CHAPTER 21

Use method_missing for Flexible Error Handling

My car worries too much. Well it's not the whole car, it's just the little embedded computer that lurks behind the door locks. That little processor seems to live in dread that I might someday accidentally leave my car unlocked. Thus, it devotes its whole being to making sure that no door stays unlocked for too long. I'm usually OK if I unlock the car and jump right in. But woe to me if I unlock the car and get a phone call. The time it takes to look at my cell phone to see who's calling is apparently too long by the exacting standards of Toyota, and the doors relock. They'll also lock in the time it takes to throw some groceries into the trunk. Or to kiss my wife goodbye. There have even been one or two occasions when the doors have locked on me a second time, in the interval it took for the initial rage to dissipate.

This kind of problem is not confined to cars. Whenever someone builds a system to handle an error condition there is always the chance that they will get it just a little wrong. You certainly see it in all kinds of software systems. People make mistakes often enough that it's useful to build software that helps deal with those mistakes. The problem is that sometimes the mitigating behavior is not quite right and you can't change it. Ideally, a good error-handing system will have some default behavior, but will also let you vary that behavior if you need to.

It's in that spirit that we're going to spend this chapter looking at `method_missing`, a feature of Ruby that allows you to handle a particular error condition: Someone has called a method that does not in fact exist. We will see how you can use `method_missing` to customize the way your code deals with this class of errors. We will also spend some time looking at `method_missing`'s close cousin, `const_missing`, and see how we can use it to increase the flexibility of our programs in some surprising ways.

Meeting Those Missing Methods

Imagine that the higher-ups in your company have decided to make the `Document` class the standard for manipulating text documents. Suddenly your little coding work of art goes from being used by a few departments to being the bytes behind the whole enterprise. For the most part the transition goes well, but you do get a series of bug reports from less-experienced engineers stating that the `Document` class "doesn't work" or that it is "utterly broken." Some of the reports say things like "I tried to get the text out of a `Document` instance and just got an exception." You think that the code does work,[1] so it seems likely that the cause of the trouble is pilot error, probably by doing something like this:

```
# Error: the method is content, not text!

doc = Document.new('Titanic', 'Cameron', 'Sail, crash, sink')
puts "The text is #{doc.text}"
```

You *think* this is what's going on, but how can you be sure?

One way to find out lies in the details of how Ruby calls a method, and, in particular, what it does when the method it is trying to call is not actually there. Take the previous example where the code tries to call a nonexistent `text` method on a `Document` instance. Initially, Ruby will look for the `text` method in the `Document` class and, failing to find it there, it will look in the superclass for `text`, and on up the line. If Ruby finds the method anywhere in the inheritance tree, then that's the method that gets called.

The real question is, what happens if, as with the phantom `text`, there just is no such method? The quick answer is that you get an exception. The not-so-quick answer is more interesting: When Ruby fails to find a method, it turns around and calls a second method. This second call, to a method with the somewhat odd name of `method_missing`, is what eventually generates the exception: It's the default imple-

1. Actually, given that you have tests, you *know* it works.

mentation of `method_missing`, found in the `Object` class[2] that raises the `NameError` exception.

The upshot of all of this is that you don't have to accept this default error-handing behavior. You are free to override `method_missing` in any of your classes and handle the case of the missing method yourself:

```
class RepeatBackToMe
  def method_missing( method_name, *args )
    puts "Hey, you just called the #{method_name} method"
    puts "With these arguments: #{args.join(' ')}"
    puts "But there ain't no such method"
  end
end
```

As you can see from this code, `method_missing` gets passed the name of the original method that was called along with the augments it was called with. If you make an instance of `RepeatBackToMe` and call some nonexistent methods on the instance, like this:

```
repeat = RepeatBackToMe.new
repeat.hello( 1, 2, 3 )
repeat.good_bye( "for", "now" )
```

Then `method_missing` will eventually fire and give you its cheerful summary of what just happened:

```
Hey, you just called the hello method
With these arguments: 1 2 3
But there ain't no such method
Hey, you just called the good_bye method
With these arguments: for now
But there ain't no such method
```

