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

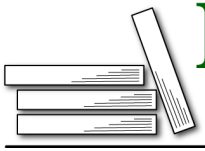
edited by Tammy Coron

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 6

Flow Control

Up to this point, the program examples in this book have been somewhat static and predictable. In other words, you get the same experience each time the program runs. Sure, when the program asks for my name, instead of responding with "Chris," I could answer with "Fromaggio the Odorous," but that's hardly a new experience, right? And just barely interactive.

After this chapter, though, you'll be able to write truly interactive programs. To do this, you need to learn four things: comparison methods, branching, looping, and logic operators. Let's take them in that order.

In the past, you made programs that *said* different things depending on your keyboard input, but after this chapter they'll actually *do* different things. But how will you determine when to do one thing instead of another? You'll use comparison methods.

Comparison Methods

Comparison methods, like > and <, allow you to ask Ruby the question, "Is this greater than (or less than) that?" In the next section you'll learn how we use the answer to these questions to make your program do different things. But before we worry about the answers, let's see how to ask the questions:

```
Line 1  puts 1 > 2
      2  puts 1 < 2

    ❮  false
       true
```

This looks fairly straightforward, yes?

Likewise, you can find out whether an object is greater than or equal to (or less than or equal to) another with the methods >= and <=:

```
Line 1  puts 5 >= 5
   2  puts 5 <= 4

   ❮  true
      false
```

And finally, you can see whether two objects are equal using == (which asks Ruby, “Are these equal?”) and != (which asks, “Are these *not* equal?”):

```
Line 1  puts 1 == 1
   2  puts 2 != 1

   ❮  true
      true
```

It’s important not to confuse = with ==. A single = is for telling a variable to point at an object (assignment), and == is for asking the question “Are these two objects equal?” (Having said that, we’ve all made this mistake before.)

Hey, guess what? You can also compare strings. When strings get compared, Ruby uses their *lexicographical ordering*, which simply means the order they appear in a dictionary. For example, `cat` comes before `dog` in the dictionary, so you have this:

```
Line 1  puts "cat" < "dog"

   ❮  true
```

There’s a catch, though. Computers usually order capital letters before lowercase letters. (That’s how they store the letters in fonts: all of the capital letters first and then the lowercase ones.) This means the computer will think `"Watermelon Steven"` comes before `"a giant woman"`. So if you want to figure out which word would come first in a real dictionary, make sure to use `downcase` (or `upcase` or `capitalize`) on both words before you try to compare them:

```
Line 1  puts "a giant woman"           < "Watermelon Steven"
   2  puts "a giant woman".downcase < "Watermelon Steven".downcase

   ❮  false
      true
```

Similarly surprising is this:

```
Line 1  puts  2  <  10
   2  puts "2" < "10"

   ❮  true
      false
```

Okay, 2 is less than 10, so no problem. But that last one? Well, the "1" character comes before the "2" character—remember, in a string those are simply characters. The "0" character after the "1" doesn't make the "1" any larger.

One last note before moving on: the comparison methods aren't giving you the strings "true" and "false"; they're giving you the special objects true and false that represent…well, truth and falsity. (Of course, true.to_s gives us the string "true", which is why puts printed true.) true and false are used all of the time in a language construct called *branching*.

Branching

Branching is a simple concept, but it's powerful. It's how we tell Ruby to sometimes do this thing, and other times do that thing. Here's what it looks like in code:

```
Line 1  puts "Hello, what's your name?"
   2    name = gets.chomp
   3    puts "Hello, #{name}."
   4
   5    if name == "Chris"
   6      puts "What a lovely name!"
   7    end
```

```
 ❮  Hello, what's your name?
⇛   Chris
 ❮  Hello, Chris.
    What a lovely name!
```

But if we put in a different name…

```
 ❮  Hello, what's your name?
⇛   Chewbacca
 ❮  Hello, Chewbacca.
```

And that's branching. If what comes after the if keyword is true, you run the code between the if and the end. If what comes after the if keyword is false, you don't. Plain and simple.

You may have noticed that the code is indented between the if and the end keywords. This makes it easier to keep track of the branching that way. Nearly all programmers do this, regardless of the language they're programming in. It may not seem that helpful in this simple example, but when programs get more complex, it makes a big difference. Often, when people send me programs that don't work but they can't figure out why, the issue is something that is both:

- easier to see what the problem is if the indentation is nice, and
- impossible to see what the problem is otherwise.

So, try to keep your indentation nice and consistent. Have your if and end keywords line up vertically, and have everything between them indented. Most Ruby programmers use an indentation of two spaces, so you should, too.

Often, you'd like a program to do one thing if an expression is true and another if it's false. That's what the else keyword is for:

```
Line 1  puts "I am a fortune-teller. Tell me your name:"
    2  name = gets.chomp
    3
    4  if name == "Chris"
    5    puts "I see great things in your future."
    6  else
    7    puts "Your future is...oh my! Look at the time!"
    8    puts "I really have to go. Sorry!"
    9  end
```

```
❮ I am a fortune-teller. Tell me your name:
⇛ Chris
❮ I see great things in your future.
```

Now let's try a different name:

```
❮ I am a fortune-teller. Tell me your name:
⇛ Boromir
❮ Your future is...oh my! Look at the time!
  I really have to go. Sorry!
```

Branching is kind of like coming to a fork in the code: do you take the path for people whose name == "Chris", or else do you take the other, less egocentric, path?

Like the branches of a tree, code can have branches that themselves have branches:

```
Line 1  puts "Hello, and welcome to seventh grade English."
    -  puts "My name is Mrs. Gabbard. And your name is...?"
    -  name = gets.chomp
    -
    5  if name == name.capitalize
    -    puts "Please take a seat, #{name}."
    -  else
    -    puts "#{name}? You mean #{name.capitalize}, right?"
    -    puts "Don't you even know how to spell your name??"
   10    reply = gets.chomp
    -
    -    if reply.downcase == "yes"
    -      puts "Hmmph! Well, sit down!"
```

```
-   else
15    puts "GET OUT!!"
-   end
-end
```

```
❮ Hello, and welcome to seventh grade English.
  My name is Mrs. Gabbard. And your name is...?
⇛ chris
❮ chris? You mean Chris, right?
  Don't you even know how to spell your name??
⇛ yes
❮ Hmmph! Well, sit down!
```

Fine, I'll capitalize my name:

```
❮ Hello, and welcome to seventh grade English.
  My name is Mrs. Gabbard. And your name is...?
⇛ Chris
❮ Please take a seat, Chris.
```

Me and Mrs. Gabbard didn't get along terribly well. (Hah! See that, Mrs. Gabbard, using "Me and Mrs. Gabbard" instead of "Mrs. Gabbard and I"? And I'm a published author now!)

Sometimes it might get confusing trying to figure out where all the `if`s, `else`s, and `end`s go. To keep it all straight, try to write the `end` *at the same time* you write the `if`. So, if you were writing the previous program, this is how it would look first:

```
Line 1 puts "Hello, and welcome to seventh grade English."
  2    puts "My name is Mrs. Gabbard. And your name is...?"
  3    name = gets.chomp
  4
  5    if name == name.capitalize
  6    else
  7    end
```

Then you fill it in with *comments*, stuff in the code to help *you* out, but that Ruby will ignore:

```
Line 1 puts "Hello, and welcome to seventh grade English."
  2    puts "My name is Mrs. Gabbard. And your name is...?"
  3    name = gets.chomp
  4
  5    if name == name.capitalize
  6      # She's civil.
  7    else
  8      # She gets mad.
  9    end
```