CHAPTER 7
Reducing Cultural Barriers

Ten years ago, a book on best practices for any given programming language would seem perfectly complete without a chapter on multilingualization (m17n) and localization (L10n). In 2009, the story is just a little bit different.

Now that we've created a network of applications and infrastructure designed not simply to be used by hackers and researchers, but by nontechnical folks as part of their day-to-day lives, we are living in a very different world. With most software serving first and foremost as a communication medium, it is unrealistic to think that all conversations should be conducted in English or require the use of specialized tools otherwise. This presents a challenge to those who implement software that needs to be accessible to a global user base.

Although some may argue that it took too long to materialize, Ruby 1.9 provides a robust and elegant solution to the m17n problem. Rather than binding its users to a particular internal encoding and requiring complex manual manipulation of text into that format, Ruby 1.9 provides facilities that make it easy to transcode text from one encoding to another. This system is well integrated so that things like pattern matching and I/O operations can be carried out in all of the encodings Ruby supports, which provides a great deal of flexibility for those who need to do encoding-specific operations. Of course, because painless transcoding is possible, you can also write code that accepts and produces text in a wide variety of encodings, but uses a single encoding throughout its internals, improving the consistency and simplicity of the underlying implementation.

In this chapter, we're going to look at both of these approaches in some detail. We'll also look at some of the shortcuts Ruby offers when it comes to day-to-day scripting needs. In these cases, you may not need full-fledged support for arbitrarily encoded text, so there are ways to skip some steps if you know exactly what you'll be dealing with. By the time we wrap up with our discussion of m17n, you'll have a few solid strategies that can be adapted to fit your needs.

Once you are comfortable with how to store, manipulate, and produce internationalized text in various character encodings, you may want to know about how to customize

your software so that its interface is adapted to whatever the native language and dialogue of its users might be. Although multilingualization and localization requirements don't necessarily come in pairs, they often do, so I'll wrap up this chapter with a brief demonstration of how to localize a simple Ruby application without excess duplication of effort.

Though we'll be talking about both topics, it's fair to say that technologically speaking, L10n is pretty easy, and m17n is fairly involved. This chapter won't necessarily teach you about either, but you should certainly review the basics of character encodings and have a working understanding of Ruby's m17n system before moving on. If you don't, go ahead and look over your favorite Ruby reference book before moving on, and maybe check out Joel Spolsky's classic article (*http://www.joelonsoftware.com/articles/Unicode.html*) on Unicode and character sets, which is the best I've seen as an introduction.

Once you're up to speed, we can move on to a practical example of m17n as seen in one of Ruby's standard libraries.

As always, we'll start with the deep dive and then go over some of the specifics later in the chapter. However, in this chapter in particular, I'm hoping that you don't skip the initial real-world example. Even if you don't fully grasp what's going on, the concepts it introduces are essential for understanding the rest of the content in this chapter. It also happens to be some of the coolest code in the book, so let's jump right in so you can see how it works.

m17n by Example: A Look at Ruby's CSV Standard Library

In a few other places in the book, I mentioned that Ruby's standard library is a good place to look for quality code that best practices can be gleaned from. When it comes to m17n, the place to look is the CSV library. This library is used for reading and writing files in the popular comma-separated-value tabular data format. We won't be talking much about the details of how this library is actually used here, but will instead be investigating a particular aspect of how it is implemented.

What we're interested in is how CSV manages to parse data that is in any of the character encodings that Ruby supports without transcoding the source text. To clarify the challenge, we can notice that generally speaking, character encodings are not necessarily compatible with one another:*

```
# coding: UTF-8

utf8_string = "foo — bar"
sjis_string = utf8_string.encode("Shift_JIS")

p utf8_string == sjis_string #=> false
```

* Throughout this chapter, note that — is the Japanese character for 1, not a double dash.

This issue is propagated well beyond `String` equality, and can cause more noisy problems in things like pattern matching or string manipulation:

```
sjis_string =~ /(\w+)\s —\s(\w+)/
encoding_test.rb:6:in '<main>': incompatible encoding regexp match
(UTF-8 regexp with Shift_JIS string) (Encoding::CompatibilityError)

utf8_string << sjis_string # RAISES
encoding_test.rb:6:in '<main>': incompatible character encodings:
UTF-8 and Shift_JIS (Encoding::CompatibilityError)
```

So the fact remains that some transcoding needs to be done. But imagine that in this example `sjis_string` represents some preloaded data that has not been converted yet. If our source encoding is set to use UTF-8, we could of course transcode the `sjis_string` to UTF-8 before operating on it. However, this could potentially be costly if our source text was large. To work around this, CSV converts its parsing patterns instead. We'll look at that in much greater detail in just a moment, but just to get a feel for it, let's look at an oversimplified example of how that might work:

```
def sjis_re(str)
  Regexp.new(str.encode("Shift_JIS"))
end

sjis_string =~ sjis_re("(\w+)\s —\s(\w+)")
p [$1, $2] #=> ["foo", "bar"]
```

Here, we use a UTF-8 string literal to build up a regular expression, but before it is constructed, the string is transcoded to Shift-JIS. As a result, the pattern it creates is capable of matching against Shift-JIS strings.

Although a bit more work than using `Regexp` literals, this presents a fairly natural way of defining regular expressions to match text that is in a different encoding from the source encoding. If we build on this concept a little bit, you can get a sense of how CSV's parser works.

Let's look at the CSV implementation from the bottom up. The end goal is to generate a parser in the encoding of the original data source that can extract cells from a tabular data format. To do this, CSV needs to be able to transcode some special characters into the encoding of the source text. These include the column separator, row separator, and literal quote character. As these can all be customized when a CSV is loaded, some general helpers for encoding strings and regular expressions come in handy:

```
# Builds a regular expression in <tt>@encoding</tt>.  All +chunks+ will be
# transcoded to that encoding.
#
def encode_re(*chunks)
  Regexp.new(encode_str(*chunks))
end

# Builds a String in <tt>@encoding</tt>.  All +chunks+ will be transcoded to
# that encoding.
#
```

```
def encode_str(*chunks)
  chunks.map { |chunk| chunk.encode(@encoding.name) }.join
end
```

In this code, `@encoding` refers to the `Encoding` object pulled from the underlying CSV string. For example, you might be trying to load a CSV file encoded in Shift-JIS:

```
# coding: UTF-8

require "csv"

CSV.read("data.csv", encoding: "Shift_JIS")
```

In essence, these two helper methods provide the same functionality as our `sjis_re` function that we built earlier. The main difference is that here, we have gleaned the necessary information from an `Encoding` object stored in the CSV instance rather than hardcoded a particular encoding to use. This makes perfect sense in the context of CSV, and hopefully is relatively easy to understand.

