Dynamic Methods

Dynamic Methods

- In static languages, like Java, the compiler requires you to define all the methods upfront
- In dynamic languages, such as Python and Ruby, methods don't have to be predefined - they need to only be "found" when invoked
- Advantages/Disadvantages?
  - Compiler can find bugs easier
  - Define methods just to make compiler happy?

Reporting system example

- Say you have a `Store` class
- Description and price of store products
- You are tasked with building a reporting system that can generate reports for different items in the store

Reporting system example

```
class Store
  def get_piano_desc
    "Excellent piano"
  end
  def get_piano_price
    120.00
  end
  def get_violin_desc
    "Fantastic violin"
  end
  def get_violin_price
    110.00
  end

  # ...many other similar methods...
end
```

Reporting system example (cont.)

```
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
```

Calling methods dynamically

- So far, we have seen how to call methods using the dot notation `obj.method`
- It turns out, there is another way to call a method in Ruby - using the `send` method
- 1st parameter is the method name/symbol; the rest (if any) are method arguments
- Send?
  – Think of it as sending a message to an object

Calling methods dynamically

```
class Dog
  def bark
    puts "Woof, woof!"
  end
  def greet(greeting)
    puts greeting
  end
end

dog = Dog.new
dog.bark # => Woof, woof!
dog.send("bark") # => Woof, woof!
dog.send(:bark) # => Woof, woof!
method_name = :bark
dog.send method_name # => Woof, woof!

dog.send(:greet, "hello") # => hello
```

Dynamic Dispatch - Advantages

- Advantages to dynamic method calling, a.k.a. "*Dynamic Dispatch*"
  - Can decide at runtime which methods to call
- The code doesn't have to find out till runtime which method it needs to call

Dynamic Dispatch example (cont.)

```
require 'yaml'
some_yaml = %{
name: John
age: 24
}

class Person; attr_accessor :name, :age; end

person = Person.new
props = YAML::load some_yaml
p props # => { "name"=>"John", "age"=>24 }
props.each {|key, value| person.send("#{key}=", value)}

p person # => #<Person:0x11a7b48 @name="John", @age=24>
```

CAUTION: More about send

- Perhaps the most surprising thing about `send` is that it lets you call object's `private` methods!
- Ruby 1.9 experimented with restricting `send` to only be able to call `public` methods (not to break encapsulation), but reverted back
- There is now also a `public_send` method that is only able to call `public` methods

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
- When executed within a class - will define 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

```
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!

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 silly 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)