In this chapter, you'll see:

- Objects: names and methods
- Data: strings, arrays, hashes, and regular expressions
- Control: if, while, blocks, iterators, and exceptions
- Building blocks: classes and modules
- YAML and marshaling
- Common idioms that you'll see used in this book

CHAPTER 4

Introduction to Ruby

Many people who are new to Rails are also new to Ruby. If you're familiar with a language such as Java, JavaScript, PHP, Perl, or Python, you'll find Ruby pretty easy to pick up.

This chapter isn't a complete introduction to Ruby. It doesn't cover topics such as precedence rules (as in most other programming languages, `1+2*3==7` in Ruby). It's only meant to explain enough Ruby that the examples in the book make sense.

This chapter draws heavily from material in *Programming Ruby [FH13]*. If you think you need more background on the Ruby language (and at the risk of being grossly self-serving), we'd like to suggest that the best way to learn Ruby and the best reference for Ruby's classes, modules, and libraries is *Programming Ruby [FH13]* (also known as the PickAxe book). Welcome to the Ruby community!

Ruby Is an Object-Oriented Language

Everything you manipulate in Ruby is an object, and the results of those manipulations are themselves objects.

When you write object-oriented code, you're normally looking to model concepts from the real world. Typically, during this modeling process you discover categories of things that need to be represented. In an online store, the concept of a line item could be such a category. In Ruby, you'd define a *class* to represent each of these categories. You then use this class as a kind of factory that generates *objects*—instances of that class. An object is a combination of state (for example, the quantity and the product ID) and methods that use that state (perhaps a method to calculate the line item's total cost). We'll show how to create classes in Classes, on page 54.

You create objects by calling a *constructor*, a special method associated with a class. The standard constructor is called new(). Given a class called LineItem, you could create line item objects as follows:

```
line_item_one = LineItem.new
line_item_one.quantity = 1
line_item_one.sku      = "AUTO_B_00"
```

You invoke methods by sending a message to an object. The message contains the method's name along with any parameters the method may need. When an object receives a message, it looks into its own class for a corresponding method. Let's look at some method calls:

```
"dave".length
line_item_one.quantity()
cart.add_line_item(next_purchase)
submit_tag "Add to Cart"
```

Parentheses are generally optional in method calls. In Rails applications, you'll find that most method calls involved in larger expressions have parentheses, while those that look more like commands or declarations tend not to have them.

Methods have names, as do many other constructs in Ruby. Names in Ruby have special rules—rules that you may not have seen if you come to Ruby from another language.

Ruby Names

Local variables, method parameters, and method names should all start with a lowercase letter or with an underscore: order, line_item, and xr2000 are all valid. Instance variables begin with an at (@) sign—for example, @quantity and @product_id. The Ruby convention is to use underscores to separate words in a multiword method or variable name (so line_item is preferable to lineItem).

Class names, module names, and constants must start with an uppercase letter. By convention they use capitalization, rather than underscores, to distinguish the start of words within the name. Class names look like Object, PurchaseOrder, and LineItem.

Rails uses *symbols* to identify things. In particular, it uses them as keys when naming method parameters and looking things up in hashes. Here's an example:

```
redirect_to :action => "edit", :id => params[:id]
```

As you can see, a symbol looks like a variable name, but it's prefixed with a colon. Examples of symbols include :action, :line_items, and :id. You can think of symbols as string literals magically made into constants. Alternatively, you can consider the colon to mean *thing named*, so :id is the thing named id.

Now that we've used a few methods, let's move on to how they're defined.

Methods

Let's write a method that returns a cheery, personalized greeting. We'll invoke that method a couple of times:

```
def say_goodnight(name)
  result = 'Good night, ' + name
  return result
end

# Time for bed...
puts say_goodnight('Mary-Ellen') # => 'Goodnight, Mary-Ellen'
puts say_goodnight('John-Boy')   # => 'Goodnight, John-Boy'
```

Having defined the method, we call it twice. In both cases, we pass the result to the puts() method, which outputs to the console its argument followed by a newline (moving on to the next line of output).