In addition to encoding regular expressions, because CSV accepts user-entered values that modify its core parser, it needs to escape them. Although the built-in `Regexp.escape()` method works with most of the encodings Ruby supports, at the time of the Ruby 1.9.1 release, it had some issues with a handful of them. To work around this, CSV rolls its own escape method:

```
# This method is an encoding safe version of Regexp.escape().  It will escape
# any characters that would change the meaning of a regular expression in the
# encoding of +str+.  Regular expression characters that cannot be transcoded
# to the target encoding will be skipped and no escaping will be performed if
# a backslash cannot be transcoded.
#
def escape_re(str)
  str.chars.map { |c| @re_chars.include?(c) ? @re_esc + c : c }.join
end
```

This means that once things like the column separator, row separator, and quote character have been specified by the user and converted into the specified encoding, this code can check to see whether the transcoded characters need to be escaped. To make things clearer, we can see that `@re_chars` is set in the `CSV` constructor as simply a list of regular expression reserved characters transcoded to the specified `@encoding`:

```
@re_chars =   %w[ \\ .  [  ]  -  ^  $  ?
                  *  +  {  }  (  )  |  #
                  \  \r \n \t \f \v ].
              map { |s| s.encode(@encoding) rescue nil }.compact
```

When we put these helpers together, we can see the big picture materialize within CSV's core parsers. Although the next code sample may look long at first, if you read through it, you'll see it's basically just a handful of regular expressions each doing a particular part of the CSV parsing job:

```
# Pre-compiles parsers and stores them by name for access during reads.
def init_parsers(options)
```

```
    # store the parser behaviors
    @skip_blanks      = options.delete(:skip_blanks)
    @field_size_limit = options.delete(:field_size_limit)

    # prebuild Regexps for faster parsing
    esc_col_sep = escape_re(@col_sep)
    esc_row_sep = escape_re(@row_sep)
    esc_quote   = escape_re(@quote_char)
    @parsers = {
      # for empty leading fields
      leading_fields: encode_re("\\A(?:", esc_col_sep, ")+"),
      # The Primary Parser
      csv_row:        encode_re(
        "\\G(?:\\A|", esc_col_sep, ")",               # anchor the match
        "(?:", esc_quote,                             # find quoted fields
               "((?>[^", esc_quote, "]*)",            # "unrolling the loop"
               "(?>", esc_quote * 2,                  # double for escaping
               "[^", esc_quote, "]*)*)",
               esc_quote,
               "|",                                   # ... or ...
               "([^", esc_quote, esc_col_sep, "]*))",  # unquoted fields
        "(?=", esc_col_sep, "|\\z)"                   # ensure field is ended
      ),
      # a test for unescaped quotes
      bad_field:      encode_re(
        "\\A", esc_col_sep, "?",                  # an optional comma
        "(?:", esc_quote,                         # a quoted field
               "(?>[^", esc_quote, "]*)",         # "unrolling the loop"
               "(?>", esc_quote * 2,              # double for escaping
               "[^", esc_quote, "]*)*",
               esc_quote,                         # the closing quote
               "[^", esc_quote, "]",              # an extra character
               "|",                               # ... or ...
               "[^", esc_quote, esc_col_sep, "]+", # an unquoted field
               esc_quote, ")"                     # an extra quote
      ),
      # safer than chomp!()
      line_end:       encode_re(esc_row_sep, "\\z"),
      # illegal unquoted characters
      return_newline: encode_str("\r\n")
    }
  end
```

By default, the `@col_sep` will be a comma, and the `@quote_char` will be a double-quote. The default `@row_sep` can vary depending on what sort of line endings are used in the file. These options can also be overridden when data is loaded. For example, if you for some reason had data columns separated by the Japanese character for 2, you could split things up that way:

```
# coding: UTF-8

require "csv"

CSV.read("data.csv", encoding: "Shift_JIS", col_sep: "二")
```

All of these options get automatically transcoded to the specified `encoding`, which means you don't need to do the UTF-8 → Shift-JIS conversion manually here. This, combined with some of the other conveniences we've gone over, makes it clear that despite the fact that CSV manages to support every encoding Ruby does, it minimizes the amount an end user needs to know about the underlying grunt work. As long as the proper encoding is specified, CSV handles all the rest, and can even do things like auto-transcoding on load if you ask it to.

When it boils down to it, there isn't a whole lot of complexity here. By working with strings and regular expressions indirectly through encoding helpers, we can be sure that any pattern matching or text manipulation gets done in a compatible way. By translating the parser rather than the source data, we incur a fixed cost rather than one that varies in relation to the size of the data source. For a need like CSV processing, this is very important, as the format is often used for large data dumps.

The downside of this approach is of course that you need to be extra careful about how you work with your source text. It is not a big deal here because CSV processing is a well-specified task that has limited feature requirements. Constantly having to remember to encode every string and regular expression you use in your project could quickly become overwhelming if you are working on a more multifaceted problem.

In the next section, we'll cover an alternative m17n solution that isn't quite as pure as the approach CSV takes, but is generally less work to implement and still works fairly well for most needs.

Portable m17n Through UTF-8 Transcoding

Although it's nice to be able to support each character encoding natively, it can be quite difficult to maintain a complex system that works that way. The easy way out is to standardize on a single, fairly universal character encoding to write your code against. Then, all that remains to be done is to transcode any string that comes in, and possibly transcode again on the way out. The character set of choice for use in code that needs to be portable from one system to another is UTF-8.

Many Ruby libraries consume UTF-8 and UTF-8 only. The choice is a reasonable one, as UTF-8 is a proper superset of ASCII, meaning that code that pays no attention to specialized character encodings is likely to work without modification. UTF-8 also is capable of representing the myriad character sets that make up Unicode, which means it can represent nearly any glyph you might imagine in any other character encoding. As a variable-length character encoding, it does this fairly efficiently, so that users who do not need extra bytes to represent large character sets do not incur a significant memory penalty.

We're now going to walk through the general process of writing a UTF-8-enabled Ruby library. Along the way, we'll occasionally look at some examples from Prawn, to give a sense of what these techniques look like when they're applied in an actual project.

Source Encodings

A key aspect of any m17n-capable Ruby projects is to properly set the source encodings of its files. This is done via the magic comments that we have already seen in some of the earlier examples in this chapter. When it comes to Prawn, you'll see that each and every source file that makes up the library starts with a line like this:

```
# coding: UTF-8
```

In order for Ruby to pick it up, this comment must be the first line in the file, unless a shebang is present, such as in the following example:

```
#!/usr/bin/env ruby
# coding: UTF-8
```

In this case, the magic comment can appear on the second line. However, in all other situations, nothing else should come before it. Although Ruby is very strict about where you place the comment, it's fairly loose about the way you write it. Case does not matter as long as it's in the form of `coding: some_encoding`, and extra text may appear before or after it. This is used primarily for editor support, allowing things such as Emacs-style strings:

```
# -*- coding: utf-8 -*-
```

However you choose to format your magic comments, actually including them is important. Their purpose is to tell Ruby what encoding your regex and string literals are in. Forgetting to explicitly set the source encoding in this manner can cause all sorts of nasty problems, as it will force Ruby to fall back to US-ASCII, breaking virtually all internationalized text.

Once you set the source encoding to UTF-8 in all your files, if your editor is producing UTF-8 output, you can be sure of the encoding of your literals. That's the first step.

