

Ruby Techniques for Rails Developers

SAMPLE CHAPTER

RUBY FOR RAILS

David A. Black

Foreword by David Heinemeier Hansson

 MANNING





Ruby for Rails
by David A. Black
Chapter 10

Copyright 2006 Manning Publications

brief contents

PART I THE RUBY/RAILS LANDSCAPE1

- 1 ■ How Ruby works 3
- 2 ■ How Rails works 33
- 3 ■ Ruby-informed Rails development 67

PART II RUBY BUILDING BLOCKS93

- 4 ■ Objects and variables 95
- 5 ■ Organizing objects with classes 121
- 6 ■ Modules and program organization 154
- 7 ■ The default object (self) and scope 177
- 8 ■ Control flow techniques 206

PART III BUILT-IN CLASSES AND MODULES231

- 9 ■ Built-in essentials 233
- 10 ■ Scalar objects 257
- 11 ■ Collections, containers, and enumerability 277

- 12 ■ Regular expressions and regexp-based string operations 312
- 13 ■ Ruby dynamics 337

PART IV RAILS THROUGH RUBY, RUBY THROUGH RAILS 369

- 14 ■ (Re)modeling the R4RMusic application universe 371
- 15 ■ Programmatically enhancing ActiveRecord models 392
- 16 ■ Enhancing the controllers and views 422
- 17 ■ Techniques for exploring the Rails source code 455
- appendix* ■ Ruby and Rails installation and resources 471

10

Scalar objects

In this chapter

- Strings
- Symbols
- Numerics
- Date and time objects

The term *scalar* means *one-dimensional*. Here, it refers to objects that represent single values, as opposed to collection or container objects that hold multiple values. There are some shades of gray: Strings, for example, can be viewed as collections of characters in addition to being single units of text. *Scalar*, in other words, is to some extent in the eye of the beholder. Still, as a good first approximation, you can look at the classes discussed in this chapter as classes of one-dimensional, bite-sized objects; doing so will help you as we move in the next chapter to the matter of collections and container objects.

The built-in objects we'll look at in this chapter include the following:

- *Strings*, which are Ruby's standard way of handling textual material of any length
- *Symbols*, which are another way of representing text in Ruby
- *Numerical objects*, including integers and floating-point numbers
- *Times and dates*, which Ruby handles (as it handles everything) as objects in their own right

The upshot of this chapter will be not only that you acquire some mastery of manipulating these objects, but also that you're positioned well to explore the containers and collections—which often contain and collect scalar objects—in the next chapter.

10.1 Working with strings

Ruby gives you two built-in classes that, between them, provide all the functionality of text: the `String` class and the `Symbol` class. We'll start with strings, which are the standard way to represent bodies of text of arbitrary content and length.

10.1.1 String basics

A *string literal* is generally enclosed in quotation marks:

```
"This is a string."
```

Single quotes can also be used:

```
'This is also a string.'
```

But a single-quoted string behaves very differently, in some circumstances, than a double-quoted string. The main difference is that *string interpolation* doesn't work with single-quoted strings. Try these two snippets, and you'll see the difference:

```
puts "Two plus two is #{2 + 2}."
puts 'Two plus two is #{2 + 2}.'
```

As you'll see if you paste these lines into `irb`, you get two very different results:

```
Two plus two is 4.
Two plus two is #{2 + 2}.
```

Single quotes disable the `#{...}` interpolation mechanism. If you need that mechanism, you can't use them.

In general, single- and double-quoted strings behave differently with respect to the need to *escape* certain characters with a backslash:

```
puts "Backslashes (\\) have to be escaped in double quotes."
puts 'You can just type \\ once in a single quoted string.'
puts "But whichever type of quotation mark you use..."
puts "You have to escape its quotation symbol, such as \"."
puts 'That applies to \' in single-quoted strings too.'
```

You can, if necessary, escape (and thereby disable) the string interpolation mechanism in a double-quoted string:

```
puts "Escaped interpolation: \"\#{2 + 2}\"."
```

You'll see other cases of string interpolation and character-escaping as we proceed. Meanwhile, by far the best way to get a feel for these behaviors firsthand is to experiment with strings in `irb`.

