JOHNS HOPKINS
WHITING SCHOOL OF ENGINEERING

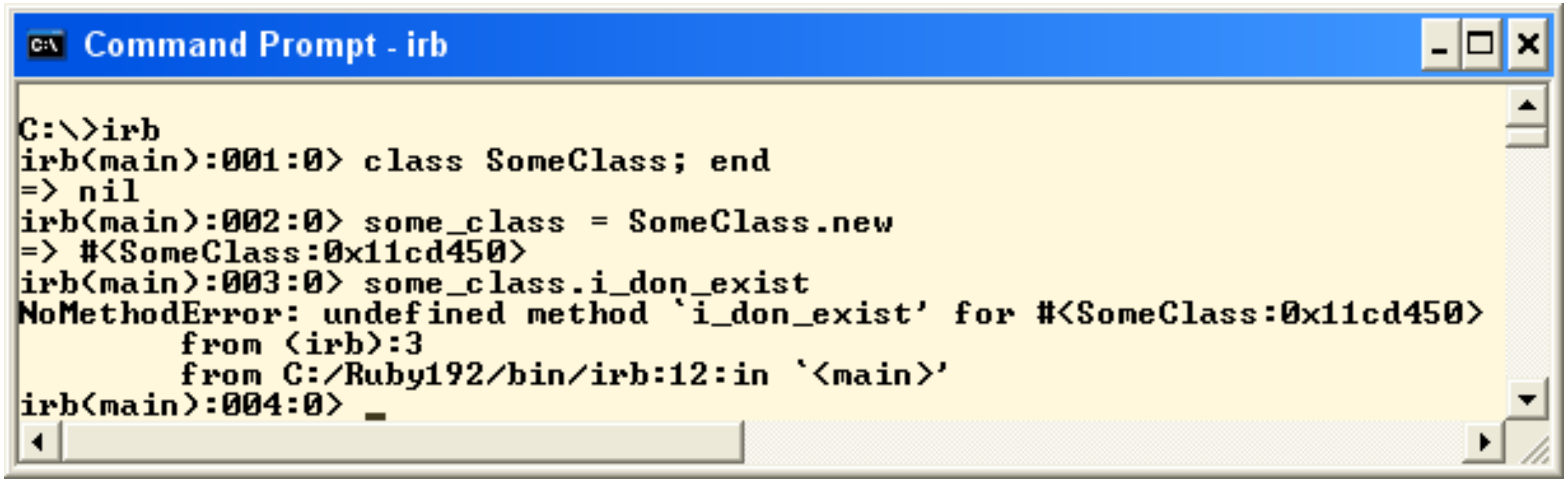
Engineering for Professionals

EXPERTISE APPLIED.

Ghost Methods

Non-existent (ghost) methods

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



```
C:\>irb
irb(main):001:0> class SomeClass; end
=> nil
irb(main):002:0> some_class = SomeClass.new
=> #<SomeClass:0x11cd450>
irb(main):003:0> some_class.i_don_exist
NoMethodError: undefined method `i_don_exist' for #<SomeClass:0x11cd450>
        from (irb):3
        from C:/Ruby192/bin/irb:12:in `<main>'
irb(main):004:0> _
```

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` 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 Method”)
- `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
```

MyOpenStruct

```
# How hard would it be to write our own OpenStruct?
class MyOpenStruct
  def initialize
    @attributes = {} # store values in hash internally
  end
  def method_missing(name, *args)
    attribute = name.to_s
    if attribute =~ /=$/ # ends with '='
      # take off the '=' and shove into hash
      @attributes[attribute.chop] = args[0]
    else
      @attributes[attribute] # extract the value from the hash
    end
  end
end
person = MyOpenStruct.new; person.name = "Frank"; puts person.name # => Frank
```

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?

Overriding respond_to?

- `missing_method` code "works", but…
- The methods don't really exist ("Duh! They are not real methods…")
- But what if someone asks our class if it supports those methods?
- The answer will be a "NO"
- Therefore - override `respond_to?` as well

Overriding respond_to?

```
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
  def respond_to? (name)
    @store.respond_to?(name)
  end
end

rs = ReportingSystem.new puts
rs.respond_to?(:get_piano_desc) # => true
```

method_missing or is it?

- Existing methods?
  - If the method being called exists in the ancestor tree - the call will not end up in the `method_missing` method
- Consider removing methods using `undef_method` or extending from `BasicObject` which has much less methods than `Object`, a.k.a. “Blank Slate” approach

Builder example

```ruby
# gem install builder (make sure to get version 3.0.0)
require 'builder'

xml = Builder::XmlMarkup.new(target: STDOUT, indent: 2)

xml.university(name: "JHU") {
  xml.class('Ruby on Rails')
  xml.class("Java on Grails")
}

# => <university name="JHU">
# =>   <class>Ruby on Rails</class>
# =>   <class>Java on Grails</class>
# => </university>

# Wait, but isn't class() a built-in method?
```

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 - maybe consider a hybrid approach
  - Define a real method from inside `method_missing` after an attempted “call”