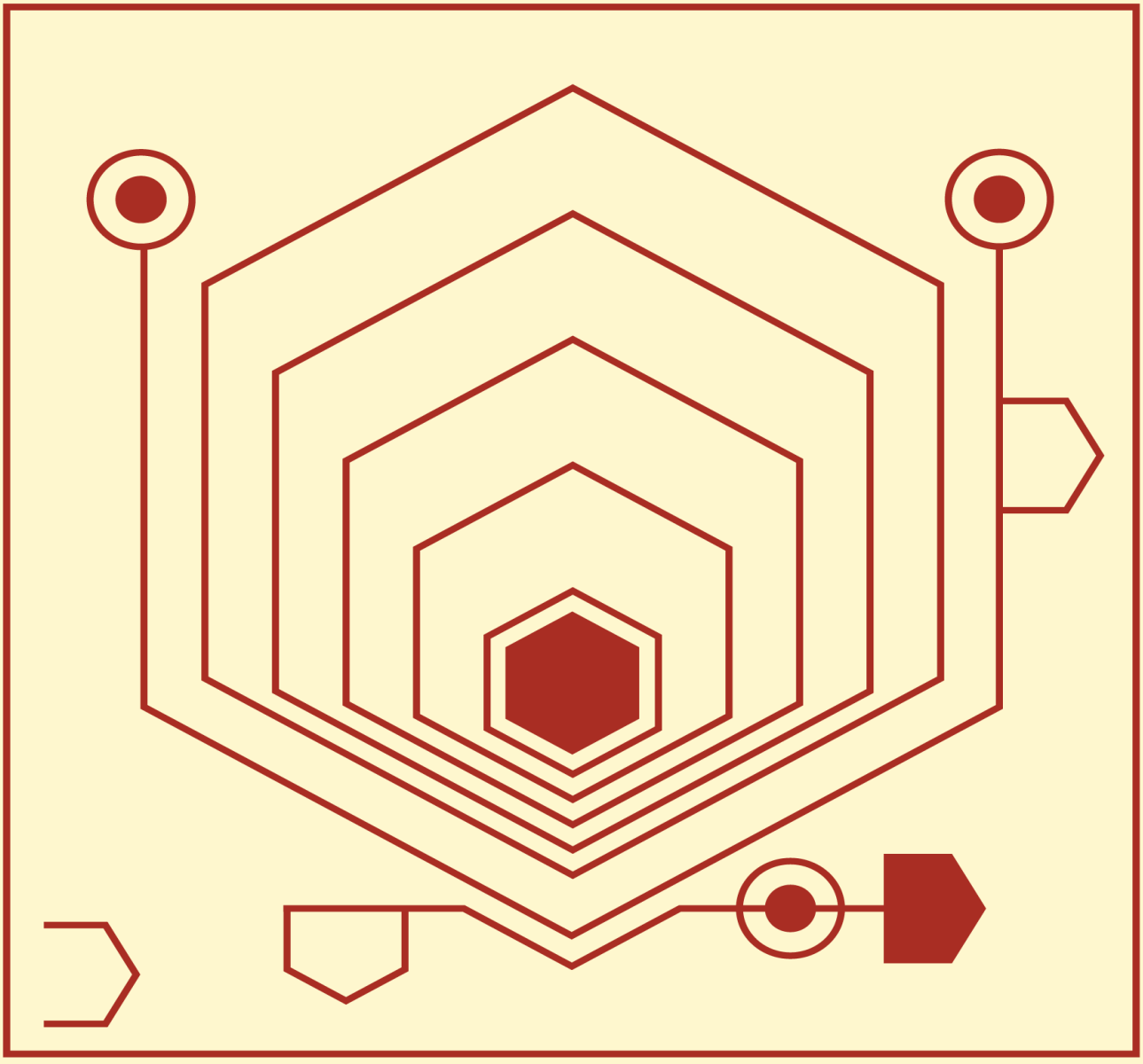

Confident Ruby

by Avdi Grimm

4.3 Conditionally call conversion methods

Indications

You want to provide support for transforming inputs using conversion protocols, without forcing all inputs to understand those protocols. For instance, you are writing a method which deals with filenames, and you want to provide for the possibility of non-filename inputs which can be implicitly converted to filenames.

Synopsis

Make the call to a conversion method conditional on whether the object can respond to that method.

Rationale

By optionally supporting conversion protocols, we can broaden the range of inputs our methods can accept.