Working with Files

By default, Ruby uses your locale settings to determine the default external character encoding for files. You can check what yours is set to by running this code:

```
$ ruby -e "p Encoding.default_external"
```

On my system, this prints out `#<Encoding:UTF-8>`, but yours might be different. If your locale information isn't set, Ruby assumes that there is no suitable default encoding, reverting to `ASCII-8BIT` to interpret external files as sequences of untranslated bytes.

The actual value your `default_external` is set to doesn't really matter when you're developing code that needs to run on systems that you do not control. Because most libraries fall under this category, it means that you simply cannot rely on `File.open()` or `File.read()` to work without explicitly specifying an encoding.

This means that if you want to open a file that is in Latin-1 (ISO-8859-1), but process it within your UTF-8-based library, you need to write code something like this:

```
data = File.read("foo.txt", encoding:"ISO-8859-1:UTF-8")
```

Here, we've indicated that the file we are reading is encoded in ISO-8859-1, but that we want to transcode it to UTF-8 immediately so that the string we end up with in our program is already converted for us. Unless you need to retain the original encoding of the text for some reason, this is generally a good idea when processing files within a UTF-8 based library.

Writing back to file works in a similar fashion. Here's what it looks like to automatically transcode text back to Latin-1 from a UTF-8 source string:

```
File.open("foo.txt", "w:ISO-8859-1:UTF-8") { |f| f << data + "Some extra text" }
```

Although the syntax is slightly different here, the idea is the same. We specify first the external character encoding that is used for the file and second the internal encoding we are working with. So with this in mind, in a UTF-8-based library, you will need to supply an encoding string of the form `external_format:UTF-8` whenever you're working with text files. Of course, if the external format happens to be UTF-8, you would just write something like this:

```
data = File.read("foo.txt", encoding: "UTF-8")
File.open("foo.txt", "w:UTF-8") { |f| f << data + "Some extra text"
```

The underlying point here is that if you want to work with files in a portable way, you need to be explicit about their character encodings. Without doing this, you cannot be sure that your code will work consistently from machine to machine. Also, if you want to make it so all of the internals of your system operate in a single encoding, you need to explicitly make sure the loaded files get translated to UTF-8 before you process the text in them. If you take care of these two things, you can mostly forget about the details, as all of the actual work will end up getting done on UTF-8 strings.

There is one notable exception to this rule, which is dealing with binary files. It used to be the case that at least on *nix, you could get away with code like this on Ruby 1.8:

```
img_data = File.read("foo.png")
```

Reading binary files this way was never a good idea, because it would cause data corruptions on Windows due to the fact that it does automatic conversions to the line endings of the files. However, there is now a reason not to write code like this, regardless of what platform you are on.

As we had mentioned before, Ruby looks to your `default_external` encoding when one is not explicitly specified. Because this is set by your locale, it can be any number of things. On my system, as my locale is set to UTF-8, Ruby thinks I'm trying to load a UTF-8 based file and interprets my binary as such. This promptly breaks things in all sorts of unpleasant ways, so it is something to be avoided. Luckily, if we simply use `File.binread()`, all these problems go away:

```
img_data = File.binread("foo.png")
img_data.encoding #=> #<Encoding:ASCII-8BIT>
```

For more complex needs, or for when you need to write a binary file, Ruby 1.9 has also changed the meaning of `"rb"` and `"wb"` in `File.open()`. Rather than simply disabling line-ending conversion, using these file modes will now set the external encoding to `ASCII-8BIT` by default. You can see this being used in Prawn's `Document#render_file` method, which simply takes the raw PDF string written by `Document#render` and outputs a binary file:

```
class Prawn::Document

  # ...

  def render_file(filename)
    File.open(filename,"wb") { |f| f << render }
  end

end
```

This approach may already seem familiar to those who have needed to deploy code that runs on Windows machines. However, this section is meant to remind folks who may not have previously needed to worry about this that they need to be careful as well.

There really isn't a whole lot to worry about when working with files using this m17n strategy, but the few things that you do need to do are important to remember. Unless you're working with binaries, be sure to explicitly specify the external encoding of your files, and transcode them to UTF-8 upon read or write. If you are working with binaries, be sure to use `File.binread()` or `File.open()` with the proper flags to make sure that your text is not accidentally encoded into the character set specified by your locale. This one can produce subtle bugs that you might not encounter until you run your code on another machine, so it's important to try to avoid in the first place.

Now that we've talked about source encodings and files, the main thing that's left to discuss is how to transcode input from users in a fairly organized way, and produce output in the formats you need. Though this is not complicated, it's worth looking at some real code to see how this is handled.

Transcoding User Input in an Organized Fashion

When dealing with user input, you need to make a decision regarding what should be transcoded and where. In the case of Prawn, we take a very minimalist approach to this. While we expect the end user to provide us with UTF-8-encoded strings, many of the features that involve simple comparisons work without the need for transcoding.

As it turns out, Ruby does a lot of special casing when it comes to strings containing only ASCII characters. Although strings that have different character encodings from one another are generally not comparable and cannot be combined through manipulations, these rules do not apply if each of them consists of only ASCII characters.

It turns out that in practice, you don't really need to worry about transcoding whenever you are comparing user input to a finite set of possible ASCII values.

Here's an example of a case where transcoding is not necessary:

```
# coding: iso-8859-1

require "prawn"

Prawn::Document.generate("output.pdf", :page_size => "LEGAL") do
  stroke_line [100,100], [200,200]
end
```

In this case, neither the filename nor the page size are transcoded within Prawn. Although we passed Latin-1 strings in, Prawn didn't bother to translate them to UTF-8, because it didn't need to. The filename is eventually passed straight through to `File.open()`, so no manipulations or comparisons are ever done on it. However, `:page_size` is used to look up page dimensions in a hash that looks something like this:

```
SIZES = {     "A4" => [595.28, 841.89 ],
           "FOLIO" => [612.00, 936.00 ],
           "LEGAL" => [612.00, 1008.00],
          "LETTER" => [612.00, 792.00 ]  }
```

Although the whole of Prawn's source is using UTF-8 literals, and the example we showed earlier is using Latin-1, the proper page size ends up getting looked up in the hash. As it turns out, the laziness of Ruby's m17n system really comes in handy.

When dealing with strings in which `String#ascii_only?` returns `true`, the object acts for all intents and purposes as if it were an ASCII string. It will play nice with strings of other encodings as long as they too consist entirely of ASCII characters.

This gives you a good criteria for what needs to be transcoded and what doesn't. If you can be sure that you never manipulate or compare a string, it can be safely ignored in most cases. In cases in which you do manipulation or comparison, if the input strings will consist of nothing more than ASCII characters in all cases, you do not need to transcode them. All other strings need to be transcoded to UTF-8 within your library unless you expect users to do the conversions themselves.

After combing your API, depending on your needs, the number of parameters that need to be converted may be a lot or a little. In the case of Prawn, even though it is a relatively complex system, it turned out to be a very small task to make the library work fairly well with the arbitrary strings that users pass in to our methods.

