Extracted from:

Learn to Program, Third Edition

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Updated for Ruby 3

Learn to Program

Third Edition

Chris Pine

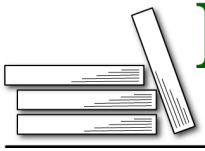
edited by Tammy Coron

The Facets of Ruby Series

Learn to Program, Third Edition

Chris Pine

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Tammy Coron
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-817-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

CHAPTER 10

More Classes and Methods

So far, you've seen several kinds (or *classes*) of objects: strings, integers, floats, arrays, a few special objects (true, false, and nil), and so on. In Ruby, these class names are always capitalized: String, Integer, Float, Array, File, and Dir. (Do you remember back on page ? when you asked the File class to open a file for you, and it handed back an actual file, which you named, in a fit of rabid creativity, f? Good times. And while you never ended up needing an actual directory object from Dir, you could have gotten one if you'd asked nicely.)

You used File.open to get a file back, but that's actually a slightly unusual way to get an object from a class. In general, you'll use the new method:

```
Line 1 alpha = Array.new  + [12345]  # Create empty array.
   2 beta  = String.new + "hello"  # Create empty string.
   3 karma = Time.new              # Current date and time.
   4
   5 puts "alpha = #{alpha}"
   6 puts "beta  = #{beta}"
   7 puts "karma = #{karma}"
```

```
alpha = [12345]
beta  = hello
karma = 2020-10-10 11:56:25 -0700
```

Because you can make arrays with array *literals* using [...], and you can make strings with string literals using "...", you rarely create these using new. Also, numbers are special exceptions: you can't create an integer with Integer.new. (Which number would it create, you know?) You can make one only using an integer literal (that is, by typing it out as you've been doing). But aside from a few exceptions, you normally create Ruby objects with new.

One way you could think of this is that a class is kind of like a cookie cutter: every time you call new on that class, you get a new cookie that's defined by

the shape of that cookie cutter. You'd never try to take a bite out of a cookie cutter, because a cookie cutter isn't a cookie. Similarly, a class is a *different kind of thing* from the objects it creates. (The String class isn't a string. It's a class, a sort of programmatic cookie cutter.)

Let's take a tour of some other frequently-used classes, and learn what you can do with the objects they create.

The Time Class

What's the story with this Time class? Time objects represent (you guessed it) moments in time. You can add (or subtract) numbers to (or from) times to get new times. So, adding 1.5 to a time object makes a new time one-and-a-half seconds later:

```
Line 1 time = Time.new  # The moment this code was run.
   2 soon = time + 60 # One minute later.
   3
   4 puts time
   5 puts soon
```

```
2020-10-10 11:57:23 -0700
2020-10-10 11:58:23 -0700
```

You can also pass values into Time.new to construct a specific time:

```
Line 1 puts Time.new(2000, 1, 1)         # Y2K.
   2 puts Time.new(1976, 8, 3, 13, 31) # When I was born.
```

```
2000-01-01 00:00:00 -0800
1976-08-03 13:31:00 -0700
```

You'll notice the -0700 and -0800 in these times. That's to account for the difference between the local time and Coordinated Universal Time, or UTC. The difference between local time and UTC could be due to time zone or daylight saving time or who knows what else. So, you can see that I was born in daylight saving time, while it was *not* daylight saving time when Y2K struck. The more values you pass in to new, the more specific your time becomes.

On the other hand, if you want to avoid time zones and daylight saving time altogether and only use UTC, there's always Time.utc:

```
Line 1 puts Time.utc(1955, 11, 5) # A red-letter day.
```

```
1955-11-05 00:00:00 UTC
```

You can compare times using the comparison methods (an earlier time is *less than* a later time), and if you subtract one time from another, you'll get the number of seconds between them. Play around with it.

If you happen to be using an older version of Ruby, there's this problem with the Time class. It thinks the world began at *epoch*: the stroke of midnight, January 1, 1970, UTC. I don't know of any satisfying way of explaining this, but here goes: at some point, probably before I was even born, some people (Unix folks, I believe) decided that a good way to represent time on computers was to count the number of seconds since the very beginning of the 70s. So, time "zero" stood for the birth of that great decade, and they called it *epoch*.

This was all long before Ruby. In those ancient days (and programming in those ancient languages), you often had to worry about your numbers getting too large. In general, a number would either be from 0 to around 4 billion or be from -2 billion to +2 billion, depending on how they chose to store it.

For whatever reasons (compatibility, tradition, cruelty…whatever), older versions of Ruby decided to go with these conventions. So (and this is the important point), you *couldn't have times more than 2 billion seconds away from epoch*. This restriction wasn't too painful, though, because this span is from sometime in December 1901 to sometime in January 2038.

In modern Ruby (versions 1.9.2 and up), you don't need to worry about any of this. Time can handle any time that has any meaning, from before the Big Bang to after the heat death of the universe (or however this grand tale ends).

Here are some exercises for you to get more familiar with Time.

