def method_missing mm_with_find_by(id, *args, &block)
    mm_without_find_by(id, *args, &block)
  rescue NameError => e
    raise e unless id.to_s =~ /^find_by_(.+)$/
    attr = $1
    find { |elem| elem.__send__(attr) == args.first }
  end;end
```



Example 4

```
module FindBySupport
  def self.included(base)
    base.class_eval do
      alias_method :mm_without_find_by, :method_missing
      alias_method :method_missing, :mm_with_find_by
    end; end; end
class Array
  include FindBySupport
end
```



Is There a Problem?

```
def self.included(base)
  base.class_eval do
    alias_method :mm_without_find_by, :method_missing
    alias_method :method_missing, :mm_with_find_by
  end
end
```

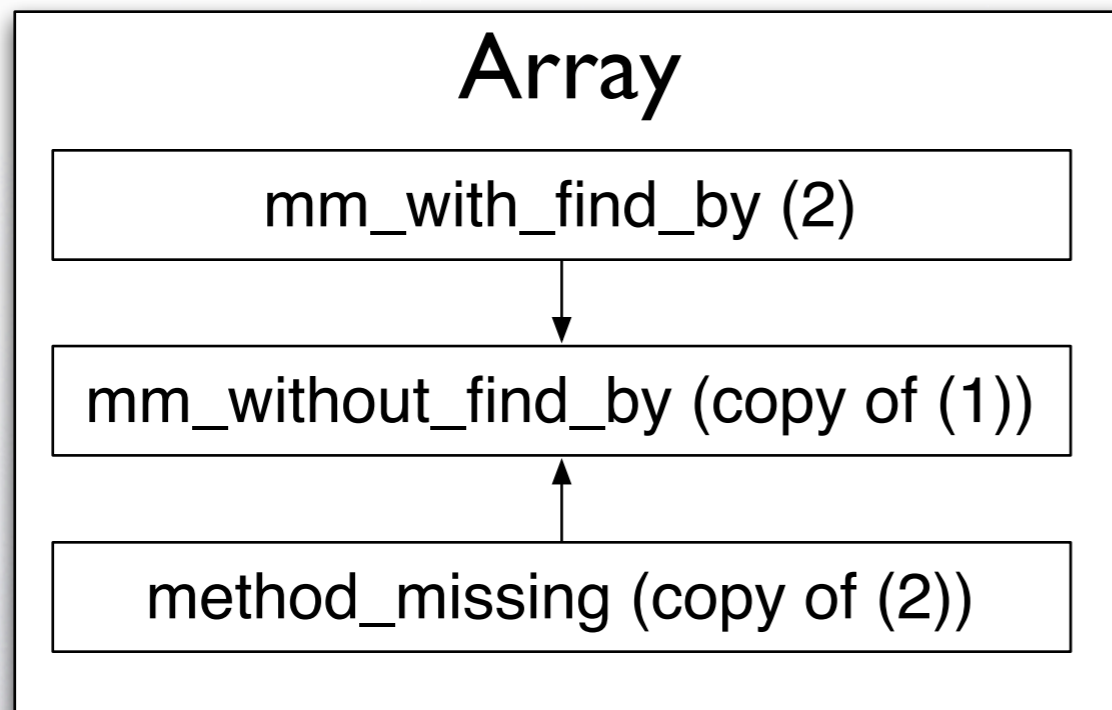
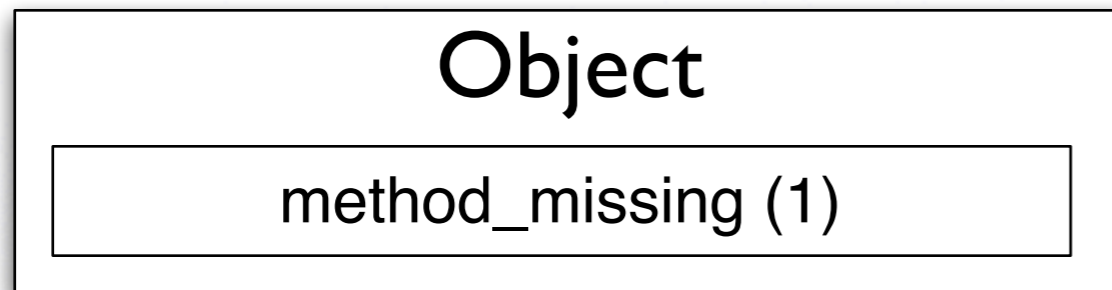


Is There a Problem?

- Care about others:
If another extension extends/redefines `Object#method_missing`, and it is loaded after our extension, its `method_missing` will not be called.
- Reason:
Our first `alias_method` call copies the original `Object#method_missing` (explanation: next slide)



Which Method Calls Which?



method from module

alias_method :mm_without_find_by,
:method_missing

alias_method method_missing,
:mm_with_find_by



Fix

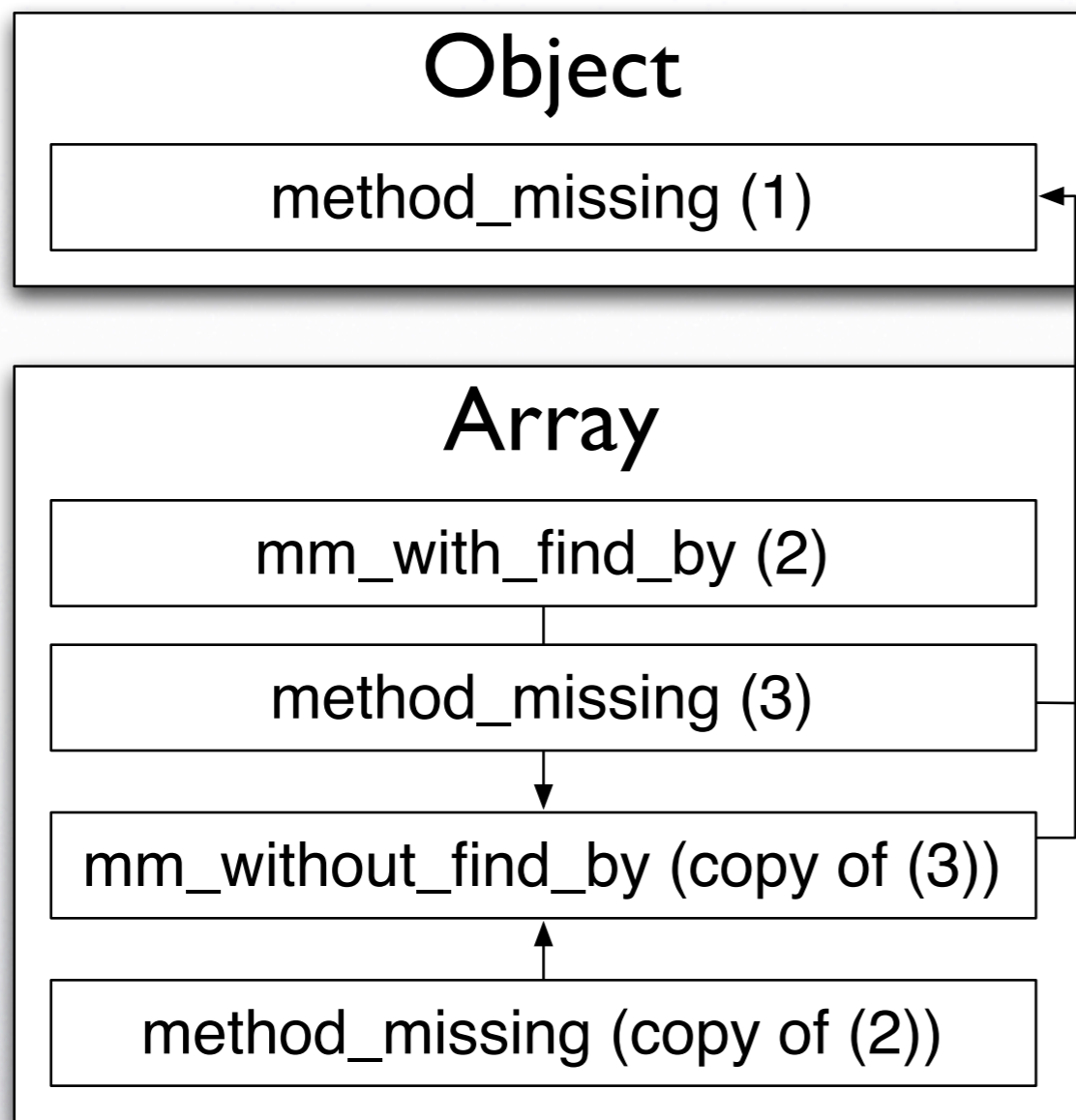
- Solution:

Before the first `alias_method` call, we add `Array#method_missing` and call `super` in it:

```
def self.included(base)
  base.class_eval do
    def method_missing(id, *args); super; end
    alias_method :mm_without_find_by, :method_missing
    alias_method :method_missing, :mm_with_find_by
  end
end
```



Which Method Calls Which?



method from module

method_missing calling super

alias_method :mm_without_find_by,
:method_missing

alias_method method_missing,
:mm_with_find_by



Is There a Problem?

- Fixed one problem, introduced another:
If another extension extends (or redefines) `Array#method_missing`, and is loaded before our extension, our extension would remove it.
- Solution:
Only define `method_missing` if needed.



Fix

```
def self.included(base)
  base.class_eval do
    unless (instance_methods(false) +
            private_instance_methods(false)).
              include?('method_missing')
      def method_missing(id, *args); super; end
    end
    alias_method :mm_without_find_by, :method_missing
    alias_method :method_missing, :mm_with_find_by
  end;end
```



Is There a Problem?

```
def self.included(base)
  base.class_eval do
    unless (instance_methods(false) +
            private_instance_methods(false)).
              include?('method_missing')
      def method_missing(id, *args); super; end
    end
    alias_method :mm_without_find_by, :method_missing
    alias_method :method_missing, :mm_with_find_by
  end;end
```

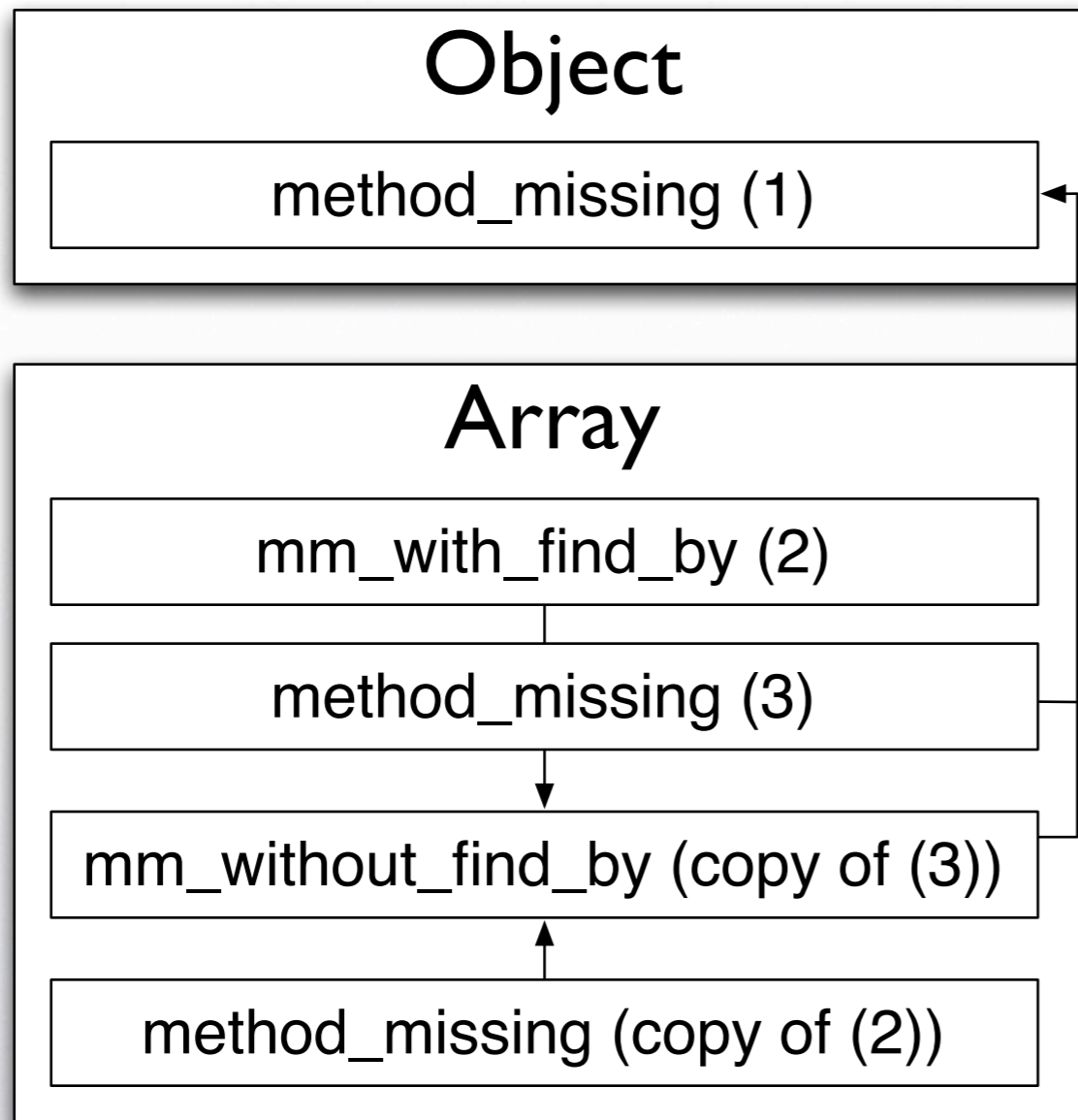


Is There a Problem?

- Care about others:
 - Extending/redefining `mm_with_find_by` has no effect.
- Reason:
 - Our second `alias_method` call copies `mm_with_find_by` (explanation: next slide)



Which Method Calls Which?



method from module

method_missing calling super

alias_method :mm_without_find_by,
:method_missing

alias_method method_missing,
:mm_with_find_by



Fix

- Instead of using `alias_method`, redefine `method_missing` and call `mm_with_find_by` in it.

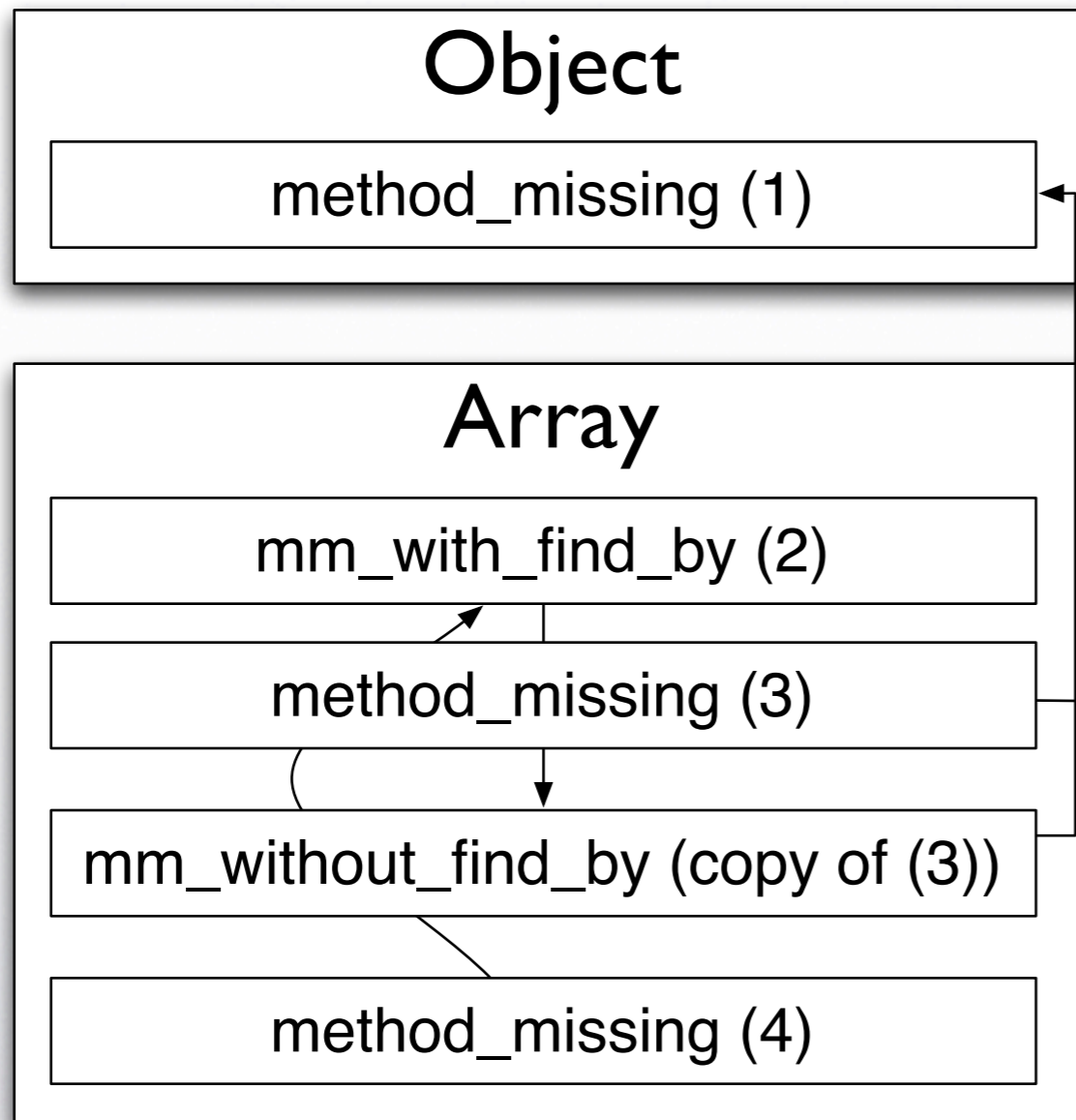


Fix