WARNING `irb` ALWAYS PRINTS OUT ITS EVALUATIONS When you use `irb` to familiarize yourself with string-quoting behaviors, keep in mind that every time you type an expression into `irb`, `irb` evaluates the expression and displays its string representation. This result can be confusing: String representations are double-quoted strings and therefore contain a lot of backslashes, for character-escaping purposes. The best thing to do is to use the `puts` command, so you can see what the string will look like on output. (When you do, the return value printed by `irb` is `nil`, because that's the return value of all calls to `puts`.)

Other quoting mechanisms

Ruby gives you several ways to write strings in addition to single and double quotation marks. But even when you're using these other techniques, keep in mind that a string is always either fundamentally single-quoted or double-quoted—even if quotation marks aren't physically present.

Table 10.1 summarizes Ruby's quoting mechanisms. The main reason Ruby provides mechanisms other than literal quotation marks (`%q` and `%Q`) is that they make it easier to write strings that contain quotation marks (or apostrophes, which are the same as single quotation marks).

Table 10.1 Summary of string quoting mechanisms

Token	Single- or double-quoted	Example	Print output
'	Single	'You\'ll have to "escape" single quotes.'	You'll have to "escape" single quotes.
"	Double	"You'll have to \"escape\" double quotes."	You'll have to "escape" double quotes.
%q	Single	%q{'Single-quoted' example—no escape needed.}	'Single-quoted' example—no escape needed.
%Q	Double	%Q{"Double-quoted" example—no escape needed..}	"Double-quoted" example—no escape needed.

The examples in table 10.1 use curly braces as delimiters for the strings. You can use almost any punctuation character. For example, the expression `%q.string.` represents the string “string”; the two periods serve as delimiters. As long as the second delimiter matches the first (in the sense of being the same or, in the case of braces, brackets, and parentheses, being the matching one), the delimiter pair will work. Curly braces, however, are more or less standard; unless your string contains a closing curly brace, it’s just as well to stick to that practice.

Representing strings is only the first stage. There’s also the matter of what you do *with* strings. We’ll turn now to an exploration of some of Ruby’s important string operations.

10.1.2 String operations

To put it non-technically, you can do a ton of stuff with strings. Here, we’ll look at a selection of string-manipulation methods.

It’s a good idea to keep the following general points in mind as we get deeper into the study of strings:

- Most of the string methods we’ll look at return a new `String` object, leaving the original string itself unchanged.
- A number of these methods, however, have bang versions that perform the change on the original string instead of returning a new string.
- A few non-bang methods perform changes on the original string. The names of these methods make it clear that this is happening (such as `replace`), even though there’s no `!` on the name.
- Some string methods return something other than a string—for example, the `to_i` (to integer) conversion method.

Another point to keep in mind is that discussion of several important string methods will be postponed until after we've looked at regular expressions in chapter 12. But we'll cover the bulk of the string ground here, and put strings through their paces: combining them, changing them, getting substrings from them, and more.

Combining two (or more) strings

There are several techniques for combining strings. These techniques differ as to whether the second string is permanently added to the first or whether a new, third string is created out of the first two—in other words, whether the operation changes the receiver.

To create a new string consisting of two or more strings, you can use the `+` operator (the syntactic sugar form of the `+` method) to run the original strings together. Here's what `irb --simple-prompt` has to say about adding strings:

```
>> "a" + "b"
=> "ab"
>> "a" + "b" + "c"
=> "abc"
```

The string you get back from `+` is always a new string. You can test this by assigning a string to a variable, using it in a `+` operation, and checking to see what its value is after the operation:

```
>> str = "Hi "
=> "Hi "
>> str + "there."
=> "Hi there." ← ❶
>> str
=> "Hi " ← ❷
```

The expression `str + "there."` evaluates to the new string “Hi there.” ❶ but leaves `str` unchanged ❷.

To add (append) a second string permanently to an existing string, use the `<<` method, which also has a syntactic sugar, pseudo-operator form:

```
>> str = "Hi "
=> "Hi "
>> str << "there."
=> "Hi there."
>> str
=> "Hi there." ← ❶
```

In this example, the original string `str` has had the new string appended to it, as you can see from the evaluation of `str` at the end ❶.

Another way to combine strings is through string interpolation:

```
>> str = "Hi "  
=> "Hi "  
>> "#{str} there."  
=> "Hi there."
```

The result is a new string: “Hi there.” String interpolation is a general-purpose technique, but you can use it for this kind of simple additive purpose, among others.