2. More specifically, the default `method_missing` actually lives in the `Kernel` module, which is included by `Object`.

Handling Document Errors

The `method_missing` method is tailor-made to help with the kind of problems you are having with your new `Document` users. At the very least, you can give them a customized message if they call a bad method:

```
class Document
  # Most of the class omitted...

  def method_missing( method_name, *args )
    msg =  %Q{
      You tried to call the method #{method_name}
       on an instance of Document. There is no such method.
    }
    raise msg
  end
end
```

Alternatively, you could return the same old error message, but quietly log the details of the mishap for later analysis:

```
class Document
  # Most of the class omitted...

  def method_missing( method_name, *args )
    File.open( 'document.error', 'a' ) do |f|
      f.puts( "Bad method called: #{method_name}" )
      f.puts( "with #{args.size} arguments" )
    end
    super
  end
end
```

You might even apply a little of that Ruby style programming customer service and try to help your user figure out which method he or she really meant:

```
require 'text'  # From the text gem

class Document
  include Text
```

```
    # Most of the class omitted...

  def method_missing( missing, *args )
    candidates = methods_that_sound_like( missing.to_s )

    message = "You called an undefined method: #{missing}."

    unless candidates.empty?
      message += "\nDid you mean #{candidates.join(' or ')}?"
    end
    raise raise NoMethodError.new( message )
  end

  def methods_that_sound_like( name )
    missing_soundex = Soundex.soundex( name.to_s )
    public_methods.sort.find_all do |existing|
      existing_soundex = Soundex.soundex( existing.to_s )
      missing_soundex == existing_soundex
    end
  end
end
```

This last `method_missing` implementation tries to figure out which method the user actually intended, based on the theory that the name of the method the user called is similar to the name of the method that the user meant to call. The code tries to guess the correct method by using the `Soundex` module (from the `text` gem) to compute a soundex code. The idea behind soundex is that similar-sounding words tend to generate the same soundex code. Thus, if you mistakenly call `document.contnt` you will get:

```
You called an undefined method: contnt.
Did you mean content or content=?
```

Coping with Constants

Methods are not, of course, the only things that can go missing in a Ruby program. Sometimes coders also misplace constants. In the same way that Ruby provides `method_missing` to cope with calls to nonexistent methods, it also gives you `const_missing` to deal with AWOL constants.

As you might guess, const_missing works a lot like method_missing: It gets called whenever Ruby detects a reference to an undefined constant. There are a couple of differences between the two _missing methods, one obvious and one more subtle. The obvious difference is that const_missing takes only a single argument, a symbol containing the name of the missing constant. References to constants, unlike method calls, do not have arguments.

The less obvious difference is that const_missing needs to be a class method:

```
class Document
    # Most of the class omitted...

  def self.const_missing( const_name )
    msg =  %Q{
      You tried to reference the constant #{const_name}
      There is no such constant in the Document class.
    }
    raise msg
  end
end
```

Like our method_missing examples, this example presses const_missing into service to help somewhat shaky Document users.

In the Wild

The best example of applying method_missing to improve error handling is probably the Rails **whiny nil** facility. The idea behind whiny nils is to help journeyman Rails developers cope with the inevitable situation where they think they have an instance of some object, perhaps an array or an ActiveRecord model, but what they actually have is nil. For example, your Rails application might go looking for a certain Author record in the database:

```
book_author = Author.find( :first,
    :conditions => { :name => 'Bilbo Baggins' })
```

Unfortunately, Bilbo is a fictional *character,* not a real author, so book_author ends up being nil. Sadly, your code might not notice the nil and try to save the nonexistent author:

```
book_author.save
```

This, inevitably, is going to lead to something painful. The Rails whiny nil feature takes some of the sting out by catching the bad call with a `method_missing` implementation.[3] Like our `Document` `method_missing`, the Rails version of `method_missing` raises a `NoMethodError` with a customized message:

```
#<NoMethodError: You have a nil object when you didn't expect it!
```