You don't need a semicolon at the end of a statement as long as you put each statement on a separate line. Ruby comments start with a # character and run to the end of the line. Indentation isn't significant (but two-character indentation is the de facto Ruby standard).

Ruby doesn't use braces to delimit the bodies of compound statements and definitions (such as methods and classes). Instead, you simply finish the body with the end keyword. The return keyword is optional, and if it's not present, the results of the last expression evaluated are returned.

Data Types

While everything in Ruby is an object, some of the data types in Ruby have special syntax support, in particular for defining literal values. In the preceding examples, we used some simple strings and even string concatenation.

Strings

The previous example also showed some Ruby string objects. One way to create a string object is to use *string literals*, which are sequences of characters between single or double quotation marks. The difference between the two forms is the amount of processing Ruby does on the string while constructing the literal. In the single-quoted case, Ruby does very little. With only a few exceptions, what you type into the single-quoted string literal becomes the string's value.

With double-quotes, Ruby does more work. It looks for *substitutions*—sequences that start with a backslash character—and replaces them with a binary value. The most common of these is \n, which is replaced with a newline character. When you write a string containing a newline to the console, the \n forces a line break.

Then, Ruby performs *expression interpolation* in double-quoted strings. In the string, the sequence #{expression} is replaced by the value of expression. We could use this to rewrite our previous method:

```
def say_goodnight(name)
  "Good night, #{name.capitalize}"
end
puts say_goodnight('pa')
```

When Ruby constructs this string object, it looks at the current value of name and substitutes it into the string. Arbitrarily complex expressions are allowed in the #{...} construct. Here we invoked the capitalize() method, defined for all strings, to output our parameter with a leading uppercase letter.

Strings are a fairly primitive data type that contain an ordered collection of bytes or characters. Ruby also provides means for defining collections of arbitrary objects via *arrays* and *hashes*.

Arrays and Hashes

Ruby's arrays and hashes are indexed collections. Both store collections of objects, accessible using a key. With arrays, the key is an integer, whereas hashes support any object as a key. Both arrays and hashes grow as needed to hold new elements. It's more efficient to access array elements, but hashes provide more flexibility. Any particular array or hash can hold objects of differing types; you can have an array containing an integer, a string, and a floating-point number, for example.

You can create and initialize a new array object by using an *array literal*—a set of elements between square brackets. Given an array object, you can access individual elements by supplying an index between square brackets, as the next example shows. Ruby array indices start at zero:

```
a = [ 1, 'cat', 3.14 ]   # array with three elements
a[0]                     # access the first element (1)
a[2] = nil               # set the third element
                         # array now [ 1, 'cat', nil ]
```

You may have noticed that we used the special value nil in this example. In many languages, the concept of *nil* (or *null*) means *no object*. In Ruby, that's not the case; nil is an object, like any other, that happens to represent nothing.

The <<() method is often used with arrays. It appends a single value to its receiver:

```
ages = []
for person in @people
  ages << person.age
end
```

Ruby has a shortcut for creating an array of words:

```
a = [ 'ant', 'bee', 'cat', 'dog', 'elk' ]
# this is the same:
a = %w{ ant bee cat dog elk }
```

Ruby hashes are similar to arrays. A hash literal uses braces rather than square brackets. The literal must supply two objects for every entry: one for the key, the other for the value. For example, you may want to map musical instruments to their orchestral sections:

```
inst_section = {
  :cello     => 'string',
  :clarinet  => 'woodwind',
  :drum      => 'percussion',
  :oboe      => 'woodwind',
  :trumpet   => 'brass',
  :violin    => 'string'
}
```

The thing to the left of the => is the key, and that on the right is the corresponding value. Keys in a particular hash must be unique; if you have two entries for :drum, the last one will *win*. The keys and values in a hash can be arbitrary objects: you can have hashes in which the values are arrays, other hashes, and so on. In Rails, hashes typically use symbols as keys. Many Rails hashes have been subtly modified so that you can use either a string or a symbol interchangeably as a key when inserting and looking up values.