Replacing a string's contents

To replace the contents of a string, you use `replace`. Again, the examples here are geared for use in `irb`, where you're shown the value of each expression as you enter it:

```
>> str = "Hi there."  
=> "Hi there."  
>> str.replace("Good-bye.") ← ❶  
=> "Good-bye."  
>> str  
=> "Good-bye." ← ❷
```

The final value of `str` ❷ is “Good-bye.”, the string with which you have replaced ❶ `str`'s original contents. Keep in mind that replacing a string's contents isn't the same as creating a completely new string. `str` still refers to the same string, which means other variables referring to that string will also reflect the change:

```
>> str = "Hi there."  
=> "Hi there."  
>> x = str ← ❶  
=> "Hi there."  
>> str.replace("Good-bye.") ← ❷  
=> "Good-bye."  
>> x  
=> "Good-bye."
```

In this example, `str` and `x` refer to one and the same string object; that's established when you assign `str` to `x` ❶. When that one and only string object has its contents replaced via a method call on `str` ❷, the string's new contents are also reflected in `x`.

`replace` thus lets you change a string in such a way that all existing references to it (variables) still refer to the same string. It's an example of a non-bang method that changes an object in place. The name, `replace`, conveys this fact, without the need for the exclamation point. (Also, a bang method usually exists in a pair with a non-bang version, and it's impossible to imagine what “replacing the contents of a string object” without changing the string would even mean.)

We'll look next at several useful methods for manipulating and massaging strings. We won't examine everything that strings can do, but we'll discuss some of the most important string facilities and behaviors in Ruby.

Massaging strings

Ruby strings have a number of methods, all with logical names, that let you massage and tweak strings. Some of the most common are summarized in table 10.2. All of these methods have bang (!) equivalents so that you can perform the operation in place on an existing string via a variable.

Table 10.2 Miscellaneous string manipulations

Method	Example	Result
capitalize	"ruby".capitalize	"Ruby"
upcase	"cobol".upcase	"COBOL"
downcase	"UNIX".downcase	"unix"
swapcase	"rUBY".swapcase	"Ruby"
strip	" lose the outer spaces "	"lose the outer spaces"
lstrip	" lose the left spaces "	"lose the left spaces "
rstrip	" lose the right spaces "	"lose the right spaces"
chop	"remove last character"	"remove last characte"
chomp	"remove training newline\n"	"remove trailing newline"
reverse	" gnirts eht esrever"	"reverse the string"

As you'll see if you choose any of these methods and try it in irb, the non-bang version returns a new string, and the bang version modifies the old string in place. Here's an example, using `reverse` and its bang counterpart:

```
>> str = "Hello"
=> "Hello"
>> str.reverse ← ❶
=> "olleH"
>> str
=> "Hello"
>> str.reverse! ← ❷
=> "olleH"
>> str
=> "olleH"
```

The first reverse operation ❶ reverses the string; `irb` reports the value of the expression as “olleH”. But the string is still “Hello”, as you can see when you ask `irb` to show you the value of `str`. The bang version, `reverse!` ❷, *does* change the original string permanently—as you can see, again, by asking `irb` to display `str`.

Meanwhile, we'll look next at working with substrings and individual characters.

Grabbing characters and substrings

Strings come with a pair of get/set methods: the ubiquitous `[]` and `[]=` methods. To grab the *n*th character of a string, you use `[]` with an index (starting at zero). But beware: You get back a number, not a character. Specifically, you get the character's ASCII value. For example, here's how to get the ASCII value of the character “c”:

```
>> "abc"[2]
=> 99
```

You can turn this number back into a character with the `chr` method:

```
>> "abc"[2].chr
=> "c"
```

You can also use a negative index. If you do, the index is counted from the right side of the string:

```
>> "abc"[-2].chr
=> "b"
```

(You'll see more negative, right-hand indexing when we look in detail at arrays in chapter 11.)

You can grab a substring of a string by giving two arguments to `[]`, in which case the first argument is the starting index and the second argument is the length of the substring you want. For example, to get a four-character substring starting at the sixth character (remember, strings are zero-indexed), you do this:

```
>> "This is a string"[5,4]
=> "is a"
```

TIP USING SUBSTRING SYNTAX TO GET ONE CHARACTER Because you can grab substrings of any length using the two-argument form of `String#[]`, you can grab any one character (without having to convert it back from an ASCII value) by requesting a substring of length one: for example, `"abc"[2,1]` is “c”.