A Few Things to Try

- *One billion seconds!* Find out the exact second you were born, or as close as you can get. Figure out when you'll turn (or perhaps when you did turn) one billion seconds old. Then go mark your calendar!
- *Happy birthday!* Ask what year a person was born, then the month, and then the day. Figure out how old they are, and give them a ☺ (smiley emoji) for each birthday they've had.

The Hash Class

Another useful class is the Hash class. Hashes are a lot like arrays: they have a bunch of slots that can point to various objects. But in an array, the slots are lined up in a row, and each one is numbered (starting from zero). In a hash, the slots aren't in a row (they're sort of jumbled together), and you can use *any* object to refer to a slot, not only a number. It's good to use hashes when you have a bunch of things you want to keep track of, but

they don't fit into an ordered list. For example, you could store European capitals like this:

```
caps_array = [] # array literal, same as Array.new
caps_hash  = {} # hash literal,  same as Hash.new

caps_array[0]       = "Oslo"
caps_array[1]       = "Paris"
caps_array[2]       = "Madrid"
caps_array[3]       = "Rome"
caps_hash["Norway"] = "Oslo"
caps_hash["France"] = "Paris"
caps_hash["Spain" ] = "Madrid"
caps_hash["Italy" ] = "Rome"

caps_array.each do |city|
  puts city
end

caps_hash.each do |country, city|
  puts "#{country}: #{city}"
end
```

```
Oslo
Paris
Madrid
Rome
Norway: Oslo
France: Paris
Spain: Madrid
Italy: Rome
```

If you use an array, you have to remember that slot `0` is for `"Oslo"`, slot `1` is for `"Paris"`, and so on. But if you use a hash, it's easy. Slot `"Norway"` holds the name `"Oslo"`, of course. There's nothing to remember. You might have noticed that when you used `each`, the objects in the hash came out in the same order you put them in. Older versions of Ruby (1.8 and earlier) didn't work this way, so if you have an older version installed, be aware of this (or better yet, simply upgrade).

That last example included an empty hash literal, and populated it with European capitals one at a time. But as with arrays, you can populate a hash using only the literal:

```
greetings = ["hello", "howdy", "hi"]               # array literal
smoothies = {"mango" => "yum", "garlic" => "yuck"} # hash literal

puts greetings
puts smoothies
puts smoothies["mango"]
```

```
hello
howdy
hi
{"mango"=>"yum", "garlic"=>"yuck"}
yum
```

Though people usually use strings to name the slots in a hash, you could use any kind of object, even arrays and other hashes:

```
weird_hash = Hash.new

weird_hash[12] = "monkeys"
weird_hash[[]] = "the void"
weird_hash[Time.new] = "no time like the present"
```

Hashes and arrays are good for different things. It's up to you to decide which one is best for a particular problem. Are you storing things that are sequential or ordered? Then an array makes most sense. Is it more like a collection of different things? Then consider using a hash. In general, arrays are better for collections of *the same kind of thing*, while hashes are better for collections of *different kinds of things*:

```
myself = {"name" => "Chris", "pairs_of_shoes" => 17}
imelda = {"name" => "Imelda", "pairs_of_shoes" => 3400}

people = [myself, imelda]
```

Names and numbers of pairs of shoes aren't the same kind of thing at all, so we store those together in a hash. But those two hashes are both representing people (at least a little bit about them), so we use an array to group the people-hashes together.

I probably use hashes at least as often as arrays. They're wonderful.

The Range Class

`Range` is another great class. Ranges represent intervals of numbers. Usually. (You can also make ranges of strings, times, or pretty much anything else you can place in an order—where you can say things like `this < that` and such. In practice, I never use ranges for anything but integers.)

Let's make some range literals and see what they can do:

```
# This is a range literal.
numbers = 1..5

# Convert range to array.
puts([1, 2, 3, 4, 5] == numbers.to_a)

# Iterate over a range of strings:
```

```
("a".."z").each do |letter|
  print letter
end
puts

god_bless_the_90s = 1990..1999
puts god_bless_the_90s.min
puts god_bless_the_90s.max
puts(god_bless_the_90s.include?(1999  ))
puts(god_bless_the_90s.include?(2000  ))
puts(god_bless_the_90s.include?(1994.5))
```

```
true
abcdefghijklmnopqrstuvwxyz
1990
1999
true
false
true
```

Do you really need ranges? No, not really. You could write programs without them. It's the same with hashes and times, I suppose. But it's all about style, intention, and capturing snapshots of your brain right there in your code.

And this is all just the tip of the iceberg. Each of these classes has way more methods than covered here, and this isn't even a tenth of the classes that come with Ruby. But you don't *need* most of them. They are simply time-savers. You can pick them up gradually as you go. That's how most of us do it.

Now that you've learned about `Time`, `Hash`, and `Range`, let's revisit `String`. I'd feel like I was doing you a disservice if we didn't look a little more at what you can do with strings. Plus, if we do, I can give you more interesting exercises. Mind you, we're still not going to cover even half of the string methods, but you've *got* to see a little more.