Rails puts a cool spin on all of this by comparing the missing method name with the names of the methods supported by arrays and ActiveRecord model classes. If it finds a match, Rails will add a helpful suggestion to the exception it throws:

```
You might have expected an instance of ActiveRecord::Base.
The error occurred while evaluating nil.save.
```

Similarly, Rake uses `const_missing` to provide helpful warnings about deprecated names. It seems that earlier versions of Rake defined the core Rake classes at the top level, without any encapsulating modules. In those bygone days the class of a Rake task was simply `Task`, with no module. Somewhere along the way, however, the Rake classes were all enclosed in the `Rake` module, so `Task` morphed into `Rake::Task`. But what to do about all of those Rakefiles out there that still referred to plain old `Task`? You pull out `const_missing`, that's what:

```
  def const_missing(const_name)
    case const_name
    when :Task
      Rake.application.const_warning(const_name)
      Rake::Task
    when :FileTask
      Rake.application.const_warning(const_name)
      Rake::FileTask
    when :FileCreationTask
      Rake.application.const_warning(const_name)
      Rake::FileCreationTask
```

3. We will deal with the question of how you add a new method to the existing `NilClass` in Chapter 24.

```
    when :RakeApp
      Rake.application.const_warning(const_name)
      Rake::Application
    else
      rake_original_const_missing(const_name)
    end
  end
```

This clever bit of code catches the references to those old, naked class names and returns the correct class, printing out a helpful warning along the way.

Perhaps the most spectacular use of `const_missing` lies in the way that Rails uses it to load Ruby code as needed. Your ActiveRecord model classes are, for example, all loaded on an as-needed basis, driven by `const_missing`. The basic idea is to include a `const_missing` method that figures out what file to require from the name of the missing class, loads that file, and then returns the newly loaded class, something like this:

```
def self.const_missing( name )
  file_name = "#{name.to_s.downcase}"
  require file_name
  raise "Undefined: #{name}" unless const_defined?(name)
  const_get(name)
end
```

Once you have this `const_missing` method defined in your class (or some superclass of your class), any reference to the missing constant `Wizard` will trigger a `require 'wizard'`.[4]

Staying Out of Trouble

There are a few things to remember about `method_missing`- and `const_missing`-based error handling. First, you don't want to use it unless you really need it. The garden-variety Ruby error handling will suffice for about 99.9% of all of your misspelled

4. As an alternative, built into Ruby is the `autoload` method, which allows you to specify which file any given class is to be found. While the `autoload` method can be useful, it is nowhere near as flexible as the `const_missing`-based technique.

or misplaced methods, and there really is no reason to pepper your classes with `method_missing` implementations unless, as in our little tale of "the `Document` class is broken" woe, you really need to. Save the fancy `method_missing` for those cases where you really do need to do something fancy with the error.

Second, keep in mind that the penalty for screwing up in `method_missing` and `const_missing` can be pretty high. Think about it: Ruby executes `method_missing` any time there is a method call that it can't locate. Be very, very careful that you don't inadvertently call a nonexistent method inside your `method_missing` method. If you do, Ruby will repeat the whole `method_missing` kerfuffle, eventually arriving back at the `method_missing` method—which is likely to reach out again for the same nonexistent method, and off you are onto an all-expenses-paid trip to infinite recursion land. If you value your programs, treat your `method_missing` and `const_missing` code like anything else that you write. It ain't done until you've written the tests. Only more so.

Wrapping Up

In this chapter we looked at how Ruby deals with references to things that don't actually exist. We've seen that Ruby will call `method_missing` if you try to call a method that doesn't actually exist and `const_missing` if you reference a constant that exists only in your imagination. We saw how you can use `method_missing` and `const_missing` to improve the error handling in your code and even how you can use `const_missing` as a mechanism for auto-loading code.

Although error handling is the most obvious use for `method_missing`, it is certainly not the only use. In fact, error handling accounts for only a very small fraction of `method_missing` use in the Ruby code base. The other, more common uses of `method_missing` are where we now turn our attention.