Essentially the only time we needed to normalize encodings of strings was when dealing with text that ends up getting displayed within the PDFs. Because our system essentially works by mapping Unicode code points to the actual outlines of characters (glyphs) that end up getting rendered to the document, we cannot simply deal with any text the user provides us. It needs to be transcoded.

If we look at the core `text()` method, we can see where this actually occurs in Prawn:

```
def text(text,options={})
  # we'll be messing with the strings encoding, don't change the users
```

```
    # original string
    text = text.to_s.dup   ## A ##

    save_font do
      options = text_options.merge(options)
      process_text_options(options)

      font.normalize_encoding(text) unless @skip_encoding  ## B ##

      # ... remainder of method is not important
    end
  end
```

On the line marked *A*, we see that the `text()` method makes a copy of the string, because we will change its encoding later. The few lines that follow it are not of interest, until we reach the line marked *B*. This line calls `Prawn::Font#normalize_encoding` with the text that is to be rendered, and that method is responsible for doing the transcoding. This method is actually implemented by subclasses of `Prawn::Font`, because different font systems require different encodings. In the interest of simplicity, we'll look at TTF fonts, where UTF-8 text is used:

```
module Prawn
  class Font
    class TTF < Font

      def normalize_encoding(text)
        text.encode!("UTF-8")
      end

    end
  end
end
```

Here we can see that the code for `Prawn::Font::TTF#normalize_encoding()` is just a thin wrapper on `String#encode!()`, which is used to modify the encoding of a string in place. The reason it exists at all is because different font systems have different requirements; for example, `Prawn::Font::AFM` uses Windows-1252 under the hood.

The underlying idea here is simple, though. You can clean up your code significantly by identifying the points where encodings matter in your code. Oftentimes, there will be a handful of low-level functions that are at the core of your system, and they are the places where transcoding needs to be done. In the case of Prawn, even with things like formatted text boxes and tables and other fancy means of putting content on the page, the core method that is used by all of those features is the single `text()` method. At this point, we call a `normalize_encoding` method that abstracts away the actual encoding that gets used. In the case of Prawn, this is especially important—although the core library works in UTF-8, depending on which font you use, text strings entered from the user may end up being transcoded into various different encodings.

Although some libraries may be more complex than others to implement in this manner, for a wide variety of purposes, this works fairly well. The main reason for using

UTF-8 is that it provides a good base encoding that most other encodings can be transcoded into. If you take advantage of Ruby's shortcuts when it comes to ASCII strings, you can safely ignore a lot of the finer points about whether two different encodings play nice together.

Though a library implemented in this way isn't truly set up to deal with arbitrary character encodings in the way that something like CSV implements its m17n support, it provides an easy solution that may work in most cases. Although the looseness of such a system does have some drawbacks, the advantages are often worth it.

Roughly, the process of building a UTF-8 based system goes like this:

- Be sure to set the source encoding of every file in your project to UTF-8 using magic comments.
- Use the `external:internal` encoding string when opening any I/O stream, specifying the internal encoding as UTF-8. This will automatically transcode files to UTF-8 upon read, and automatically transcode from UTF-8 to the external encoding on write.
- Make sure to either use `File.binread()` or include the `"b"` flag when dealing with binary files. Otherwise, your files may be incorrectly interpreted based on your locale, rather than being treated as a stream of unencoded bytes.
- When dealing with user-entered strings, only transcode those that need to be manipulated or compared to non-ASCII strings. All others can be left in their native encoding as long as they consist of ASCII characters only or they are not manipulated by your code.
- Do not rely on `default_external` or `default_internal`, and be sure to set your source encodings properly. This ensures that your code will not depend on environmental conditions to run.
- If you need to do a ton of text processing on user-entered strings that may use many different character mappings, it might not be a great idea to use this approach.

Although you may run into some other challenges depending on what your actual project is like, the tips above should get you most of the way to a working solution.

Each approach we've discussed so far illustrates a trade-off between high compatibility and simplicity of implementation. While UTF-8-based systems with limited automatic transcoding support represent a middle-of-the-road approach, we'll now look at how to use every possible shortcut to quickly get your job done at the expense of portability.

m17n in Standalone Scripts

Ruby is a scripting language at heart. Although the earlier m17n represents an elegant and highly flexible system, it would be tedious to think about all of its details when you just want to process some datafiles or run a quick one-liner on the command line.

Although we won't spend much time going over the minute details, I want to make sure that you get a chance to see how Ruby lets the m17n system get out of your way a bit when it comes to day-to-day scripting needs.

Inferring Encodings from Locale

The key things that Ruby's m17n system can modify when it comes to encodings are the source encoding, the default external encoding of files, and the default internal encoding that loaded files should be transcoded to. The following simple script inspects each of these values to see how they relate to your system's locale:

```
puts "Source encoding: #{__ENCODING__.inspect}"
puts "Default external: #{Encoding.default_external.inspect}"
puts "Default internal: #{Encoding.default_internal.inspect}"
puts "Locale charmap: #{ Encoding.locale_charmap.inspect}"
puts "LANG environment variable: #{ENV['LANG'].inspect}"
```

When we run this code without a magic comment, I get the following output on my system. Your exact values may vary, as I'll explain in a moment:

```
$ ruby encoding_checker.rb

Source encoding: #<Encoding:US-ASCII>
Default external: #<Encoding:UTF-8>
Default internal: nil
Locale charmap: "UTF-8"
LANG environment variable: "en_US.UTF-8"
```

Here, we see that the source encoding has fallen back to US-ASCII because none was explicitly set. However, the default external matches our locale charmap, which is inferred from the `LANG` environment variable. Although your locale may not be set to UTF-8, you should see that these values match each other on your system, as long as `LANG` is set to one of the encodings Ruby supports. A default internal encoding of `nil` tells us that text should not be transcoded automatically while reading and writing files, but be kept in the same encoding as the locale.

To see how these values can be influenced, we can change our locale via the `LANG` environment variable, and check what happens:

```
$ LANG="en_US.ISO8859-1" ruby encoding_checker.rb

Source encoding: #<Encoding:US-ASCII>
Default external: #<Encoding:ISO-8859-1>
Default internal: nil
Locale charmap: "ISO8859-1"
LANG environment variable: "en_US.ISO8859-1"
```

Here we see that rather than UTF-8, Latin-1 is used as the default external now. This means that if you modify this environment variable globally, Ruby will use it to determine what fallback encoding to use when dealing with I/O operations.

So on my system, where UTF-8 is the default external encoding, I can open files knowing that if I don't specify any particular character mapping, they will load in as UTF-8:

```
>> File.read("hello.txt")
=> "Hello, world\n"
>> File.read("hello.txt").encoding
=> #<Encoding:UTF-8>
```

On your system, your locale charmap will determine this. As long as your operating system is properly configured, Ruby should have a sensible default encoding for use in I/O operations.

