In this lecture, we will discuss…

- Ghost methods

Nonexistent (Ghost) Methods

- **Question:** If a method is invoked and it's not found, was it really called at all?

```
irb(main):001:0> class SomeClass; end
=> nil
irb(main):002:0> some_class = SomeClass.new
=> #<SomeClass:0x007fb552ae1c80>
irb(main):003:0> some_class.i_dont_exist
NoMethodError: undefined method `i_dont_exist' for #<SomeClass:0x007fb552ae1c80>
        from (irb):3
        from /Users/kalmanhazins/.rbenv/versions/2.1.2/bin/irb:11:in `<main>'
```

method_missing… method

- Ruby looks for the method invoked in the class to which it belongs
- Then it goes up the ancestors tree (classes and modules)
- If it still doesn't find the method, it calls `method_missing` method
- The default `method_missing` implementation throws `NoMethodError`

Overriding method_missing

- Since `method_missing` is just a method, you can easily override it
- You have access to
  - Name of the method called
  - Any arguments passed in
  - A block if it was passed in

Overriding method_missing

```
class Mystery
  # no_methods defined
  def method_missing (method, *args)
    puts "Looking for..."
    puts "\"#{method}\" with params (#{args.join(',')}) ?"
    puts "Sorry... He is on vacation..."
    yield "Ended up in method_missing" if block_given?
  end
end

m = Mystery.new
m.solve_mystery("abc", 123123) do |answer|
  puts "And the answer is: #{answer}"
end

# => Looking for...
# => "solve_mystery" with params (abc,123123) ?
# => Sorry... He is on vacation...
# => And the answer is: Ended up in method_missing
```

Ghost Methods

- `method_missing` gives you the power to "fake" the methods
- Called "ghost methods" because the methods don't really exist
- Ruby's built-in classes use `method_missing` and dynamic methods all over the place…

Struct and OpenStruct

- `Struct`
  - Generator of specific classes, each one of which is defined to hold a set of variables and their accessors (“Dynamic Methods”)
- `OpenStruct`
  - Object (similar to `Struct`) whose attributes are created dynamically when first assigned (“Ghost methods”)

Struct and OpenStruct

```
Customer = Struct.new(:name, :address) do # block is optional
  def to_s
    "#{name} lives at #{address}"
  end
end
jim = Customer.new("Jim", "-1000 Wall Street")
puts jim # => Jim lives at -1000 Wall Street

require 'ostruct' # => need to require ostruct for OpenStruct

some_obj = OpenStruct.new(name: "Joe", age: 15)
some_obj.sure = "three"
some_obj.really = "yes, it is true"
some_obj.not_only_strings = 10
puts "#{some_obj.name} #{some_obj.age} #{some_obj.really}"
# => Joe 15 yes, it is true
```

So Now, Instead Of This…

```
require_relative 'store'
class ReportingSystem
  def initialize
    @store = Store.new
  end
  def get_piano_desc
    @store.get_piano_desc
  end
  def get_piano_price
    @store.get_piano_price
  end

  # ...many more simimlar methods...
end

rs = ReportingSystem.new
puts "#{rs.get_piano_desc} costs #{rs.get_piano_price.to_s.ljust(6, '0')}"
# => Excellent piano costs 120.00
```

…We Can Do This!

```
require_relative 'store'

class ReportingSystem
  def initialize
    @store = Store.new
  end
  def method_missing(name, *args)
    super unless @store.respond_to?(name)
    @store.send(name)
  end
end

rs = ReportingSystem.new
puts "#{rs.get_piano_desc} costs #{rs.get_piano_price.to_s.ljust(6, '0')}"
# => Excellent piano costs 120.00
```

Why do we care to use "super" here?

method_missing and Performance

- Since the invocation is indirect, could be a little slower
- Most of the time, it will probably not matter too much
- If it does, you can consider a hybrid approach
  - Define a real method from inside `method_missing` after an attempted “call”

Summary

- ✧ Ghost methods allow you to call methods as if they are there even though they are not
- ✧ Method behavior can be defined at runtime, for example based on database columns existing or not!

What's Next?

- ✧ Introduction to Active Record