The string set method `[]=` works the opposite way from `[]`: It changes the string (in place) by inserting the substring you specify into the position you give. It also has a two-argument form. Here it is in action:

```
>> s = "This is a string."
=> "This is a string."
>> s[-1] = "!" ← ❶
=> "!"
>> s
=> "This is a string!"
>> s[2,2] = "at" ← ❷
=> "at"
>> s
=> "That is a string!"
```

This example includes two set operations; after each one, we print out the string. The first ❶ changes the string's last character from `.` (period) to `!` (exclamation point). The second ❷ changes the third and fourth characters from `is` to `at`. The result is that evaluating `s` now results in “That is a string!”

These techniques give you fine-grained control over the contents of strings, enabling you to do just about any manipulation you're likely to need.

This survey has given you a good foundation in string manipulation, although by no means have we exhausted the topic. Here, as usual, `irb` is your friend. Test things, experiment, and see how the string methods interact with each other.

Meanwhile, we're going to move on to the matter of string comparisons.

10.1.3 Comparing strings

As you know, Ruby objects can be compared in numerous ways; what the comparisons mean, as well as which are available, varies from object to object. Strings have a full set of comparison capabilities; strings are *comparable*, in the technical sense that the class `String` mixes in the `Comparable` module.

We'll look here at the various kinds of comparisons you can perform between one string and another.

Comparing two strings for equality

Like Ruby objects in general, strings have several methods for testing equality. The most common one is `==` (double equals sign), which comes with syntactic sugar allowing you to use it like an operator. This method tests for equality of string content:

```
>> "string" == "string"
=> true
>> "string" == "house"
=> false
```

The two literal `"string"` strings are different objects, but they have the same content. Therefore, they pass the `==` test.

Another equality-test method, `String#eq1?`, tests two strings for identical content. In practice, it usually returns the same result as `==`. (There are subtle differences in the implementations of these two methods, but you can use either. You'll find that `==` is more common.) A third method, `String#equal?`, tests whether two strings are the same object:

```
>> "a" == "a"
=> true
>> "a".equal?("a")
=> false
```

The first test succeeds because the two strings have the same contents. The second test fails, because the first string isn't the same object as the second string. This is a good reminder of the fact that strings that appear identical to the eye may, to Ruby, have different object identities.

String comparison and ordering

As officially comparable objects, strings define a `<=>` method; hanging off this method are the usual comparison pseudo-operators (the methods whose syntactic sugar representation makes them look like operators). One of these methods is the `==` method we've already encountered. The others, in a similar vein, compare strings based on alphabetical/ASCII order:

```
>> "a" <=> "b"
=> -1
>> "b" > "a"
=> true
>> "a" > "A"
=> true
>> "." > ","
=> true
```

Remember that the spaceship method/operator returns -1 if the right object is greater, 1 if the left object is greater, and 0 if the two objects are equal. In the first case in the sequence above, it returns -1, because the string "b" is greater than the string "a". However, "a" is greater than "A", because the order is done by ASCII value, and the ASCII values for "a" and "A" are 97 and 65, respectively. Similarly, the string "." is greater than "," because the ASCII value for a period is 46 and that for a comma is 44.

At this point, we'll leave strings behind—although you'll continue to see them all over the place—and turn our attention to *symbols*. Symbols, as you'll see, are a close cousin of strings.

10.2 Symbols and their uses

Symbols are instances of the built-in Ruby class `Symbol`. They have a literal constructor: the leading colon. You can always recognize a symbol literal (and distinguish it from a string, a variable name, a method name, or anything else) by this token:

```
:a
:book
:"Here's how to make a symbol with spaces in it."
```

You can also create a symbol programmatically, by calling the `to_sym` method (also known by the synonym `intern`) on a string, as `irb` shows:

```
>> "a".to_sym
=> :a
>> "Converting string to symbol with intern...".intern
=> :"Converting string to symbol with intern..."
```

Note the tell-tale leading colons on the evaluation results returned by `irb`.

You can also easily convert a symbol to a string:

```
>> :a.to_s
=> "a"
```

These examples illustrate how closely related symbols are to strings. Indeed they are related, in that they share responsibility for representing units of text. However, strings and symbols differ in some important ways.

