CHAPTER 23

Use method_missing to Build Flexible APIs

Programmers in the business of producing systems for end users spend a lot of time thinking about interfaces. Should you do a plain Google-style search page or spiff things up with all kinds of AJAXy doo-dads? Should you let them pick the car model with a menu or a pull-down or a bunch of radio buttons? Should you support both drag and drop? When end users are involved it is critical that you think very carefully about interface issues. I'd like to think that much of this is simple professionalism: We work hard to build the best possible system because that's what we do. There is, however, the uncomfortable fact that end users tend to make their opinions pretty clear. Screw up an end user interface and you won't have to wait long or listen hard to experience their dissatisfaction, be it actual shouting or the ominous sound of feet going elsewhere.

The interfaces that programmers deal with—the APIs—have a lot more to do with method names and argument lists than they do with menus and radio buttons. But quality matters there too. Build a better API and you make the job of the coder trying to use that API easier. In this chapter we will explore how you can use `method_missing` to create extremely intuitive APIs, APIs where your users can, with some constraints, make up method names and just have them work. Who knows, you might save your users so much time that they will be able to build a better interface for *their* users.

Building Form Letters One Word at a Time

Imagine that your `Document` class is now being used in many different departments in your company, with whole teams of programmers building ever more interesting applications around it. One of these groups is involved in creating form letters, the kind we are all too familiar with, letters that proclaim YOU MAY ALREADY BE A WINNER or that Whiter Teeth Are Just a Phone Call Away!!

Since junk mail is a volume business, the programmers behind it are looking to you to create specialized code to make it easier to generate large numbers of letters. To that end, you come up with a `Document` subclass:

```
class FormLetter < Document
  def replace_word( old_word, new_word )
    @content.gsub!( old_word, "#{new_word}" )
  end
end
```

The only new feature of the `FormLetter` class is the `replace_word` method, which will run through the text of the letter, replacing one string with another. The idea is that you can start with a very generic template document containing placeholders like `FIRSTNAME` and `LASTNAME` and then swap in real first and last names with some simple code:

```
offer_letter = FormLetter.new( "Special Offer", "Acme Inc",
%q{
    Dear Mr. LASTNAME

    Are you troubled by the heartache of hangnails?
    ...

    FIRSTNAME, we look forward to hearing from you.
})

offer_letter.replace_word( 'FIRSTNAME', 'Russ' )
offer_letter.replace_word( 'LASTNAME', 'Olsen' )
```

Looking at the code, you realize that virtually all of the form letters will need to replace `FIRSTNAME` and `LASTNAME`. To make this common case easier, you add a couple of convenience methods:

```
class FormLetter < Document

  def replace_word( old_word, new_word )
    @content.gsub!( old_word, "#{new_word}" )
  end

  def replace_firstname( new_first_name )
    replace_word( 'FIRSTNAME', new_first_name )
  end

  def replace_lastname( new_last_name )
    replace_word( 'LASTNAME', new_last_name )
  end
end
```

You send your handiwork off to the form letter people and go back to your ordinary work. The next day the junk mail folks are back to tell you that (a) they love the FormLetter class, and (b) could you please just tweak it a bit?

It seems that the programmers using the FormLetter class would like a few more convenience methods along the lines of replace_firstname and replace_lastname. Specifically, they would like replace_gender, a method that would swap out GENDER for "sir" or "madam." Oh, and also replace_streetnumber, replace_streetname, replace_city, replace_state, replace_country, replace_zipcode, replace_occupation, replace_age, and so on. For a simple little class, FormLetter seems like it's turning into a full-time job spent writing an inconveniently large number of convenience methods.

Magic Methods from method_missing

But do you actually have to write all of those methods? Not really: All you need is method_missing. Think about what method_missing does: It lets you know that someone is trying to call a method on your object, a method that does not actually exist. In the last couple of chapters we've seen how you can use method_missing to deal with programming errors and to help with delegation. Another thing you can do with method_missing is try to figure out what the user is asking you to do and actually *do it*. What if, whenever someone called a nonexistent method on one of your FormLetter objects, you looked at the method name to see whether you could make sense out of it? If you can, you do the right thing. If not, there is always NameException to raise.