The use of symbols as hash keys is so commonplace that Ruby has a special syntax for it, saving both keystrokes and eyestrain:

```
inst_section = {
  cello:    'string',
  clarinet: 'woodwind',
  drum:     'percussion',
  oboe:     'woodwind',
  trumpet:  'brass',
  violin:   'string'
}
```

Doesn't that look much better?

Feel free to use whichever syntax you like. You can even intermix usages in a single expression. Obviously, you'll need to use the arrow syntax whenever the key is *not* a symbol. One other thing to watch out for—if the *value* is a symbol, you'll need to have at least one space between the colons or else you'll get a syntax error:

```
inst_section = {
  cello:    :string,
  clarinet: :woodwind,
  drum:     :percussion,
  oboe:     :woodwind,
  trumpet:  :brass,
  violin:   :string
}
```

Hashes are indexed using the same square bracket notation as arrays:

```
inst_section[:oboe]     #=> :woodwind
inst_section[:cello]    #=> :string
inst_section[:bassoon]  #=> nil
```

As the preceding example shows, a hash returns nil when indexed by a key it doesn't contain. Normally this is convenient because nil means false when used in conditional expressions.

You can pass hashes as parameters on method calls. Ruby allows you to omit the braces but only if the hash is the last parameter of the call. Rails makes extensive use of this feature. The following code fragment shows a two-element hash being passed to the redirect_to() method. Note that this is the same syntax that Ruby uses for keyword arguments:

```
redirect_to action: 'show', id: product.id
```

One more data type is worth mentioning: the regular expression.

Regular Expressions

A regular expression lets you specify a *pattern* of characters to be matched in a string. In Ruby, you typically create a regular expression by writing /pattern/ or %r{pattern}.

For example, we can use the regular expression /Perl|Python/ to write a pattern that matches a string containing the text *Perl* or the text *Python*.

The forward slashes delimit the pattern, which consists of the two things that we're matching, separated by a vertical bar (|). The bar character means either the thing on the left or the thing on the right—in this case, either *Perl* or *Python*. You can use parentheses within patterns, just as you can in arithmetic expressions, so we could also write this pattern as /P(erl|ython)/. Programs typically use the =~ match operator to test strings against regular expressions:

```
if line =~ /P(erl|ython)/
  puts "There seems to be another scripting language here"
end
```

You can specify *repetition* within patterns. /ab+c/ matches a string containing an *a* followed by one or more *b*s, followed by a *c*. Change the plus to an asterisk, and /ab*c/ creates a regular expression that matches one *a*, zero or more *b*s, and one *c*.

Backward slashes start special sequences; most notably, \d matches any digit, \s matches any whitespace character, and \w matches any alphanumeric (*word*) character, \A matches the start of the string and \z matches the end of the string. A backslash before a wildcard character, for example \., causes the character to be matched as is.

Ruby's regular expressions are a deep and complex subject; this section barely skims the surface. See the PickAxe book for a full discussion.

This book will make only light use of regular expressions.

With that brief introduction to data, let's move on to logic.

Logic

Method calls are statements. Ruby also provides a number of ways to make decisions that affect the repetition and order in which methods are invoked.

Control Structures

Ruby has all the usual control structures, such as if statements and while loops. Java, C, and Perl programmers may well get caught by the lack of braces around the bodies of these statements. Instead, Ruby uses the end keyword to signify the end of a body:

```
if count > 10
  puts "Try again"
elsif tries == 3
  puts "You lose"
else
  puts "Enter a number"
end
```

Similarly, while statements are terminated with end:

```
while weight < 100 and num_pallets <= 30
  pallet = next_pallet()
  weight += pallet.weight
  num_pallets += 1
end
```

Ruby also contains variants of these statements. unless is like if, except that it checks for the condition to *not* be true. Similarly, until is like while, except that the loop continues until the condition evaluates to be true.

Ruby *statement modifiers* are a useful shortcut if the body of an if, unless, while, or until statement is a single expression. Simply write the expression, followed by the modifier keyword and the condition:

```
puts "Danger, Will Robinson" if radiation > 3000
distance = distance * 1.2 while distance < 100
```

Although if statements are fairly common in Ruby applications, newcomers to the Ruby language are often surprised to find that looping constructs are rarely used. *Blocks* and *iterators* often take their place.