Although this is very handy, one thing to keep in mind is that the locale settings on your system do not affect the default source encoding. This means that for all the source files you create, you need to set the actual source encoding via magic comment. This restriction is actually a good thing, as it makes it hard to make your Ruby programs so fragile that a change to `LANG` can break them. However, there is an exception in which allowing the locale to influence the source can be a good thing, and that's for quick one-liners.

Whenever we use `ruby -e`, the purpose is to do some sort of quick, one-off task. Those who use this technique often might have already found that there are all sorts of shortcuts that can be used when running in this mode. As it turns out, this applies to m17n as well:

```
$ ruby -e "p __ENCODING__"
#<Encoding:UTF-8>
```

Rather than falling back to ASCII, we can see that it infers the source encoding from the locale. To further illustrate that, we can reset `LANG` to Latin-1 again:

```
$ LANG="en_US.ISO8859-1" ruby -e "p __ENCODING__"
#<Encoding:ISO-8859-1>
```

As we can see, the source encoding is taken to be whatever our system is using for its locale, rather than requiring an explicit setting. This makes a lot of sense, because when we write one-liners, we typically want to use whatever our terminal is using as an encoding. It would also seem quite strange to embed a magic comment into a one-line script.

Because `ruby -e` will also infer the default external encoding in the same way that we've shown before when dealing with actual scripts, we end up with a powerful tool on our hands.

As you can see, Ruby's locale-based fallbacks make it possible in some cases to just ignore the details of the m17n system, or if necessary, to set an environment variable once and forget about it. The fact that `ruby -e` uses a locale-based source encoding means that you can also continue using Ruby as a command-line tool without too much friction. But now that we've looked at how Ruby's character mappings can be influenced by external means, we should take a look at how we can get a bit more fine-grained control over these things at the Ruby level.

Customizing Encoding Defaults

If we want to be explicit about our default external encoding, we can set it within our files via a simple accessor. In this example, we're indicating that the files we work with should be treated as Latin-1 by default, even though our script is encoded in UTF-8:

```
# coding: UTF-8
Encoding.default_external = Encoding.find("ISO-8859-1")

data = File.read("hello.txt")
p data.encoding #=> #<Encoding:ISO-8859-1>
```

If we plan to transcode our data upon load, we can also set the `default_internal` encoding. In the following example, we specify that I/O operations should by default be transcoded using the source encoding:

```
# coding: UTF-8
Encoding.default_external = Encoding.find("ISO-8859-1")
Encoding.default_internal = __ENCODING__

data = File.read("hello.txt")
p data.encoding #=> #<Encoding:UTF-8>
```

Here we see that the text loaded from the file ends up encoded in UTF-8. However, this is done in a safe way. The file is first loaded in using the `default_internal` encoding and then translated to the `default_external` encoding. To illustrate this further, the previous example is functionally equivalent to this more explicit one:

```
# coding: UTF-8

data = File.read("hello.txt", encoding: "ISO-8859-1:UTF-8")
p data.encoding #=> #<Encoding:UTF-8>
```

The difference is that when you set `default_external` and `default_internal`, the changes apply globally. Because this is a somewhat invasive change to make, you can see that messing with the internal and external encodings is reasonable to do only in scripting applications, since it does not apply well to code that needs to be portable.

However, in cases where it is reasonable to make this change globally, it can be used to avoid repeating encodings incessantly every time you do I/O operations. You can even specify these options on the command line, to allow them to be specified at the time the script is run:

```
$ ruby -E iso-8859-1:utf-8 hello.rb
#<Encoding:UTF-8>
```

Now our script is almost entirely devoid of explicit m17n details:

```
# coding: UTF-8

data = File.read("hello.txt")
p data.encoding #=> #<Encoding:UTF-8>
```

This is about as far as we'll want to take it in most cases, but there is one more example worth sharing before we move on.

As you may know, Ruby 1.8 did not have a comprehensive m17n system like Ruby 1.9 does. However, it did ship with an ability to enable a sort of UTF-8 mode, which would tell the interpreter that the strings it was working with as well the data pulled in from external files should be interpreted as UTF-8 text. This was provided by the *kcode* system, and specified via the `-Ku` flag. As it turns out, this works on Ruby 1.9.1 and even allows you to modify the source encoding of your files.

If we run the `encoding_checker.rb` file from before using `-Ku`, here's what you can expect to see:

```
$ ruby -Ku encoding_checker.rb
Source encoding: #<Encoding:UTF-8>
Default external: #<Encoding:UTF-8>
Default internal: nil
Locale charmap: "UTF-8"
LANG environment variable: "en_US.UTF-8"
```

Your locale settings may be different, but you should see that the source encoding and default external encoding have been set to UTF-8. Although this feature is primarily for backward compatibility, it is worth knowing about for a good "set it and forget it" default if you plan to work exclusively with UTF-8 in your scripts. Of course, if you have to work in most other encodings, you'll need to use some combination of the techniques we've covered earlier in this section, so this is far from a magic bullet.

Now that we've covered a few different ways to simplify your scripts to get some of the m17n roadblocks out of the way, here's a recap of the most important details to remember before we move on:

- The `LANG` environment variable that specifies your system locale is used by Ruby to determine the default external encoding of files. A properly set locale can allow Ruby to automatically load files in their native encodings without explicitly stating what character mapping they use.
- Although magic comments are typically required in files to set the source encoding, an exception is made for `ruby -e`-based command-line scripts. The source encoding for these one-liners is determined by locale. In most cases, this is what you will want.
- You can specify a default internal encoding that Ruby will automatically transcode loaded files into when no explicit internal encoding is specified. It is often reasonable to set this to match the source encoding in your scripts.
- You can set default external/internal encodings via the command-line switch `-Eexternal:internal` if you do not want to explicitly set them in your scripts.
- The `-Ku` flag still works for putting Ruby into "UTF-8" mode, which is useful for backward compatibility with Ruby 1.8.

- All of the techniques described in this section are suitable mostly for scripts or private use code. It is a bad idea to rely on locale data or manually set external and internal encodings in complex systems or code that needs to run on machines you do not have control over.

We've covered a lot of ground so far, and we'll be wrapping up with m17n to move on to L10n in just a short while. However, before we do, I'd like to cover just a couple short notes on how Ruby's new m17n system impacts lower-level code, and how to get around the issues it creates.

m17n-Safe Low-Level Text Processing

In previous versions of Ruby, strings were pretty much sequences of bytes rather than characters. This meant the following code seldom caused anyone to bat an eyelash:

```
File.open("hello.txt") { |f|
  loop do
    break if f.eof?
    chunk = "CHUNK: #{f.read(5)}"
    puts chunk unless chunk.empty?
  end
}
```

The purpose of the previous example is to print out the contents of the file in chunks of five bytes, which, when it comes to ASCII, means five characters. However, multibyte character encodings, especially variable-length ones such as UTF-8, cannot be processed using this approach. The reason is fairly simple.

Imagine this code running against a two-character, six-byte string in UTF-8 such as "吴佳". If we read five bytes of this string, we end up breaking the second character's byte sequence, resulting in the mangled string "吴\xE4\xBD". Of course, whether this is a problem depends on your reason for reading a file in chunks.