Example: opening files

Earlier [page 40], we said that `File.open` calls `#to_path` on its `filename` argument. But `String` does not respond to `#to_path`, yet it is still a valid argument to `File.open`.

```
"/home/avdi/.gitconfig".respond_to?(:to_path) # => false
File.open("/home/avdi/.gitconfig")
# => #<File:/home/avdi/.gitconfig>
```

Why does this work? Let's take a look at the code. Here's the relevant section from MRI's `file.c`:

```
CONST_ID(to_path, "to_path");
tmp = rb_check_funcall(obj, to_path, 0, 0);
if (tmp == Qundef) {
  tmp = obj;
 }
StringValue(tmp);
```

This code says, in effect: "see if the passed object responds to `#to_path`. If it does, use the result of calling that method. Otherwise, use the original object. Either way, ensure the resulting value is a `String` by calling `#to_str`"

Here's what it would look like in Ruby:

```
if filename.respond_to?(:to_path)
  filename = filename.to_path
end

unless filename.is_a?(String)
  filename = filename.to_str
end
```

Conditionally calling conversion methods is a great way to provide flexibility to client code: code can *either* supply the expected type, *or* pass something which responds to a documented conversion method. In the first case, the target object isn't forced to understand the conversion method. So you can safely use conversion methods like `#to_path` which are context-specific. Of course, for client code to take advantage of this flexibility, the optional conversion should be noted in the class and/or method documentation.

This is especially useful when we are defining our own conversion methods, which we'll talk more about in the next section [page 59].

Violating duck typing, just this once

But wait... didn't we say that calling `#respond_to?` violates duck-typing principles? How is this exception justified?

To examine this objection, let's define our own `File.open` wrapper, which does some extra cleanup before opening the file.

```
def my_open(filename)
  filename.strip!
  filename.gsub!(/^~/,ENV['HOME'])
  File.open(filename)
end

my_open("  ~/.gitconfig   ")     # => #<File:/home/avdi/.gitconfig>
```

Let's assume for the sake of example that we want to make very sure we have the right sort of input *before* the logic of the method is executed. Let's look at all of our options for making sure this method gets the inputs it needs.

We could explicitly check the type:

```
def my_open(filename)
  raise TypeError unless filename.is_a?(String)
  # ...
end
```

I probably don't have to explain why this is silly. It puts an arbitrary and needlessly strict constraint on the class of the input.

We could check that the object responds to every message we will send to it:

```
def my_open(filename)
  unless %w[strip! gsub!].all?{|m| filename.respond_to?(m)}
    raise TypeError, "Protocol not supported"
  end
  # ...
end
```

No arbitrary class constraint this time, but in some ways this version is even worse. The protocol check at the beginning is terribly brittle, needing to be kept in sync with the list of methods that will actually be called. And `Pathname` arguments are still not supported.

We could call `#to_str` on the input.

```
def my_open(filename)
  filename = filename.to_str
  # ...
end
```

But this still doesn't work for `Pathname` objects, which define `#to_path` but not `#to_str`

We could call `#to_s` on the input:

```
def my_open(filename)
  filename = filename.to_s
  # ...
end
```

This would permit both `String` and `Pathname` objects to be passed. But it would also allow invalid inputs, such as `nil`, which are passed in error.

We could call `to_path` on every input, and alter `String` so it responds to `#to_path`:

```
class String
  def to_path
    self
  end
end

def my_open(filename)
  filename = filename.to_path
  # ...
end
```

This kind of thing could quickly get out of hand, cluttering up core classes with dozens of monkey-patched conversion methods for specific contexts. Yuck.

Finally, we could conditionally call `#to_path`, followed by a `#to_str` conversion, just as `File.open` does:

```
def my_open(filename)
  filename = filename.to_path if filename.respond_to?(:to_path)
  filename = filename.to_str
  # ...
end
```

Of all the strategies for checking and converting inputs we've looked at, this one is the most flexible. We can pass `String`, `Pathname`, or any other object which defines a conversion to a path-type `String` with the `#to_path` method. But other objects passed in by mistake, such as `nil` or a `Hash`, will be rejected early.

This use of `#respond_to?` is different from most type-checking in a subtle but important way. It doesn't ask "are you the kind of object I need?". Instead, it says "can you give me the kind of object I need?" As such, it strikes a useful balance. Inputs are checked, but in a way that is *open for extension*.

Conclusion

Sometimes we want to have our cake and eat it too: a method that can take input either as a core type (such as a `String`), or as a user-defined class which is convertible to that type. Conditionally calling a conversion method (such as `#to_path`) only if it exists is a way to provide this level of flexibility to our callers, while still retaining confidence that the input object we finally end up with is of the expected type.