Blocks and Iterators

Code blocks are chunks of code between braces or between do…end. A common convention is that people use braces for single-line blocks and do/end for multiline blocks:

```
{ puts "Hello" }       # this is a block

do                     ###
  club.enroll(person)    # and so is this
  person.socialize       #
end                    ###
```

To pass a block to a method, place the block after the parameters (if any) to the method. In other words, put the start of the block at the end of the source line containing the method call. For example, in the following code, the block containing puts "Hi" is associated with the call to the greet() method:

```
greet  { puts "Hi" }
```

If a method call has parameters, they appear before the block:

```
verbose_greet("Dave", "loyal customer")  { puts "Hi" }
```

A method can invoke an associated block one or more times by using the Ruby yield statement. You can think of yield as being something like a method call that calls out to the block associated with the method containing the yield.

You can pass values to the block by giving parameters to yield. Within the block, you list the names of the arguments to receive these parameters between vertical bars (|).

Code blocks appear throughout Ruby applications. Often they're used in conjunction with iterators—methods that return successive elements from some kind of collection, such as an array:

```
animals = %w( ant bee cat dog elk )    # create an array
animals.each {|animal| puts animal }   # iterate over the contents
```

Each integer *N* implements a times() method, which invokes an associated block *N* times:

```
3.times { print "Ho! " }     #=>  Ho! Ho! Ho!
```

The & prefix operator allows a method to capture a passed block as a named parameter:

```
def wrap &b
  print "Santa says: "
  3.times(&b)
  print "\n"
end
wrap { print "Ho! " }
```

Within a block, or a method, control is sequential except when an exception occurs.

Exceptions

Exceptions are objects of the Exception class or its subclasses. The raise method causes an exception to be raised. This interrupts the normal flow through the code. Instead, Ruby searches back through the call stack for code that says it can handle this exception.

Both methods and blocks of code wrapped between begin and end keywords intercept certain classes of exceptions using rescue clauses:

```
begin
  content = load_blog_data(file_name)
rescue BlogDataNotFound
  STDERR.puts "File #{file_name} not found"
rescue BlogDataFormatError
  STDERR.puts "Invalid blog data in #{file_name}"
rescue Exception => exc
  STDERR.puts "General error loading #{file_name}: #{exc.message}"
end
```

rescue clauses can be directly placed on the outermost level of a method definition without needing to enclose the contents in a begin/end block.

That concludes our brief introduction to control flow. At this point you have the basic building blocks for creating larger structures.

Organizing Structures

Ruby has two basic concepts for organizing methods: classes and modules. We cover each in turn.

Classes

Here's a Ruby class definition:

```
Line 1  class Order < ApplicationRecord
     -    has_many :line_items
     -    def self.find_all_unpaid
     -      self.where('paid = 0')
     5    end
     -    def total
     -      sum = 0
     -      line_items.each {|li| sum += li.total}
     -      sum
    10    end
     -  end
```

Class definitions start with the class keyword and are followed by the class name (which must start with an uppercase letter). This Order class is defined to be a subclass of the ApplicationRecord class.

Rails makes heavy use of class-level declarations. Here, has_many is a method that's defined by Active Record. It's called as the Order class is being defined. Normally these kinds of methods make assertions about the class, so in this book we call them *declarations*.

Within a class body, you can define class methods and instance methods. Prefixing a method name with self. (as we do on line 3) makes it a class method; it can be called on the class generally. In this case, we can make the following call anywhere in our application:

```
to_collect = Order.find_all_unpaid
```

Objects of a class hold their state in *instance variables*. These variables, whose names all start with @, are available to all the instance methods of a class. Each object gets its own set of instance variables.