10.2.1 Key differences between symbols and strings

One major difference between symbols and strings is that only one symbol object can exist for any given unit of text. Every time you see the notation for a particular symbol (`:a`), you're seeing *the same symbol object* represented. That differs from the situation with strings. If you see two identical-looking string literals

```
"a"
"a"
```

you're seeing two different string objects (as the string comparison examples in section 10.1.3 demonstrated). With symbols, any two that look the same *are* the same—the same object. You can test this with the `equal?` comparison method, which returns `true` only if both the method's receiver and its argument are the same object:

```
>> :a.equal?(:a)
=> true
>> "a".equal?("a")
=> false
```

It's true that two similar-looking *symbol* literals are the same object but false that two similar-looking *string* literals are.

Another important difference between strings and symbols is that symbols, unlike strings, are *immutable*; you can't add, remove, or change parts of a symbol. The symbol `:abc` is always a different symbol from `:a`, and you can't add `:bc` to `:a` to get `:abc`. Strings are different: You *can* add "bc" to "a", as we've seen.

Symbols have a reputation as “weird strings,” because they're string-like in many ways but also exhibit these differences. Why do they exist? In part because they're an element of the system Ruby uses internally to store and retrieve identifiers. When you assign something to a variable—say, with `x=1`—Ruby creates a corresponding symbol: in this case, the symbol `:x`. The language uses symbols internally but also lets programmers see and use them.

This situation can lead to confusion. Ruby's use of symbols is separate from yours. In your program, the symbol `:x` and the variable `x` aren't connected. The name of the variable is, in an informal sense, a “symbol”—the letter `x`—but it's not a symbol object. If you're interested in how Ruby defines and uses symbol objects internally, you should find out about it. (You might start with the archives of the ruby-talk mailing list, where symbols are discussed frequently; see the appendix.) But you don't need to know the internals to use symbols; and if you do study the internals, you need to keep that knowledge separate from symbol semantics as they apply to your programs.

Think of it this way: Ruby may use the symbol `:x`, and you may use the symbol `:x`, but it's also true that Ruby may use the number 100, and so may you. You don't have to know how Ruby uses 100 internally in order to use 100 in your code. It's worth knowing, however, that symbols are efficient in terms of memory usage and processing time. Strings, on the other hand, come with an entourage of behaviors and capabilities (like being made longer than they started out, having their contents changed, and so on) that makes them more expensive to maintain and process.

You'll often see symbol literals used as arguments to methods and, especially, as hash keys. Hashes that serve as arguments to methods (a common Rails scenario) are a doubly likely candidate for symbol usage.

10.2.2 Rails-style method arguments, revisited

Symbols play a big role in the kind of programming-as-configuration used in Rails, which we looked at in chapter 3. In a case like this

```
class Work < ActiveRecord::Base
  belongs_to :composer
  # etc.
```

`:composer` (the thing works belong to) is represented by a symbol. This symbol is an argument to the `belongs_to` method.

As noted in chapter 3, because you can get a symbol from a string with `to_sym` or `intern`, you can theoretically write the previous method call like this:

```
belongs_to "title".intern
```

This is, of course, not recommended. But it's not as absurd a point to make as it may at first appear. You should recognize `intern` when you come across it. Also, not every Ruby programmer always opts for literal constructs (like `:title`) over programmatic ones (like `"title".intern`). You'll often see people use

```
a = Array.new
```

rather than

```
a = []
```

even though the square brackets (the literal array constructor) achieve the same goal of creating a new, empty array. (You'll learn about arrays in detail in chapter 11.)

In the case of method calls in Rails applications, a consensus exists on the syntax of method calls whose arguments are symbols. You'll probably never see `intern` or `to_sym` used in such a context. Using symbol literals is second nature in Rails development. But you should be aware of exactly what you're seeing and where it fits into the Ruby landscape.

Among other places, you'll see (and have already seen, in part 1) symbols in Rails method calls in constructs like this:

```
<%= link_to "Click here",
  :controller => "book",
  :action     => "show",
  :id         => book.id %>
```

This is an example of a method argument hash: Each of the symbols is a *key*, and each of the values to the right is a *value*. This style of method call is common in Rails application code. (We'll look further at method argument hashes in chapter 11, once we've discussed hashes.)

Symbols are fast, and they have a sleek look that adds to the cleanness of code. Rails usage favors them in many contexts, so it's a good idea (for that reason as well as for the sake of your general Ruby literacy) to become acquainted with them on an equal footing with strings.

Returning to the scalar world at large, let's move on to a realm of objects that are as fundamental to Ruby, and to programming in general, as any: numerical objects.

