The Pragmatic Programmers

Programming Ruby 3.3

The Pragmatic Programmers' Guide

Noel Rappin
with Dave Thomas
Edited by Katharine Dvorak

The Facets of Ruby Series

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.prag-prog.com.

Many books on programming languages look about the same. They start with chapters on basic types: integers, strings, and so on. Then they look at expressions like `2 + 3` before moving on to `if` and `while` statements and loops. Then, perhaps around Chapter 7 or 8, they'll start mentioning classes. We find that somewhat tedious.

Instead, when we designed this book, we had a grand plan. We wanted to document the language from the top down, starting with classes and objects and ending with the nitty-gritty syntax details. It seemed like a good idea at the time. After all, most everything in Ruby is an object, so it made sense to talk about objects first.

Or so we thought.

Unfortunately, it turns out to be difficult to describe a language top-down. If you haven't covered strings, `if` statements, assignments, and other details, it's difficult to write examples of classes. Throughout our top-down description, we kept coming across low-level details we needed to cover so that the example code would make sense.

So we came up with another grand plan (they don't call us pragmatic for nothing). We'd still describe Ruby starting at the top. But before we did that, we'd add a short chapter that described all the common language features used in the examples along with the special vocabulary used in Ruby—a mini-tutorial to bootstrap us into the rest of the book. And that mini-tutorial is this chapter.

Ruby Is an Object-Oriented Language

Let's say it again. Ruby is an object-oriented language. In programming terms, an *object* is a thing that combines data with the logic that manipulates that data, and a language is "object-oriented" if it provides language constructs that make it easy to create objects. Typically, object-oriented languages allow their objects to define what their data is, define their functionality, and provide a common syntax to allow other objects to access that functionality.

Many languages claim to be object-oriented, and those languages often have a different interpretation of what object-oriented means and a different terminology for the concepts they employ. Unlike other object-oriented languages such as Java, JavaScript, and Python, all Ruby types are objects, and there are no non-object basic types that behave differently.

Before we get too far into the details, let's briefly look at the terms and notations that we'll be using to talk about Ruby.

When you write object-oriented programs, you're looking to model concepts from the outside world or from your logical domain. During this modeling process, you'll discover categories of related data and behavior that need to be represented in code. In a system representing a jukebox, the concept of a "song" could be such a category. A song might combine state (for example, the name of the song) and methods that use that state (perhaps a method to play the song). In Ruby, you'd define a *class* called `Song` to represent the general case of what songs do.

Once you have these classes, you'll typically want to create a number of separate *instances* of each. For the jukebox system containing a class called `Song`, you'd have separate instances for popular hits with different names such as "Ruby Tuesday," "Enveloped in Python," "String of Pearls," "Small Talk," and so on. Each of these instances has its own state but shares the common behavior of the class. The word *object* is often used interchangeably with *instance*.

In Ruby, instances are created by calling a *constructor*, which is a special method associated with a class. The standard constructor is called `new`. As we'll see later in Chapter 3, Classes, Objects, and Variables, on page ?, the `new` method is defined for you by Ruby, and you don't need to define it on your own. You might create instances like this:

```
song1 = Song.new("Ruby Tuesday")
song2 = Song.new("Enveloped in Python")
# and so on
```

These instances are both derived from the same class, but they each have unique characteristics. Every object has a unique *object identifier* (abbreviated as *object id*), accessible via the property `object_id`. In this example, if you were to check `song1.object_id` and `song2.object_id`, you'd find they have different values.

For each instance, you can define *instance variables*, variables with values that are unique to that instance. These instance variables hold an object's state. Each of our songs, for example, will have an instance variable that holds that song's title.

Within each class, you can define *instance methods*. Each method is a chunk of functionality that may be called in the context of the class and usually from outside the class, although you can set constraints on what methods can be used externally. These instance methods have access to the object's instance variables and hence to the object's state. A `Song` class, for example, might define an instance method called `play`. If a variable referenced a particular `Song` instance, you'd be able to call that instance's `play` method and play that song.

