In this lecture, we will discuss…

- Defining methods dynamically

Defining Methods Dynamically

- A.k.a. "*Dynamic Method*"
- Not only can you call methods dynamically (with `send`) - you can also define methods dynamically
- `define_method :method_name` and a block which contains the method definition
- Defines an instance method for the class

Dynamic Method Example

```ruby
class Whatever
  define_method :make_it_up do
    puts "Whatever..."
  end
end

whatever = Whatever.new
whatever.make_it_up # => Whatever...
```

So Now, Instead of This…

```ruby
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 similar 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!

Extracts product name

```
require_relative 'store'
class ReportingSystem

  def initialize
    @store = Store.new
    @store.methods.grep(/^get_(.*)_desc/) { ReportingSystem.define_report_methods_for $1 }
  end

  def self.define_report_methods_for (item)
    define_method("get_#{item}_desc") { @store.send("get_#{item}_desc")}
    define_method("get_#{item}_price") { @store.send("get_#{item}_price")}
  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
```

Improved Reporting System

- ✧ No more duplication
  - Now, you don't have to write all of those repetitive methods anymore
- ✧ **Bonus:** If someone adds a new item to the `Store` class - your `ReportingSystem` class already "knows about it" (as long as the same method naming pattern is adhered to)

Summary

- ✧ Defining methods dynamically can dramatically reduce the amount of code that needs to be written

What's Next?

- ✧ Ghost methods