10.3 Numerical objects

In Ruby, numbers are objects. You can send messages to them, just as you can to any object:

```
n = 98.6
m = n.round
puts m ← ❶
x = 12
if x.zero?
  puts "x is zero"
else
  puts "x is not zero" ← ❷
end

puts "The ASCII character equivalent of 97 is #{97.chr}" ← ❸
```

As you'll see if you run this code, floating-point numbers know how to round themselves ❶ (up or down). Numbers in general know ❷ whether they are zero. And integers can convert themselves to the character that corresponds to their ASCII value ❸.

Numbers are objects; therefore, they have classes—a whole family tree of them.

10.3.1 Numerical classes

Several classes make up the numerical landscape. Figure 10.1 shows a slightly simplified view (mixed-in modules aren't shown) of those classes, illustrating the inheritance relations among them.

The top class in the hierarchy of numerical classes is `Numeric`; all the others descend from it. The first branch in the tree is between floating-point and integral numbers: the `Float` and `Integer` classes. Integers are broken into two classes: `Fixnum` and `Bignum`. (Bignums, as you may surmise, are very large integers. When you use or calculate an integer that's big enough to be a `Bignum`, Ruby handles the conversion automatically for you; you don't have to worry about it.)

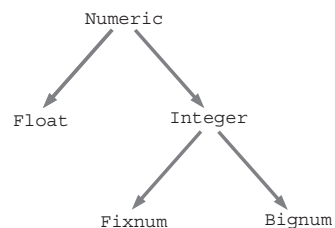


Figure 10.1
Numerical class hierarchy

10.3.2 Performing arithmetic operations

For the most part, numbers in Ruby behave as the rules of arithmetic and arithmetic notation lead you to expect. The examples in table 10.3 should be reassuring in their boringness.

Table 10.3 Common arithmetic expressions and their evaluative results

Expression	Result	Comments
<code>1 + 1</code>	2	Addition
<code>10/5</code>	2	Integer division
<code>10/3</code>	3	Integer division (no automatic floating-point conversion)
<code>10.0/3.0</code>	3.3333333333	Floating-point division
<code>1.2 + 3.4</code>	4.6	Floating-point addition
<code>-12 - -7</code>	-5	Subtraction
<code>10 % 3</code>	1	Modulo (remainder)

Note that when you divide integers, the result will always be an integer. If you want floating-point division, you have to feed Ruby floating-point numbers (even if all you're doing is adding `.0` to the end of an integer).

Ruby also lets you manipulate numbers in non-decimal bases. Hexadecimal integers are indicated by a leading `0x`. Here are some simple-prompt `irb` evaluations of hexadecimal integer expressions:

```
>> 0x12
=> 18
>> 0x12 + 12 ← ❶
=> 30
```

The second `12` in the last expression ❶ is a decimal `12`; the `0x` prefix applies only to the numbers it appears on.

Integers beginning with `0` are interpreted as *octal* (base eight):

```
>> 012
=> 10
>> 012 + 12
=> 22
>> 012 + 0x12
=> 28
```

You can also use the `to_i` method of strings to convert numbers in any base to decimal. To perform such a conversion, you need to supply the base you want to

convert *from* as an argument to `to_i`. The string is then interpreted as an integer in that base, and the whole expression returns the decimal equivalent. You can use any base from 2 to 36, inclusive. Here are some examples:

```
>> "10".to_i(17)
=> 17
>> "12345".to_i(13)
=> 33519
>> "ruby".to_i(35)
=> 1194794
```

Keep in mind that most of the arithmetic operators you see in Ruby are *methods*. They don't look that way because of the operator-like syntactic sugar that Ruby gives them. But they really are methods, and they can be called as methods:

```
>> 1.+(1)
=> 2
>> 12./(3)
=> 4
>> -12.-(-7)
=> -5
```

In practice, no one writes arithmetic operations that way; you'll always see the syntactic sugar equivalents (`1 + 1` and so forth). But seeing examples of the method-call form is a good reminder of the fact that they *are* methods—and also of the fact that you if you define, say, a method called `+` in a class of your own, you can use the operator syntactic sugar. (And if you see arithmetic operators behaving weirdly, it may be that someone has redefined their underlying methods.)

We'll turn now to the next and last category of scalar objects we'll discuss in this chapter: time and date objects.

