CHAPTER 6

Use Symbols to Stand for Something

I have to admit that I tend to be a bit anthropomorphic about the technologies I work with. I just can't help but think of all those complex piles of software as somehow alive, each with its own personality—sometimes friendly, sometimes not. Early in my career I imagined FORTRAN as a grouchy old camel—capable of carrying a huge load, but fairly ugly and not a creature you would want to turn your back on. Later on I had this mental image of the -> operator in the C programming language (it dereferences pointers) as an arrow in flight: also very powerful, also nothing to mess with. These days, the colon that precedes every Ruby symbol always makes me think of the eyes peering out from the tilted head of a confused but friendly dog. The key word here is confused—symbols probably have the dubious distinction of being the one bit of syntax that perplexes the greatest number of new Ruby programmers.

In this chapter I am going to try to stamp out all of that confusion and show symbols for what they really are: very simple, useful programming language constructs that are a key part of the Ruby programming style. So let's get started and see why symbols are such handy little mutts to have around.

The Two Faces of Strings

Sometimes a good way to explain a troublesome topic is to engage in a little creative fiction. You start out with an oversimplified explanation and, once that has sunk in a

bit, you work your way from there back to the real world. In this spirit, let's start our exploration of symbols with a slight simplification: Symbols are really just strings. This is not as far fetched as it sounds: Think about the string `"dog"` and its closest symbolic cousin, `:dog`. The thing that hits you in the face about these two objects is that they both are essentially three characters: a "d", an "o", and a "g".

Strings and symbols are also reasonably interchangeable in real life code: Take this familiar example of some ActiveRecord code, which finds all of the records in the books table:[1]

```
book = Book.find(:all)
```

The argument to the `find` method is simply a flag, there to tell `find` that we want all of the records in the `books` table—not just the first record, not just the last record, but all of them. The actual value that we pass into `Book.find` doesn't really matter very much. We might imagine that if we had the time and motivation, we could go into the guts of ActiveRecord and rewrite the code so that we could use a string to signal that we wanted *all* the books:

```
book = Book.find('all')
```

So there is my simplified explanation of symbols: Other than the fact that typing `:all` requires one less keystroke than typing `'all'`, there is not really a lot to distinguish a symbol from a string. So why does Ruby give us both?

Not Quite a String

The answer is that we tend to use strings of characters in our code for two rather different purposes: The first, and most obvious, use for strings is to hold some data that we are processing. Read in those `Book` objects from the database and you will very likely have your hands full of string data, things like the title of the book, the author's name, and the actual text.

The second way that we use strings of characters is to represent things in our programs, things like wanting to find *all* of the records in a table. The key thing about

1. If you are not familiar with ActiveRecord, don't worry. In ActiveRecord there is a class for each database table. In our example we have the (unseen) `Book` class that knows about the `books` table. Every ActiveRecord table class has a class method called `find`, which takes various arguments telling the method for what it should search.

`:all` in our `Book` ActiveRecord example is that ActiveRecord can recognize it when it sees it—the code needs to know which records to return, and `:all` is the flag that says it should return every one. The nice thing about using something like `:all` for this kind of "stands for" duty is that it also makes sense to the humans: You are a lot more likely to recognize what `:all` means when you come across it than `0`, or `-1`, or even (heaven forbid!) `0x29ef`.

These two uses for strings of characters—for regular data processing tasks on the one hand and for internal, symbolic, marker-type jobs on the other—make very different demands on the objects. If you are processing data, you will want to have the whole range of string manipulation tools at your fingertips: You might want the first ten characters of the title, or you might want to get its length or see whether it matches some regular expression. On the other hand, if you are using some characters to stand for something in your code, you probably are not very interested in messing with the actual characters. Instead, in this second case you just need to know whether this thing is the flag that tells you to find all the records or just the first record. Mainly, when you want some characters to stand for something, you simply need to know if *this* is the same as *that*, quickly and reliably.

Optimized to Stand for Something

By now you have probably guessed that the Ruby `String` class is optimized for the data processing side of strings while symbols are meant to take over the "stands for" role—hence the name. Since we don't use symbols for data processing tasks, they lack most of the classic string manipulation methods that we talked about in Chapter 4. Symbols do have some special talents that make them great for being symbols. For example, there can only ever be one instance of any given symbol: If I mention `:all` twice in my code, it is always exactly the same `:all`. So if I have:

```
a = :all
b = a
c = :all
```

I know that `a`, `b`, and `c` all refer to exactly the same object. It turns out that Ruby has a number of different ways to check whether one object is equal to another,[2] but

2. For more on object equality, see Chapter 12.

with symbols it doesn't matter: Since there can only be one instance of any given symbol, `:all` is always equal to itself no matter how you ask:

```
# True! All true!

a == c
a === c
a.eql?(c)
a.equal?(c)
```

In contrast, every time you say `"all"`, you are making a brand new string. So if you say this:

```
x = "all"
y = "all"
```

Then you have manufactured two different strings. Since both the strings happen to contain the same three characters, the two strings are equal in some sense of the word, but they are emphatically not identically the same object. The fact that there can only be one instance of any given symbol means that figuring out whether *this* symbol is the same as *that* symbol is not only foolproof, it also happens at lightning speeds.