Instance variables aren't directly accessible outside the class. To make them available, write methods that return their values:

```ruby
class Greeter
  def initialize(name)
    @name = name
  end

  def name
    @name
  end

  def name=(new_name)
    @name = new_name
  end
end

g = Greeter.new("Barney")
g.name   # => Barney
g.name = "Betty"
g.name   # => Betty
```

Ruby provides convenience methods that write these accessor methods for you (which is great news for folks tired of writing all those getters and setters):

```ruby
class Greeter
  attr_accessor  :name       # create reader and writer methods
  attr_reader    :greeting   # create reader only
  attr_writer    :age        # create writer only
end
```

A class's instance methods are public by default; anyone can call them. You'll probably want to override this for methods that are intended to be used only by other instance methods:

```ruby
class MyClass
  def m1      # this method is public
  end
  protected
  def m2      # this method is protected
  end
  private
  def m3      # this method is private
  end
end
```

The `private` directive is the strictest; private methods can be called only from within the same instance. Protected methods can be called both in the same instance and by other instances of the same class and its subclasses.

Classes aren't the only organizing structure in Ruby. The other organizing structure is a *module*.

Modules

Modules are similar to classes in that they hold a collection of methods, constants, and other module and class definitions. Unlike with classes, you can't create objects based on modules.

Modules serve two purposes. First, they act as a namespace, letting you define methods whose names won't clash with those defined elsewhere. Second, they allow you to share functionality among classes. If a class *mixes in* a module, that module's methods become available as if they'd been defined in the class. Multiple classes can mix in the same module, sharing the module's functionality without using inheritance. You can also mix multiple modules into a single class.

Helper methods are an example of where Rails uses modules. Rails automatically mixes these helper modules into the appropriate view templates. For example, if you wanted to write a helper method that's callable from views invoked by the store controller, you could define the following module in the store_helper.rb file in the app/helpers directory:

```ruby
module StoreHelper
  def capitalize_words(string)
    string.split(' ').map {|word| word.capitalize}.join(' ')
  end
end
```

One module that's part of the standard library of Ruby deserves special mention, given its usage in Rails: YAML.

YAML

YAML[1] is a recursive acronym that stands for YAML Ain't Markup Language. In the context of Rails, YAML is used as a convenient way to define the configuration of things such as databases, test data, and translations. Here's an example:

```yaml
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

In YAML, indentation is important, so this defines development as having a set of four key-value pairs, separated by colons. While YAML is one way to represent data, particularly when interacting with humans, Ruby provides a more general way for representing data for use by applications.

1. http://www.yaml.org/

Marshaling Objects

Ruby can take an object and convert it into a stream of bytes that can be stored outside the application. This process is called *marshaling*. This saved object can later be read by another instance of the application (or by a totally separate application), and a copy of the originally saved object can be reconstituted.

Two potential issues arise when you use marshaling. First, some objects can't be dumped. If the objects to be dumped include bindings, procedure or method objects, instances of the IO class, or singleton objects—or if you try to dump anonymous classes or modules—a TypeError will be raised.

Second, when you load a marshaled object, Ruby needs to know the definition of the class of that object (and of all the objects it contains).

Rails uses marshaling to store session data. If you rely on Rails to dynamically load classes, it's possible that a particular class may not have been defined at the point it reconstitutes session data. For that reason, use the model declaration in your controller to list all models that are marshaled. This preemptively loads the necessary classes to make marshaling work.

Now that you have the Ruby basics down, let's give what we learned a whirl with a slightly larger, annotated example that pulls together a number of concepts. We'll follow that with a walk-through of special features that will help you with your Rails coding.

Pulling It All Together

Let's look at an example of how Rails applies a number of Ruby features together to make the code you need to maintain more declarative. You'll see this example again in Generating the Scaffold, on page 70. For now, we'll focus on the Ruby-language aspects of the example:

```ruby
class CreateProducts < ActiveRecord::Migration[7.0]
  def change
    create_table :products do |t|
      t.string :title
      t.text :description
      t.string :image_url
➤     t.decimal :price, precision: 8, scale: 2

      t.timestamps
    end
  end
end
```

Even if you didn't know any Ruby, you'd probably be able to decipher that this code creates a table named products. The fields defined when this table is created include title, description, image_url, and price, as well as a few timestamps (we'll describe these in Chapter 23, Migrations, on page 395).

Now let's look at the same example from a Ruby perspective. We define a class named CreateProducts, which inherits from the versioned[2] Migration class from the ActiveRecord module, specifying that compatibility with Rails 7 is desired. We define one method, named change(). This method calls the create_table() method (defined in ActiveRecord::Migration), passing it the name of the table in the form of a symbol.

The call to create_table() also passes a block that is to be evaluated before the table is created. This block, when called, is passed an object named t, which is used to accumulate a list of fields. Rails defines a number of methods on this object—methods named after common data types. These methods, when called, simply add a field definition to the ever-accumulating set of names.

The definition of decimal also accepts a number of optional parameters, expressed as a hash.

To someone new to Ruby, this is a lot of heavy machinery thrown at solving such a simple problem. To someone familiar with Ruby, none of this machinery is particularly heavy. In any case, Rails makes extensive use of the facilities provided by Ruby to make defining operations (for example, migration tasks) as simple and as declarative as possible. Even small features of the language, such as optional parentheses and braces, contribute to the overall readability and ease of authoring.

Finally, a number of small features—or, rather, idiomatic combinations of features—are often not immediately obvious to people new to the Ruby language. We close this chapter with them.

Ruby Idioms

A number of individual Ruby features can be combined in interesting ways. We use these common Ruby idioms in this book:

2. http://blog.bigbinary.com/2016/03/01/migrations-are-versioned-in-rails-5.html

Methods such as empty! and empty?

Ruby method names can end with an exclamation mark (a *bang method*) or a question mark (a *predicate method*). Bang methods normally do something destructive to the receiver. Predicate methods return true or false, depending on some condition.

a || b

The expression a || b evaluates a. If it isn't false or nil, then evaluation stops and the expression returns a. Otherwise, the statement returns b. This is a common way of returning a default value if the first value hasn't been set.

a ||= b

The assignment statement supports a set of shortcuts: a op= b is the same as a = a op b. This works for most operators:

```
count += 1          # same as count = count + 1
price *= discount   #         price = price * discount
count ||= 0         #         count = count || 0
```

So, count ||= 0 gives count the value 0 if count is nil or false.

obj = self.new

Sometimes a class method needs to create an instance of that class:

```
class Person < ApplicationRecord
  def self.for_dave
    Person.new(name: 'Dave')
  end
end
```

This works fine, returning a new Person object. But later, someone might subclass our class:

```
class Employee < Person
  # ..
end

dave = Employee.for_dave  # returns a Person
```

The for_dave() method was hardwired to return a Person object, so that's what's returned by Employee.for_dave. Using self.new instead returns a new object of the receiver's class, Employee.

lambda

The lambda operator converts a block into an object of type Proc. An alternative syntax, introduced in Ruby 1.9, is ->. As a matter of style, the Rails team prefers the latter syntax. You can see example usages of this operator in Scopes, on page 320.

require File.expand_path('../../config/environment', __FILE__)

Ruby's require method loads an external source file into our application. This is used to include library code and classes that our application relies on. In normal use, Ruby finds these files by searching in a list of directories, the LOAD_PATH.

Sometimes we need to be specific about which file to include. We can do that by giving require a full filesystem path. The problem is, we don't know what that path will be—our users could install our code anywhere.

Wherever our application ends up getting installed, the relative path between the file doing the requiring and the target file will be the same. Knowing this, we can construct the absolute path to the target by using the File.expand_path() method, passing in the relative path to the target file, and passing the absolute path to the file doing the requiring (available in the special __FILE__ variable).

In addition, the web has many good resources that show Ruby idioms and Ruby gotchas. Here are a few of them:

- http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/
- http://en.wikipedia.org/wiki/Ruby_programming_language
- https://www.zenspider.com/ruby/quickref.html

By this point, you have a firm foundation to build on. You've installed Rails, verified that you have things working with a simple application, read a brief description of what Rails is, and reviewed (or for some of you, learned for the first time) the basics of the Ruby language. Now it's time to put this knowledge in place to build a larger application.