10.4 Times and dates

Ruby gives you lots of ways to manipulate times and dates—and Rails enhances and extends Ruby's time and date facilities with a variety of new methods. As a Rails developer, you're likely to use those added-on methods more than the raw Ruby ones. Still, you should gain some familiarity with Ruby's date and time libraries, for the sake of being able to use them when you need them as well as for the sake of understanding where the Rails methods come from.

Times and dates are manipulated through three classes: `Time`, `Date`, and `DateTime`. In order to reap their full benefits, you have to pull one or both of the date and time libraries into your program or irb session:

```
require 'date'
require 'time'
```

Rails automatically loads these for you, but in your own non-Rails code you have to load them yourself. (At some point in the future, all the available date- and time-related functionality may be unified into one library and made available to programs by default. But for the moment, you have to do the `require` operations.)

The full range of date and time manipulations available to you is impressive. Want to know what the day we call *April 24, 1705* would have been called in England prior to the calendar reform of 1752? Just load the `date` package, and then ask

```
>> require 'date'
=> true
>> Date.parse("April 24 1705").england.strftime("%B %d %Y")
=> "April 13 1705"
```

(Note that a successful `require` operation returns `true`. As always, `irb` explicitly shows the return value of every expression you type into it.)

Let that example stand in for all the fancy things the various date and/or time classes let you do. On the simpler side, here are some of the potentially useful date and time techniques you may find yourself using:

```
>> require 'date'
=> true
>> d = Date.today
=> #<Date: 4907505/2,0,2299161>
>> puts d
2006-01-17
```

This snippet outputs two different string representations of the `Date` object `d`. The first is the inspect string, which shows that the `Date` object has been successfully created and returned. The second comes from the date's `to_s` method, which is automatically called by `puts`. The `to_s` string, as you can see, is more human-readable.

`Date` objects respond to both a `<<` method and a `>>` method. They advance or rewind the date by a number of months; the number is indicated in the argument. For example

```
puts d << 2
puts d >> 5
```

gives you the date two months before and five months after the date stored in `d`:

```
2005-11-17
2006-06-17
```

You can also create and manipulate `Time` objects. A new `Time` object tells you, when asked, its year, month, day, minute, second, and `usec` (microsecond) values. Here's an `irb` session where a `Time` object is created and queried:

```

>> t = Time.new
=> Tue Jan 17 17:51:04 PST 2006
>> t.year
=> 2006
>> t.month
=> 1
>> t.day
=> 17
>> t.hour
=> 17
>> t.min
=> 51
>> t.sec
=> 4
>> t.usec
=> 377285

```

Time objects also let you display them or store them as strings, based on a UNIX-style format string (basically, a template that specifies how you want the date formatted). The method that does this is `strftime`.

```

>> t.strftime("%m-%d-%Y")
=> "01-17-2006"

```

In the example, the format specifiers used are `%m` (two-digit month), `%d` (two-digit day), and `%Y` (four-digit year). The hyphens between the fields are reproduced in the output as literal hyphens. Some useful format specifiers for `strftime` are shown in table 10.4.

Table 10.4 Common time and date format specifiers

Specifier	Description
<code>%Y</code>	Year (four digits)
<code>%y</code>	Year (last two digits)
<code>%b, %B</code>	Short month, full month
<code>%m</code>	Month (number)
<code>%d</code>	Day of month (left-padded with zeros)
<code>%e</code>	Day of month (left-padded with blanks)
<code>%a, %A</code>	Short day name, full day name
<code>%H, %I</code>	Hour (24-hour clock), hour (12-hour clock)
<code>%M</code>	Minute
<code>%S</code>	Second

Table 10.4 Common time and date format specifiers (continued)

Specifier	Description
<code>%c</code>	Equivalent to <code>"%a %b %d %H:%M:%S %Y"</code>
<code>%x</code>	Equivalent to <code>"%m/%d/%y"</code>

WARNING TIME FORMATS CAN BE LOCALE-SPECIFIC The `%c` and `%x` specifiers, which involve convenience combinations of other specifiers, may differ from one locale to another; for instance, some systems put the day before the month in the `%x` format. This is good, because it means a particular country's style isn't hard-coded into these formats. But you do need to be aware of it, so you don't count on specific behavior that you may not always get. When in doubt, you can use a format string made up of smaller specifiers.