If we are processing binary data, we probably don't need to worry about character encodings or anything like that. Instead, just we read a fixed amount of data according to our needs, processing it however we'd like. But many times, the reason why we read data in chunks is not to process it at the byte level, but instead, to break it up into small parts as we work on it.

A perfect example of this, and a source of a good solution to the problem, is found within the CSV standard library. As we've seen before, this library is fully m17n-capable and takes great care to process files in a encoding-agnostic way. However, it also tries to be clever about certain things, which makes m17n support a bit more challenging.

Rather than assume `\n` is the default line ending to separate rows within a CSV, the library tries to determine what the line endings of a file are from a list of possibilities by examining a file in chunks. It cannot accomplish this task by reading in a single line of text, because it does not know the line endings yet. It would be highly inefficient to

try to read in the whole file to determine the endings, because CSV data can become huge. Therefore, what is needed is a way to process the data in chunks while searching for a line ending that does not break multibyte characters.

The solution is actually relatively simple, so we'll take a look at the whole thing first and then discuss it in a little more detail:

```
def read_to_char(bytes)
  return "" if @io.eof?
  data = @io.read(bytes)
  begin
    encoded = encode_str(data)
    raise unless encoded.valid_encoding?
    return encoded
  rescue  # encoding error or my invalid data raise
    if @io.eof? or data.size >= bytes + 10
      return data
    else
      data += @io.read(1)
      retry
    end
  end
end
```

Here, `@io` is an `IO` object of some sort, typically a `File`, and the `encode_str()` method is the same function we covered toward the beginning of this chapter, which is a thin wrapper over `String#encode()`. If we walk through this step by step, we see that an empty string is returned if the stream is already at `eof?`. Assuming that it is not, a specified number of `bytes` is read.

Then, the string is encoded, and it is checked to see whether the character mapping is valid. To clarify what `valid_encoding?` does, we can look at this simple example:

```
p "吴佳".valid_encoding?       #=> true
p "吴\xE4\xBD".valid_encoding? #=> false
```

When the encoding is valid, `read_to_char` returns the chunk, assuming that the string was broken up properly. Otherwise, it raises an error, causing the `rescue` block to be executed. Here, we see that the core fix relies on buffering the data slightly to try to read a complete character. What actually happens here is that the method gets retried repeatedly, adding one extra byte to the `data` until it either reaches a total of 10 bytes over the specified chunk size, or hits the end of the file.

The reason why this works is that every encoding Ruby supports has a character size of less than 10 bytes. This is in fact a conservative estimate, but is sufficiently small to still be reasonable. Using this method, it is possible to process data in chunks in an encoding-safe way.

The code that we looked at here was pulled directly from the *csv* library, but it would be easy to tweak for your individual needs. The main idea is basically that we need to consume some extra bytes to complete a character sometimes, and we can determine whether this is necessary based on either an error from `String#encode()` or the state of

`String#valid_encoding?`. I'll leave the generalization of `read_to_char` as an exercise for you, as its implementation will likely depend on the context, but this code from CSV is a solid conceptual base to start from.

I'd like to wrap up this section by pointing out one other thing to remember about low-level operations on strings when it comes to m17n. If you're used to thinking of bytes as character codes, you'll need to rethink the way you are doing things. You might be used to getting character codes via `String#unpack`:

```
>> "Hello world".unpack("C*")
=> [72, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]
```

Though this can be made to work for a handful of encodings, if you need to keep your code character-mapping-agnostic, you'll want to actually use `String#ord` instead here:

```
>> "Hello world".chars.map(&:ord)
=> [72, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]
```

This will work fine with any encoding you work with, giving back the ordinal value based on the actual encoding rather than the byte value:

```
>> utf8_string
=> "吴佳"
>> utf8_string.chars.map(&:ord)
=> [21556, 20339]
```

There is more we could talk about here, but I'd rather not go through the gory details exhaustively. The underlying theme of working with low-level text operations in an m17n-safe way is that characters are not necessarily equivalent to bytes. If you remember that simple idea, most of the rest of the core ideas will fall into place.

Now that we've talked quite a bit about how to actually process text in a wide variety of character mappings, we'll discuss a slightly higher-level topic. If you are going to support multiple languages in your application, you'll want a clean and organized way to do it. The next section will give you some tips on how to accomplish that without too much of a hassle.

Localizing Your Code

To make an application truly multilingual, we need to do more than just be able to process and produce text in different languages. We also need to provide a way for the user interface to be translated into the natural languages we wish to support. If you wanted to support even a small handful of languages, neither building multiple versions of your application nor writing tons of special casing wherever text is displayed will be a maintainable solution. Instead, what is needed is a way to mark the relevant sections of text with meaningful tags that can then be altered by external translation files. This is the process of localization (L10n).

We're going to be looking at a tiny L10n package I put together when I realized there weren't any Ruby 1.9-based tools available that worked outside of Rails. It is called

`Gibberish::Simple`, and is a fork of the very cool `Gibberish` plug-in by Chris Wanstrath. The main modifications I made were to port the library to Ruby 1.9.1, remove all dependencies including the Rails integration, and change the system a bit so that it does not depend on core extensions. Other than that, all of the hard work was done by Chris and he deserves all the credit for the actual system, which, as you'll see in a minute, is quite easy to use.

We're going to first look at a very trivial web application using the Sinatra web framework. It implements the children's game "Rock, Paper, Scissors."† We'll look at it both before and after localization, to give you a feel for the actual process of localizing your text.

The entire application is dirt-simple, with only two screens. The first lets you choose from "Rock," "Paper," and "Scissors" for your weapon (Figure 7-1).

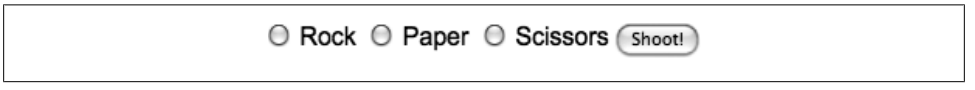

Figure 7-1. Choosing a weapon

The ERB template used to generate this view is trivial, as you might expect:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <form method="post">
    <% ["Rock", "Paper", "Scissors"].each do |weapon| %>
      <input type="radio" name="weapon" value="<%= weapon %>">
        <%= weapon %>
      </input>
    <% end %>
    <input type="submit" value="Shoot">
  </form>
</body>
</html>
```

Here we're just doing some simple dynamic form generation, and even a basic understanding of HTML should be sufficient to understand what's going on, so I won't go into details.

After submitting this form data, the game logic is done behind the scenes and the opponent's choice is revealed (Figure 7-2).

† See *http://en.wikipedia.org/wiki/Rock-paper-scissors*.

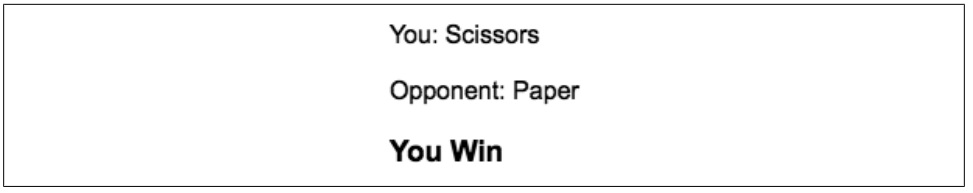

Figure 7-2. Announcing the winner

This ERB template is even more simple than the last one, as it just displays some pre-generated data:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <p> You: <%= @player_weapon %></p>
  <p> Opponent: <%= @opponent_weapon %></p>
  <h3><%= end_game_message %></h3>
</body>
</html>
```