Another aspect of symbols that makes them so well suited to their chosen career is that symbols are immutable—once you create that `:all` symbol, it will be `:all` until the end of time.[3] You cannot, for example, make it uppercase or lob off the second `'l'`. This means that you can use a symbol with confidence that it will not change out from under you.

You can see all these issues at play in hashes. Since symbol comparison runs at NASCAR speeds and symbols never change, they make ideal hash keys. Sometimes, however, engineers want to use regular strings as hash keys:

```
author = 'jules verne'
title = 'from earth to the moon'
hash = { author => title }
```

3. Or at least until your Ruby interpreter exits.

So what would happen to the hash if you changed the key out from underneath it?

```
author.upcase!
```

The answer is that nothing will happen to the hash, because the `Hash` class has special defenses built in to guard against just this kind of thing. Inside of `Hash` there is special case code that makes a copy of any keys passed in if the keys happen to be strings. The fact that the `Hash` class needs to go through this ugly bit of special pleading precisely to keep you from coming to grief with string keys is the perfect illustration of the utility of symbols.

In the Wild

In practice, the line between symbols and regular strings is sometimes a bit blurry. It is, for example, trivially easy to turn a symbol into a string: You just use the ubiquitous `to_s` method:

```
the_string = :all.to_s
```

To go in the reverse direction, you can use the `to_sym` method that you find on your strings:

```
the_symbol = 'all'.to_sym
```

The blurriness between symbols and strings sometimes also extends into the minds of Ruby programmers. For example, every object in Ruby has a method called `public_methods`, which returns an array containing the names of all of the public methods on that object. Now, you might argue that method names are the poster children for objects that stand for something (in this case a bit of code), and therefore the `public_methods` method should return an array of symbols. But call `public_methods` in a pre-1.9 version of Ruby, like this:

```
x = Object.new
pp x.public_methods
```

And you will get an array of strings, not symbols:

```
["inspect",
 "pretty_print_cycle",
 "pretty_print_inspect",
 "clone",
 ...
]
```

Is there something wrong with our reasoning? Apparently not, because in Ruby 1.9 `public_methods` does indeed return an array of symbols:

```
[:pretty_print,
 :pretty_print_cycle,
 :pretty_print_instance_variables,
 :pretty_print_inspect,
 :nil?,
 ...
]
```

The lesson here is that if you find symbols a bit confusing, you seem to be in very good company.

Staying Out of Trouble

Given the curious relationship between symbols and strings, it probably will come as no surprise that the best way to screw up with a symbol is to use it when you wanted a string, and vice versa. As we have seen, you want to use strings for data, for things that you might want to truncate, turn to uppercase, or concatenate. Use symbols when you simply want an intelligible thing that stands for something in your code.

The other way to go wrong is to forget which you need at any given time. This seems to happen a lot when using symbols as the keys in hashes. For example, take a look at this code fragment:

```
# Some broken code

person = {}
person[:name] = 'russ'
```

```
person[:eyes] = 'misty blue'

# A little later...

puts "Name: #{person['name']} Eyes: #{person['eyes']}"
```

The code here is broken, but you might have to look at it a couple of times to see that the keys of the `person` hash are symbols, but the `puts` statement tries to use strings. What you really want to say here is:

```
puts "Name: #{person[:name]} Eyes: #{person[:eyes]}"
```

This kind of mistake is common enough that Rails actually provides a Band-Aid for it in the form of the `HashWithIndifferentAccess` class. This convenient, but somewhat dubious bit of code is a subclass of `Hash` that allows you to mix and match strings and symbols with cheerful abandon.

Wrapping Up

In this chapter we have looked at symbols and saw that they exist purely to stand for something in your code. Symbols and garden variety strings have a lot in common—both are mostly just a stretch of characters. Unlike strings, symbols are specially tuned to their "stands for" purpose: Symbols are both unique—there can only ever be one `:all` symbol in your Ruby interpreter—and immutable, so that `:all` will never change. The good news is that once you understand that symbols and strings are like two siblings—related, but with different talents—you will be able to take advantage of the things that each does best.