```
def self.included(base)
  base.class_eval do
    unless method_defined? ... end
    alias_method :mm_without_find_by, :method_missing
alias_method :method_missing, :mm_with_find_by
    def method_missing(id, *args, &block)
      mm_with_find_by(id, *args, &block)
    end
  end
end
```



Which Method Calls Which?



method from module

method_missing calling super

alias_method :mm_without_find_by,
:method_missing

method_missing calling
mm_with_find_by



Real World Examples

- Violations:
 - Builder's `method_added` implementations for BlankSlate (defining `Module#method_added` afterwards has no effect)
 - Ruby on Rails's `alias_method_chain` (overwriting `..._with_...` methods has no effect)
- Builder: Patch submitted, not included yet.
- Ruby on Rails: Patch in Issue #285 (wontfix?!)



Workaround for Builder

- Problem Illustration:

We would like to log all added methods:

```
class Module
  def method_added(id); puts id; end
end
```

- If you already included builder, the code above will not have any effect.
- Workaround:
Ensure that your code is required before builder.



Workaround for alias_method_...

- Problem Illustration: You would like to disable the benchmarking of ActionController::Base:

```
class ActionController::Base
  def perform_action_with_benchmark(*args, &block)
    perform_action_without_benchmark(*args, &block)
  end
end
```

- The code above does not work, since overriding ..._with_... methods has no effect!



Expected

perform_action



perform_action_with_caching



perform_action_without_caching



perform_action_with_rescue



perform_action_without_rescue



perform_action_with_benchmark



perform_action_without_benchmark



perform_action_with_filters



perform_action_without_filters (copy of original perform_action)



Actual

perform_action (copy of perform_action_with_caching)



perform_action_without_caching (copy of perform_action_with_rescue)



perform_action_without_rescue (copy of perform_action_with_benchmark)



perform_action_without_benchmark (copy of perform_action_with_filters)



perform_action_without_filters (copy of original perform_action)



Workaround for `alias_method_...`

- What we know:
 - `..._without_...` methods are called
 - `perform_action_without_rescue` is a copy of `perform_action_with_benchmarks`
- So we just redefine `perform_action_without_rescue` instead of `perform_action_with_benchmarks`.



Workaround for `alias_method_...`

```
class ActionController::Base
  def perform_action_with_benchmark
    perform_action_without_rescue(*args, &block)
    perform_action_without_benchmark(*args, &block)
  end
end
```



Wrap-Up

Care About Others.



The Last Slide

- Please give feedback on the conference page.
- If you liked this talk,
please recommend me on workingwithrails.com .
- Questions?
 - either ask now,
 - or send me an e-mail:
tammo@tammofreese.de