**Metaprogramming and Reflection**

**Goals**:

* Learn how to use send
* Learn how to use define\_method
* Learn how to use method\_missing

**send and define\_method**

One of the powers of Ruby is **reflection** (also called **introspection**): the ability for a program to examine itself.

For starters, we can ask an object what methods it will respond to:

obj = Object.new

obj.methods

=> [:nil?, :===, :=~, :!~, :eql?, :hash, :<=>, :class, ...]

Object#methods returns an array of symbols, each the name of a method that can be sent to the object. This is helpful for debugging, but not super useful in production code.

More significantly, we can call a method by name:

[].send(:count) # => 0

When is something like send useful? Why not just call the method the normal way? Well, using send lets us write methods like this:

def do\_three\_times(object, method\_name)

3.times { object.send(method\_name) }

end

class Dog

def bark

puts "Woof"

end

end

dog = Dog.new

do\_three\_times(dog, :bark)

We can even define new methods dynamically with define\_method:

class Dog

# defines a class method that will define more methods; this is

# called a \*\*macro\*\*.

def self.makes\_sound(name)

define\_method(name) { puts "#{name}!" }

end

makes\_sound(:woof)

makes\_sound(:bark)

makes\_sound(:grr)

end

dog = Dog.new

dog.woof

dog.bark

dog.grr

**A couple notes:**

* The code inside Dog class is executed at the time Ruby defines the Dog class. makes\_sound is called at class definition time, **not** each time a new Dog object is created.
* That makes sense, because the work of makes\_sound sets up an instance method to be shared by **all** Dog objects. It's not instance-specific.
* Inside the definition of the Dog class, makes\_sound knows to call the class method because self == Dog here.
* In the makes\_sound macro, self == Dog because this is a Dog class method. define\_method is implicitly called on Dog, adding a new method named name. The block is the code to run when the method is (later) called on an instance of Dog.

We don't write macros every single day, but they are frequently quite useful. Some of the most famous macro methods are:

* attr\_accessor: defines getter/setter methods given an instance variable name.
* belongs\_to/has\_many: defines a method to perform a SQL query to fetch associated objects.

**method\_missing**

When a method is called on an object, Ruby first looks for an existing method with that name. If no such method exists, then it calls the Object#method\_missing method. It passes the method name (as a symbol) and any arguments to #method\_missing.

The default version simply raises an exception about the missing method, but you may override #method\_missing for your own purposes:

class T

def method\_missing(\*args)

p args

end

end

T.new.adfasdfa(:a, :b, :c) # => [:adfasdfa, :a, :b, :c]

Here's a simple example:

class Cat

def say(anything)

puts anything

end

def method\_missing(method\_name)

method\_name = method\_name.to\_s

if method\_name.start\_with?("say\_")

text = method\_name[("say\_".length)..-1]

say(text)

else

# do the usual thing when a method is missing (i.e., raise an

# error)

super

end

end

end

earl = Cat.new

earl.say\_hello # puts "hello"

earl.say\_goodbye # puts "goodbye"

Using method\_missing, we are able to "define" an infinite number of methods; we allow the user to call any method prefixed say\_ on a Cat. This is very powerful; it isn't possible to do this using define\_method itself.

However, overriding method\_missing can result in difficult to understand/debug to code, and should not be your first resort when attempting metaprogramming. Only if you want this infinite expressability should you use method\_missing; prefer a macro if the user just wants to define a small set of methods.

**An Advanced Example: Dynamic Finders**

What if we overrode #method\_missing in ActiveRecord::Base to work for #find\_by\_\* methods, so we could do the following:

User.find\_by\_first\_name\_and\_last\_name("Ned", "Ruggeri")

User.find\_by\_username\_and\_state("ruggeri", "California")

We could do this by parsing the "missing" method name and combining the column names (separated by ands) with the given arguments. It might look something like this:

class ActiveRecord::Base

def method\_missing(method\_name, \*args)

method\_name = method\_name.to\_s

if method\_name.start\_with?("find\_by\_")

# attributes\_string is, e.g., "first\_name\_and\_last\_name"

attributes\_string = method\_name[("find\_by\_".length)..-1]

# attribute\_names is, e.g., ["first\_name", "last\_name"]

attribute\_names = attributes\_string.split("\_and\_")

unless attribute\_names.length == args.length

raise "unexpected # of arguments"

end

search\_conditions = {}

attribute\_names.length.times do |i|

search\_conditions[attribute\_names[i]] = args[i]

end

# Imagine search takes a hash of search conditions and finds

# objects with the given properties.

self.search(search\_conditions)

else

# complain about the missing method

super

end

end

end

**NB:** Dynamic finders were actually a feature of Rails until just recently. Rails 4.2 deprecated (supported, but didn't recommend) dynamic finders, and as of Rails 5, they are no longer supported. Although, they are quite handy, they tend to lead to overly verbose code and are not very performant. For these reasons we also recommend against using them. Check out [this](http://chrisholtz.com/blog/in-defense-of-dynamic-finders) blog post if you'd like to learn more.

**Type Introspection**

So far we focused on finding, defining, and calling methods at runtime. We can also find class information:

"who am i".class # => String

"who am i".is\_a?(String) # => true

I commonly use Object#class when debugging or using pry to see what kind of thing I'm dealing with, so that I can then know what class to look up the documentation for.

Here we can see that even classes are objects in Ruby:

Object.is\_a?(Object) # => true

# such meta, wow

Deep. Let's dig deeper:

Object.class # => Class

Okay, all classes are instances of a Class class.

Class.superclass # => Module

Class.superclass.superclass # => Object

Classes are types of Modules (not important), which are Objects. In Ruby everything is an Object, even Classes!

To summarize: Object is of type Class, which is a subclass of Object itself. Whoa!

**Methods with Varying Argument Types**

Say we have written a method perform\_get that fetches a resource over the internet. As a convenience to the user, we'd like perform\_get to take either a String, which is the literal URL to fetch, or a hash, with the URL broken into parts

perform\_get("http://www.google.com/+")

perform\_get(

:scheme => :http,

:host => "www.google.com",

:path => "/+"

)

In the case where we give perform\_get a hash, it's going to need to do some extra work to construct the URL to get. How might this work? Perhaps like so:

def perform\_get(url)

if url.is\_a?(Hash)

# url is actually a hash of url options, call another method

# to turn it into a string representation.

url = make\_url(url)

end

# ...

end

This is a quite common trick used by library writers to make their methods much more flexible. You may not write a method like this often, but as you grow more experienced, this kind of trick will come in handy from time to time.