Here is that idea in code:

```
class FormLetter < Document
  def replace_word( old_word, new_word )
    @content.gsub!( old_word, "#{new_word}" )
  end

  def method_missing( name, *args )
    string_name = name.to_s
    return super unless string_name =~ /^replace_\w+/
    old_word = extract_old_word(string_name)
    replace_word( old_word, args.first )
  end

  def extract_old_word( name )
    name_parts = name.split('_')
    name_parts[1].upcase
  end
end
```

This `method_missing` implementation deduces what it should do from the name of the method being called, If the method looks like `replace_<<some word>>`, then the `method_missing` above will extract the word from the name of the method, convert it to all uppercase, and call `replace_word`. So if you call `replace_gender( 'Dude' )`, then the `method_missing` in the code above will end up calling `replace_word( 'GENDER', 'Dude' )`.

If, on the other hand, the method being called doesn't look anything like `replace_<<some word>>`, then you have a legitimate screwup. In that case, `method_missing` bails out by calling `super`, which will probably result in a `NameException`.

The best part of the new `FormLetter` class is no longer having to write any of the thousand or so convenience methods that the junk mail folks requested. In the same way that delegating `method_missing` in the last chapter enabled us to put a wrapper around any object—with any number of methods—the handful of `method_missing` lines in this latest version of `FormLetter` lets you create an infinite number of convenience methods, everything from `replace_firstname` to `replace_lastcarmodelbought`.

This variation on `method_missing` is sometimes called **magic methods**, since users of the class can make up method names and, as long as the names comply with the rules coded into `method_missing`, the methods will just magically work.

It's the Users That Count—All of Them

Now, a skeptic might observe that neither replace_firstname nor replace_lastcarmodelbought have added any particularly new capability to the FormLetter class.[1] These two methods, however they are implemented, simply expose an existing feature of the FormLetter class in a slightly different package. So why should we bother?

We bother because our users asked us to bother. If you reread my tale of intrigue and junk mail, you'll see that it was the coders who were using the FormLetter class that asked for the convenience methods. Those methods make the code generating the form letters cleaner and easier for the people who count—the programmers who need to deal with it. One of the key values of the Ruby programming culture is that the look of the code matters. It matters because the people who use the code, read the code, and maintain the code matter. Good software engineering is all about making everyone's job easier, not just because we want to go home on time but because we all want to turn out the best possible end product. So we add convenience methods, build method_missing methods, and go to enormous lengths to make our APIs easy to use because programmers with easy-to-use APIs tend to have the time to craft easy-to-use—and working—systems.

Staying Out of Trouble

One nice thing about using method_missing to build a flexible API, as opposed to using it for delegation, is that there is much less danger of a made-up method name colliding with a real method on the object. After all, you are making up the naming convention, so you can avoid any obvious name conflicts. You do have to be on your guard, however, because unfortunate naming collisions can still slip in. For example, in our method_missing powered FormLetter class, we cannot replace the word WORD in our document. If we try:

```
letter = FormLetter.new( 'Example', 'Acme', 'The word is WORD' )
letter.replace_word( 'Abracadabra' )
```

1. A real skeptic might observe that the whole FormLetter project is simply furthering the cause of junk mail and is therefore the work of the devil, but that's another issue.

We get a strange error:

```
#<ArgumentError: wrong number of arguments (1 for 2)>
```

The problem is that there is an actual `replace_word` method in the `FormLetter` class; it's the method that does the real work of changing the text. Since there is a real method there, it and not `method_missing` gets called and blows up. The real `replace_word` method takes two arguments, not one. The solution here is pretty easy: Simply rename the real method or devise a different naming convention for the magic methods. The deeper lesson is that you need to watch out for this sort of thing.