Syntactically, a method is invoked using dot syntax, here are some examples:

intro/puts_examples.rb

```
"gin joint".length  # => 9
"Rick".index("c")   # => 2
42.even?            # => true
sam.play(song)      # => "duh dum, da dum de dum ..."
```

Each line shows a method being called. The item before the dot is called the *receiver* of the method, and what comes after the period is the name of the method to be invoked. The first example asks the string `"gin joint"` for its length. The second asks a different string to find the index within it of the letter `c`. The third line asks the number 42 if it's even (the question mark is part of the method name `even?`). Finally, we ask an object called `sam` to play us a song (assuming there's an existing variable called `sam` that references an appropriate object which we've defined elsewhere).

When we talk about methods being sent, we often say that we send a *message* to the object. The message contains the method's name along with any arguments the method may expect. The object responds to the message by invoking the method with that name. This idea of expressing method calls in the form of messages to objects comes from the programming language Smalltalk. When an object receives a message, it looks into its own class for a corresponding method. If found, that method is executed. If the method *isn't* found, Ruby goes off to look for it—we'll get to that in Method Lookup, on page ?.

It's worth noting here a major difference between Ruby and other object-oriented languages. In Java, for example, you'd find the absolute value of some number by calling a separate function and passing in that number. You could write this:

```
num = Math.abs(num);     // Java code
```

In Ruby, the ability to determine an absolute value is built into the numbers class which takes care of the details internally. You send the message abs to a number object and let it do the work:

```
num = -1234        # => -1234
positive = num.abs # => 1234
```

The same applies to all Ruby objects. In Python, you'd write len(name), but in Ruby, it would be name.length, and so on. This consistency of behavior is what we mean when we say that Ruby is a pure object-oriented language with no basic types.

Some Basic Ruby

Not everybody likes to read heaps of boring syntax rules when they're picking up a new language, so we're going to cheat. In this section, we'll hit the highlights—the stuff you'll *need* to know if you're going to write Ruby programs. Later, in Part IV, Ruby Language Reference, on page ?, we'll go into all the gory details.

Let's start with a short Ruby program. We'll write a method that returns a personalized greeting. We'll then invoke that method a couple of times:

intro/hello1.rb

```
def say_hello_goodbye(name)
  result = "I don't know why you say goodbye, " + name + ", I say hello"
  return result
end

# call the method
puts say_hello_goodbye("John")
puts say_hello_goodbye("Paul")
```

produces:

```
I don't know why you say goodbye, John, I say hello
I don't know why you say goodbye, Paul, I say hello
```

As the example shows, Ruby syntax is uncluttered. You don't need semicolons at the ends of statements 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. Code layout is up to you; indentation isn't significant. That said, two-character indentation—spaces, not tabs—is the overwhelming choice of the Ruby community.

Methods are defined with the keyword def, followed by the method name—in this case, the name is say_hello_goodbye—and then the method's parameters between parentheses. (In fact, the parentheses are optional, but we recommend you use them.) Ruby doesn't use braces to delimit the bodies of compound statements and definitions. Instead, you finish the body with the keyword end. Our method's body is pretty short. The first line concatenates the literal string "I don't know why you say goodbye, " and the parameter name and the literal string ", I say hello" and assigns the result to the local variable result. The next line returns that result to the caller. Note that we didn't have to declare the variable result; it sprang into existence when we assigned a value to it.

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

```
I don't know why you say goodbye, John, I say hello
I don't know why you say goodbye, Paul, I say hello
```

The line puts say_hello_goodbye("John") actually contains two method calls: one to the method say_hello_goodbye with the argument "John" and the other to the method puts whose argument is the result of the call to say_hello_goodbye . Why does one call have its arguments in parentheses while the other doesn't? In this case, it's purely a matter of taste—the puts method is available to all objects and is often written without parentheses around its argument. Ruby doesn't require parentheses unless they are directly needed for the interpreter to parse the statement the way you want. The following lines are equivalent:

```
puts say_hello_goodbye("John")
puts(say_hello_goodbye("John"))
```

Life isn't always simple, and precedence rules can make it difficult to know which argument goes with which method invocation. So, we recommend using parentheses in all but the simplest cases. You'll see that Ruby programs often omit the parentheses when the method doesn't have an explicit receiver and only has one argument.

This example also shows Ruby string objects. Ruby has many ways to create a string object, but the most common is to use *string literals*, which are sequences of characters between single or double quotation marks. The two forms differ in the amount of processing Ruby does on the string while constructing the literal. In the single-quoted case, Ruby does very little. With a few exceptions, what you enter in the string literal becomes the string's value.