Here are some more examples of time format specifiers in action:

```
>> t.strftime("Today is %x")
=> "Today is 01/17/06"
>> t.strftime("Otherwise known as %d-%b-%y")
=> "Otherwise known as 17-Jan-06"
>> t.strftime("Or even day %e of %B, %Y.")
=> "Or even day 17 of January, 2006."
>> t.strftime("The time is %H:%m.")
=> "The time is 17:01."
```

Many more date and time representations and manipulations are possible in Ruby. A third class beyond `Date` and `Time`, `DateTime`, adds more methods and facilities. It's a rich programming area, although also a vexing one; there's some sentiment among Ruby programmers that it would make sense to unify some or all of the functionality currently spread across three classes into one class, if possible. Some find it incongruous, too, that date and time facilities are split between those that are available by default and those that have to be loaded at runtime. Wherever these and other discussions lead, the functionality is there if and when you wish to explore it.

We've reached the end of our survey of scalar objects in Ruby. Next, in chapter 11, we'll look at collections and container objects.

10.5 Summary

In this chapter, you've seen the basics of the most common and important scalar objects in Ruby: strings, symbols, numerical objects, and time/date objects. Some of these topics involved consolidating points made earlier in the book; others

were completely new in this chapter. At each point, we've examined a selection of important, common methods. We've also looked at how some of the scalar-object classes relate to each other. Strings and symbols both represent text; and though they are different kinds of objects, conversions from one to the other are easy and common. Numbers and strings interact, too. Conversions aren't automatic, as they are (for example) in Perl; but Ruby supplies conversion methods to go from string to numerical object and back, as well as ways to convert strings to integers in as many bases as the 10 digits and 26 letters of the alphabet can accommodate.

Time and date objects have a foot in both the string and numerical camps. You can perform calculations on them, such as adding n months to a given date; and you can also put them through their paces as strings, using techniques like the `Time#strftime` method in conjunction with output format specifiers.

The world of scalar objects in Ruby is rich and dynamic. Moreover, most of what you do with both Ruby and Rails will spring from what you have learned here about scalar objects: direct manipulation of these objects, manipulation of objects that share some of their traits (for example, CGI parameters whose contents are strings), or collections of multiple objects in these categories. Scalar objects aren't everything; but they lie at the root of virtually everything else. The tour we've taken of important scalar classes and methods in this chapter will stand you in good stead as we proceed, next, to look at collections and containers—the two- (and sometimes more) dimensional citizens of Ruby's object world.

RUBY FOR RAILS

David A. Black

The word is out: with Ruby on Rails you can build powerful Web applications easily and quickly! And just like the Rails framework itself, Rails applications are Ruby programs. That means you can't tap into the full power of Rails unless you master the Ruby language.

Ruby for Rails helps Rails developers achieve Ruby mastery. Each chapter deepens your Ruby knowledge and shows you how it connects to Rails. You'll gain confidence working with objects and classes and learn how to leverage Ruby's elegant, expressive syntax for Rails application power. And you'll become a better Rails developer through a deep understanding of the design of Rails itself and how to take advantage of it.


Newcomers to Ruby will find a Rails-oriented Ruby introduction that's easy to read and that includes dynamic programming techniques, an exploration of Ruby objects, classes, and data structures, and many neat examples of Ruby and Rails code in action.

Ruby for Rails: the Ruby guide for Rails developers!

What's Inside

- Classes, modules, and objects
- Collection handling and filtering
- String and regular expression manipulation
- Exploration of the Rails source code
- Ruby dynamics
- Many more programming concepts and techniques!

A Ruby community leader, **David A. Black** is a director of Ruby Central, the parent organization of the annual International Ruby Conference (RubyConf) and the International Rails Conference. David is a Ruby core contributor and the creator and maintainer of the Rails-based Ruby Change Request Archive (RCRchive). He lives and works as a consultant in New Jersey.

 **MANNING** \$44.95 US/\$60.95 Canada

“Closes the gap between Ruby as a language and Rails as a framework. The breadth of knowledge is astounding!”

—Benjamin S. Gorlick
Software Engineer
and Developer

“Code examples are concise *and* useful. I highly recommend it.”

—Mark Eagle, Java Architect
MATRIX Resources, Inc

“A comprehensive tutorial on Ruby and on Rails”

—Bob Hutchison, CTO
Recursive Design Inc.

“I absolutely recommend it!”

—Andrew Oswald
Java Architect
Chariot Solutions



Ask the Author



Ebook edition

www.manning.com/black



9 781932 394696

ISBN 1-932394-69-9