Tying the whole application together is the actual Sinatra application. It handles the game logic and generates the appropriate messages, but is not in any way complex. Here's the whole thing, which consists of nothing more than two URI handlers and a helper function that determines who won:

```
require "sinatra"

get '/rps' do
  erb :rps_start
end

post '/rps' do
  @player_weapon    = params[:weapon]
  @opponent_weapon  = %w[Rock Paper Scissors].sample
  erb :rps_end
end

helpers do
  def end_game_message
    return "It's a tie" if @player_weapon == @opponent_weapon

    winning_combos = [["Paper","Rock"],["Rock","Scissor"],["Scissors","Paper"]]
    if winning_combos.include?([@player_weapon, @opponent_weapon])
      "You Win"
    else
      "You Lose"
    end
  end
end
```

Although tiny, this does represent a fully functional application. The fact that we're presenting an interface for users to interact with means that interface can be localized. We'll now look at how to go about adding two languages, French and Chinese.

One option would be to code up separate views and some specialized translation logic for each language, but that approach doesn't scale particularly well. Instead, what we can do is come up with unique identifiers for each text segment in our application, and then create translation files that fill in the appropriate values depending on what language is selected. Ignoring how we actually integrate these translation files into our application for the moment, we can look at a couple of them to get a feel for how they look.

The following simple YAML file covers all the necessary Chinese translations for this application:

```
win: 你赢了
lose: 你输了
tie: 平局

rock: 石头
paper: 布
scissors: 剪刀

shoot: 开
you: 你
opponent: 对手
```

As you can see, these are direct mappings between English identifiers for text that appears in our application and their Chinese translations. Because all of the translation files will follow this same basic structure, you'll notice that the French translation file looks pretty much the same:

```
win: Tu gagnes
lose: Tu perds
tie: Egalité

rock: Caillou
paper: Feuille
scissors: Ciseaux

shoot: On y va !
you: Toi
opponent: Adversaire
```

With some translation files in hand, we can work on integrating `Gibberish::Simple` into our application. Before we do that, we'll look at a simple script that shows the basic mechanics of how things work. In the following example, it is assumed that *lang/cn.yml* and *lang/fr.yml* exist and are populated with the values here:

```
# coding: UTF-8

require "gibberish/simple"
```

```
# Tell Gibberish that lang/ exists in the same root directory as this file
Gibberish::Simple.language_paths << File.dirname(__FILE__)

# Let us use the T() translation helper globally
include Gibberish::Simple

p T("You Win", :win) #=> "You Win"

Gibberish::Simple.use_language(:fr) do
  p T("You Win", :win) #=> "Tu gagnes."
end

Gibberish::Simple.use_language(:cn) do
  p T("You Win", :win) #=> "你贏了"
end

# Because there is no matching file, this falls back to the defaults

Gibberish::Simple.use_language(:en) do
  p T("You Win", :win) #=> "You Win"
end
```

Here we see that our text display calls are wrapped in a method call to `T()`, which stands for translate. When no language is specified, this code displays the default value that is specified by the first argument. When a language is specified via the `use_language` block, `T()` will look up the appropriate translation via the unique tag for it. If `use_language` is given a language that does not have a matching YAML file, the defaults are reverted to. As you can see, for these basic needs, there isn't a whole lot to it.

Now, we're ready to take a look at the localized version of "Rock, Paper, Scissors." We can start with the main application and work our way out to the views. Though some changes were necessary, you'll see that the code is still pretty easy to follow:

```
require "sinatra"
require "gibberish/simple"

Gibberish::Simple.language_paths << File.dirname(__FILE__)
include Gibberish::Simple

get '/rps' do
  redirect '/rps/en'
end

get '/rps/:lang' do
  erb :rps_start
end

post '/rps/:lang' do
  @player_weapon    = params[:weapon]
  @opponent_weapon  = %w[Rock Paper Scissors].sample
  erb :rps_end
end

helpers do
```

```
  def end_game_message
    return T("It's a tie", :tie) if @player_weapon == @opponent_weapon

    winning_combos = [["Paper","Rock"],["Rock","Scissor"],["Scissors","Paper"]]
    if winning_combos.include?([@player_weapon, @opponent_weapon])
      T("You Win", :win)
    else
      T("You Lose", :lose)
    end
  end

  def weapon_name(weapon)
    T(weapon, weapon.downcase.to_sym)
  end

  def translated(&block)
    Gibberish::Simple.use_language(params[:lang], &block)
  end
end
```

The first thing to notice is that I've decided to embed the language choice into the URI. This is the code that does that:

```
get '/rps' do
  redirect '/rps/en'
end

get '/rps/:lang' do
  erb :rps_start
end
```

You can see that going to `/rps` actually redirects you to `/rps/en`, which represents the English version of the game. The need for this code is mainly a consequence of the fact that we're building something web-based without a database backend. In other applications, you could store the current language in whatever way makes the most sense for the individual solution. The key idea here is only that we need to be able to tell `Gibberish::Simple` what language we want to work in. You'll see how it's used in a moment, but this `:lang` parameter is used by our `translated` helper to set the current language from within a view:

```
def translated(&block)
  Gibberish::Simple.use_language(params[:lang], &block)
end
```

The rest of the remaining changes are simply wrapping anything that will eventually be displayed to the user in `T()` calls. Notice this does not affect the actual logic of our code, which still works in terms of comparing "Rock," "Paper," and "Scissors" regardless of the language we're displaying the UI in. This is an important aspect of localizing your code: you want to do it as late as possible so that your business logic is not affected by translations. The `weapon_name` helper serves exactly this purpose:

```
def weapon_name(weapon)
  T(weapon, weapon.downcase.to_sym)
end
```

Beyond the additions of a couple helpers and some calls to `T()`, much of the code is left unchanged. The more significant work involved with localizing this application is done at the view level. What follows is the template from the first screen, the weapon selection form:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <form method="post">
    <% translated do %>
      <% ["Rock", "Paper", "Scissors"].each do |weapon| %>
        <input type="radio" name="weapon" value="<%= weapon %>">
          <%= weapon_name(weapon) %>
        </input>
      <% end %>
      <input type="submit" value="<%= T('Shoot!',:shoot) %>">
   <% end %>
  </form>
</body>
</html>
```

Here, we see our helpers from the main application in action. The `translated` block is just syntactic sugar that infers from the URI which language to pass to `Gibberish::Simple.use_langauge`. Within this block, every string displayed to the user must be directly or indirectly passed through `T()` to be translated. However, we explicitly leave the values of the actual parameters untranslated, allowing our basic game logic to remain unmodified.

The second view is a bit easier because, as mentioned before, it's strictly displaying information:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <% translated do %>
    <p><%= T("You", :you) %>: <%= weapon_name(@player_weapon) %></p>
    <p><%= T("Opponent", :opponent) %>: <%= weapon_name(@opponent_weapon) %></p>
    <h3><%= end_game_message %></h3>
  <% end %>
</body>
</html>
```

Here, we're just looking up a few more tagged text segments, so there's nothing new to worry about. With these three files modified, heading to the `/rps/en` URI gives you the same screens shown at the beginning of the chapter.