You also need to be aware of the likelihood that using `method_missing` will muck up the `respond_to?` method. Every `Object` instance includes a method called `respond_to?` which should return true if the object in question has a particular method. For example, if `doc` is an instance of `Document`, then `doc.respond_to?(:words)` will return true, while `doc.respond_to?(:abuse)` will return false, since there `Document` has a `words` method but no `abuse` method. The problem is that the default implementation of `respond_to?` only knows about the real methods; it has no way of knowing that you have slapped a `method_missing` on your class that will allow instances to cheerfully handle `doc.replace_gender`. Depending on how elaborate your `method_missing` implementation is, you may be able to fix `respond_to?`:

```
  def respond_to?(name)
    string_name = name.to_s
    return true if string_name =~ /^replace_\w+/
    super
  end
```

As always, you need to balance the extra work involved with the likelihood that anyone is ever going to call `respond_to?` on this particular object.

In the Wild

You can find a very basic example of the magic method technique in the Ruby standard class `OpenStruct`. An `OpenStruct` instance is a cross between a simple data container object and a hash. Like a hash, you can use an `OpenStruct` to store whatever data you want, by whatever name you choose; the trick is that with `OpenStruct` you access your data with the `object.value` dot notation:

```
    require 'ostruct'

    author = OpenStruct.new
    author.first_name = 'Stephen'
    author.last_name = 'Hawking'

    puts author.first_name
    puts author.last_name
```

If you peek inside of the `OpenStruct` class you will find that it is built around a regular hash called `@table` and powered by `method_missing`. Here's the `OpenStruct` `method_missing`:

```
      def method_missing(mid, *args) # :nodoc:
        mname = mid.id2name
        len = args.length

        if mname =~ /=$/
          # Some error handling deleted...
          mname.chop!
          self.new_ostruct_member(mname)
          @table[mname.intern] = args[0]
        elsif len == 0
          @table[mid]

        else
          raise NoMethodError,
            "undefined method `#{mname}' for #{self}", caller(1)
        end
      end
```

The magic part of the `OpenStruct` `method_missing` is figuring out if the caller is trying to get an existing value from the hash or set a new value. To figure out which, the code looks at the name of the missing method: If the method name ends with an =, then the caller is trying to set a new value on the `OpenStruct` instance. If the method name doesn't end with an =, then this must be an attempt to access a value that has already been set. As magic goes, this is more of a card trick than making the Grand Canyon disappear, but it does illustrate the technique very nicely. ActiveRecord uses a much more serious bit of magic method conjuring: The ActiveRecord objects that represent database tables let you make up arbitrary finder methods. Imagine, for example, that you have an authors table in your database, a table that contains `fname` and `lname` fields. You also have an ActiveRecord class to go with your `authors` table:

```
class Author < ActiveRecord::Base
end
```

Given all this, ActiveRecord allows you to make up methods to find your records:

```
authors_named_henry = Author.find_by_fname(  'Henry'  )
james_family_authors = Author.find_by_lname( 'James' )
```

You can even combine the fields:

```
henry_james = Author.find_by_fname_and_lname( 'Henry', 'James' )
```

All of this courtesy of `method_missing`.

Wrapping Up

In this chapter we looked at using `method_missing` to create an infinite number of virtual methods, methods that don't actually exist as distinct blocks of source code but are there when you call them. To create these "there but not actually there" methods, we let `method_missing` catch the method call, parse the method name, and figure out what to do from there.

The `method_missing`-based technique of this and the last couple of chapters illustrates one of the basic ideas behind Ruby, that one of the main jobs of a programming language is to help the programmer get code executed when and where the programmer decides that code needs to be executed. This is what code blocks and modules do; they both allow you to package up code in one place and use it somewhere else. In this light, `method_missing` is just more of the same. It enables you to wedge some code in just where you need it.