In the double-quoted case, Ruby does more work. First, it looks for substitution sequences that start with a backslash character and replaces them with some binary value. The most common of these substitutions is \n, which is replaced with a newline character. When a string containing a newline is output, that newline becomes a line break:

```
puts "Hello and goodbye to you,\nGeorge"
```

produces:

```
Hello and goodbye to you,
George
```

The second thing that Ruby does with double-quoted strings is expression interpolation. Within the string, the sequence #{EXPRESSION} is replaced by the value of EXPRESSION. We could use this to rewrite our previous method:

```
def say_hello_goodbye(name)
  result = "I don't know why you say goodbye, #{name}, I say hello"
  return result
end
puts say_hello_goodbye("Ringo")
```

produces:

```
I don't know why you say goodbye, Ringo, I say hello
```

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. In the following example, we invoke the capitalize method, defined for all strings, to output our parameter with a leading uppercase letter:

```
def say_hello_goodbye(name)
  result = "I don't know why you say goodbye,  #{name.capitalize}, I say hello"
```

```
  return result
end
puts say_hello_goodbye("john")
```

produces:

```
I don't know why you say goodbye,  John, I say hello
```

For more information on strings and the other Ruby standard types, see Chapter 7, Basic Types: Numbers, Strings, and Ranges, on page ?.

We could simplify our `say_hello_goodbye` method some more. In the absence of an explicit `return` statement, the value returned by a Ruby method is the value of the last expression evaluated, so we can get rid of the temporary variable and the `return` statement altogether.

```
def say_hello_goodbye(name)
  "I don't know why you say goodbye, #{name}, I say hello"
end
puts say_hello_goodbye("Paul")
```

produces:

```
I don't know why you say goodbye, Paul, I say hello
```

This version is considered more *idiomatic*, by which we mean that it's more in line with how expert Ruby programmers have chosen to write Ruby programs. Idiomatic Ruby tends to lean into Ruby's shortcuts and specific syntax. A good clearinghouse for the guidelines for idiomatic Ruby style can be found in the documentation for the Standard gem at https://github.com/testdouble/standard, which has been used for the code in this book (except where we deliberately break its rules to make a point).

We promised that this section would be brief. We have one more topic to cover: Ruby names. For brevity, we'll be using some terms (such as *class variable*) that we aren't going to define here. But, by talking about the rules now, you'll be ahead of the game when we actually come to discuss class variables and the like later.

Ruby uses a convention that may seem strange at first: the first characters of a name indicate how broadly the variable is visible. Local variables, method parameters, and method names should all start with a lowercase letter or an underscore (Ruby itself has a couple of methods that start with a capital letter, but in general this isn't something to do in your own code).

Global variables are prefixed with a dollar sign, `$`, and instance variables begin with an "at" sign, `@`. Class variables start with two "at" signs, `@@`. Although we talk about global and class variables here for completeness, you'll find they are rarely used in Ruby programs. There's a lot of evidence that global variables make programs harder to maintain. Class variables aren't as dangerous as global variables, but they are still tricky to use safely—people tend not to use them much because they often use easier ways to get similar functionality. Finally, class names, module names, and other constants must start with an uppercase letter. Samples of different names are given in Table 1, Example variable, class, and constant names, on page 8.

Following this initial character, a name can contain any combination of letters, digits, and underscores, with the exception that the character following an `@` sign may not be a digit. But, by convention, multiword instance variables are written with underscores between the words, like `first_name` or `zip_code`, and multiword class names are written in MixedCase (sometimes called CamelCase) with each word capitalized, like `FirstName` or `ZipCode`. Constant

Local Variable:	name fish_and_chips x_axis thx1138 _x _26
Instance Variable:	@name @point_1 @X @_ @plan9
Class Variable:	@@total @@symtab @@N @@x_pos @@SINGLE
Global Variable:	$debug $CUSTOMER $_ $plan9 $Global
Class Name:	String ActiveRecord MyClass
Constant Name:	FEET_PER_MILE DEBUG

Table 1—Example variable, class, and constant names

names are written in all caps, with words separated by underscores, like `FIRST_NAME` or `ZIP_CODE`. Method names may end with the characters ?, !, and =.