When we hit `/rps/cn`, we get the screen for weapon selection (Figure 7-3).

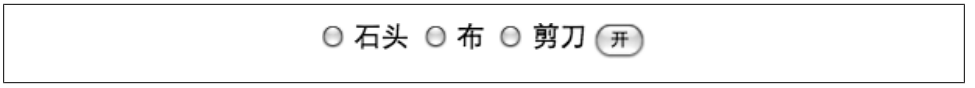

Figure 7-3. Weapon selection (Chinese)

Pressing the button brings us to the results screen (Figure 7-4).

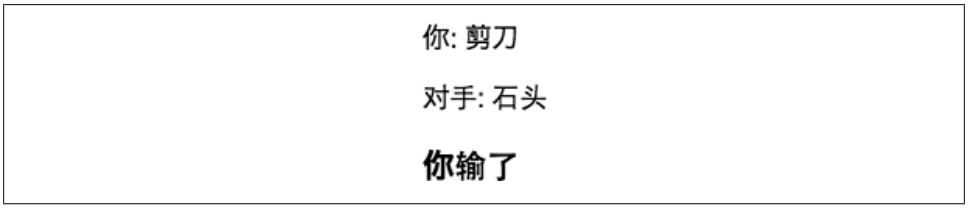

Figure 7-4. Announcing the winner (Chinese)

When we switch to the `/rps/fr` URI, we get to pick our weapons in French (Figure 7-5).

Figure 7-5. Weapon selection (French)

And, of course, we can see our final results as well (Figure 7-6).

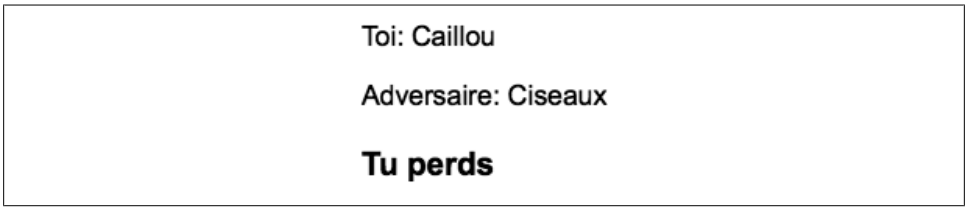

Figure 7-6. Announcing the winner (French)

At this point, we can take full advantage of the fact that the application has been localized. Adding new translations is as easy as dropping new YAML files into the `lang` directory. This is clear evidence of why it is better to offer general localization support than it is to offer a "Chinese Version" and "French Version" and however many other versions of your application you can imagine. This simple approach lets you easily customize the text used in your interface without much modification to your underlying business logic.

Of course, the example we've shown here is about as simple as you might imagine. In more complicated cases, you may have phrases where some text needs to be in a different order, depending on which language you are dealing with. This is especially

common with things like names. To deal with things like this, you can create templates that allow for substitutions. We'll run through a simple example before wrapping things up here, just to give you an idea of how to work with situations like this.

In American English, names are typically represented as "Given Surname," such as "Gregory Brown." Here's how we'd express that as a default in `Gibberish::Simple`:

```
data = { given_name: "Gregory", surname: "Brown" }
p T("{given_name} {surname}",  [:name, data]) #=> "Gregory Brown"
```

If we want to invert this order, we can easily do so in our templates. Here we've added a `:name` tag to the *lang/cn.yml* file that demonstrates how this is done:

```
name: "{surname}{given_name}"
```

Now, when we are dealing with a Chinese name, you can see that it gets composed with the last name first, and no space separating the last and first name:

```
data = { given_name: "佳", surname: "吴" }
Gibberish::Simple.use_language(:cn) do
  p T("{given_name} {surname}", [:name, data]) #=> "吴佳"
end
```

As you might imagine, this technique can be used to cover a lot of ground, providing substantial flexibility in how you display your text segments. Not only can each language have its own substitutions for text, but it can also control the order in which it is presented.

What we've covered so far should sufficiently cover most ordinary localization needs, so we'll use this as a convenient stopping point. Although I certainly recommend taking a look at `Gibberish::Simple`, most of the ideas we've covered here apply to any generalized L10n strategy you might implement. When it comes down to it, you need to remember only a few things:

- The first step in localizing an application is identifying the unique text segments that need to be translated.
- A generalized L10n system provides a way to keep all locale-specific content in translation files rather than tied up in the display code of your application.
- Every string that gets displayed to the user must be passed through a translation filter so that it can be customized based on the specified language. In `Gibberish::Simple`, we use `T()` for this; other systems may vary.
- Translation should be done at as late a stage as possible, so that L10n-related modifications to text do not interfere with the core business logic of your program.
- In many cases, you cannot simply interpolate strings in a predetermined order. `Gibberish::Simple` offers a simple templating mechanism that allows each translation file to specify how substrings should be interpolated into a text segment. If you roll your own system, be sure to keep this in consideration.

- Creating helper functions to simplify your translation code can come in handy when generating dynamic text output. For an example of this, go back and look at how `weapon_name()` was used in the simple Sinatra example discussed here.
- Because adding individual localization tags can be a bit tedious, it's often a good idea to wait until you have a fully fleshed-out application before integrating a general L10n system, if it is possible to do so.

If you keep these ideas in mind, you'll have no trouble integrating L10n support into your applications. Once a system is in place, it is quite cool to see how quickly translation files can change the overall interface to support different languages on the fly.

Conclusions

The process of multilingualization and localization is something that has been overlooked by programmers for far too long. This is mostly due to the fact that accomplishing any level of globalized support in software systems was until fairly recently a highly complex task. The demand for such functionality was also considerably lower before networked software dominated our ecosystem, due to the fact that software user bases typically were not globalized themselves.

In 2009, it is a whole different scene. If we want to write software that can be used comfortably by people all around the world, we need to rise to the occasion and make our code capable of speaking (or at least processing) the myriad collection languages that people are comfortable with. With Ruby 1.9, we have a powerful system for writing code that respects the cultural influences of our users, and we should take advantage of it whenever we can. The techniques shown in this chapter will help you make your software more accessible, whether it is open source or commercial. However, this chapter does not attempt to teach m17n, L10n in general, or the gritty details of how everything fits together in the context of Ruby. I strongly encourage you to read up on those topics before trying to apply any of the ideas you've gained here.