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

Programming Ruby 3.3

The Pragmatic Programmers' Guide

by Noel Rappin, with Dave Thomas

Version: P1.0 (January 2024)

Table of Contents

Early Praise for *Programming Ruby 3.3: The Pragmatic Programmers' Guide*

The book has such breadth and depth, making it a useful long-term companion. I'd say this is a big win for the Ruby community.

→ Stefan Magnuson
Software Developer

Programming Ruby 3.3: The Pragmatic Programmers' Guide is a valuable resource to anyone looking to get started with developing software tools and systems in Ruby. Thanks to thorough technical explanations accompanied by demonstrative code examples, this book will equip you with a mastery of all the building blocks of Ruby and help you unlock its full power.

→ Nishant Roy
Engineering Manager

I'm ecstatic to see the book that inspired an entire generation of Rubyists revived. I'm excited to see—and use—what ...

Preface

This is the fifth edition of *Programming Ruby*, which many Ruby developers call “The Pickaxe Book.” It covers Ruby up to and including Ruby 3.3.

Since the previous edition of this book, Ruby has continued to grow and evolve. New syntax has been added; old syntax has been refined. Major new features, such as pattern matching and type signatures, are now part of the language. Tools that didn’t exist or were in their early stages of development then are now in constant use by Ruby developers around the world. The entire ecosystem is thriving.

The Pickaxe Book continues to be your guide to learning Ruby the language and understanding how Ruby’s parts work together and how you can use the most popular and important Ruby tools.

Why Ruby?

When Dave Thomas and Andy Hunt wrote the first edition, they explained the appeal of Ruby. Among other things, they wrote, “When we discovered Ruby, we realized that we’d found what we’d been looking for. More than any other language with which we have worked, Ruby *stays out of your way*. You can concentrate on solving the problem at hand, instead of struggling with compiler and language issues. That’s how it can help you become a better programmer: by giving you the chance to spend your time creating solutions for your users, not for the compiler.”

That belief is even stronger today. More than thirty years after Ruby’s first release on February 24, 1993, Ruby still enables developers to focus on their solutions—from the smallest utility ...

A Word about Ruby Versions

This edition of The Pickaxe Book documents Ruby up to and including Ruby 3.3. New Ruby version releases come out annually on December 25. The book's code was developed against Ruby 3.3, preview 2, but we don't expect substantial changes in the released version of Ruby 3.3.

In this book, we don't typically note what version of Ruby introduced a new feature, but you can find a brief list of the largest changes in Appendix 5, *Ruby Changes*. We recommend referring to the Ruby Evolution page by Victor Shepelev at https://rubyreferences.github.io/rubychanges/evolution.xhtml for a full listing of the changes implemented since Ruby 2.0.

Exactly what version of Ruby did we use to write this book? Let's ask Ruby:

```
$ ruby ...
```

Notation Conventions

Literal code examples are shown using a sans-serif font:

```
class SampleCode
  def run
    #...
  end
end
```

In this book, a class name followed by a hash followed by a method name, as in `Fred#do_something`, is a reference to an instance method (in this case, the method `do_something` of class `Fred`). Class methods are written with a dot as in `Fred.new`, and `Fred.EOF` is a class constant. In other Ruby documentation, you may see class methods written as `Fred::new`. This is perfectly valid Ruby syntax; we just happen to think that `Fred.new` is less distracting to read and is much more common to see in practice.

The decision to use a hash character to indicate instance methods was a tough one. It isn’t valid Ruby syntax, but ...

Road Map

The main text of this book is divided into five parts, each with its own personality and each addressing different aspects of the Ruby language.

Part I, Facets of Ruby, is a Ruby tutorial. It starts with notes on getting Ruby running on your system followed by a short chapter on the terminology and concepts that are unique to Ruby. The initial chapter also includes enough basic syntax so that the other chapters will make sense. The rest of the tutorial is a top-down look at Ruby. There we talk about classes and objects, types, expressions, and all the other things that make up the language. We end with a chapter on unit testing.

Part II, Ruby in Its Setting, investigates one of the great things about Ruby, which is how well it integrates ...

Resources

Visit the Ruby website at http://www.ruby-lang.org to see what's new. You can find a list of community resources, including the official mailing list and Discord server, at https://www.ruby-lang.org/en/community.

And we'd certainly appreciate hearing from you. Comments, suggestions, errors in the text, and problems in the examples are all welcome. Email us at rubybook@pragprog.com.

If you find errors in the book, you can add them to the errata page at https://devtalk.com/books/programming-ruby-3-2-5th-edition/errata. If you're reading the PDF version of the book, you can also report an erratum by clicking the link in the page footers.

You'll find links to the source code for almost all of the book's code examples at https://www.pragprog.com/titles/ruby5 ...

Acknowledgments

In January 2001, I bought myself a programming book as a birthday present. It had a pickaxe on the cover, and it was written by the two people who wrote *The Pragmatic Programmer*. It was about this new programming language from Japan that I had heard about on the Extreme Programming mailing list, and which sounded very interesting.

I can’t thank Dave Thomas and Andy Hunt enough. It’s hard to even begin to list what I’ve gained from purchasing that initial book and from my association with The Pragmatic Bookshelf. Thanks also to Chad Fowler for his work on subsequent versions of the book. I inherited a great book from the three of you, and I hope this version will continue to bring people into the Ruby language and the Ruby community. ...

Part 1Facets of Ruby

Welcome to Ruby! Part I is a tutorial covering all the Ruby you'll need to be able to understand a good-sized Ruby application. We'll explore the most important parts of the syntax and the standard library, and go beyond the basics in a couple of places where Ruby has a particularly interesting or powerful tool at hand.

Chapter 1
Getting Started

We're going to spend a lot of time in this book talking about the Ruby language. Before we do, we want to make sure you can get Ruby installed and running on your computer. That way, you can try the sample code and experiment on your own as you read along. If you want to learn Ruby you should get into the habit of writing code as you're reading.

If you aren't comfortable with using a command line, we can help. Please turn to Appendix 3, *Command-Line Basics*, and we'll give you all the information you need to get started.

Installing Ruby

There is a good chance your operating system already has Ruby installed. Try typing `ruby --version` at a command prompt—you may be pleasantly surprised. But you're likely to find that the Ruby version is out of date. For example, at the time of this writing, MacOS ships with Ruby 2.6.10, which is multiple versions behind the current Ruby.

The examples in this book are written against Ruby 3.3. While most of the code will work in older versions of Ruby, for performance and security reasons you should try to get on the most current version. Refer to Appendix 5, *Ruby Changes*, for a listing of the features added and changes made to Ruby at each iteration.

You can install Ruby in a variety of different ways, so providing general ...

Installing Ruby for Windows

Ruby isn’t available as a default option in Windows the way it’s in Unix distributions or MacOS, but it can be installed and used and can interact with the underlying environment to automate Windows-specific resources.

We’re going to focus on two ways to install Ruby on Windows: using the Windows Subsystem for Linux (WSL)[11], which allows you to run a Linux command-line terminal in your Windows system, and using RubyInstaller[12] to install a Windows application that lets you execute Ruby programs.

The two different kinds of Ruby can both be installed on the same machine and have different purposes. Using WSL gives you a command shell that’s effectively a Linux distribution, allowing you to seamlessly use any of ...

Running Ruby

Now that Ruby is installed, you'd probably like to run some programs. Unlike compiled languages, you have two ways to run Ruby: you can type in code interactively, or you can create program files and run them. Typing in code interactively is a great way to experiment with the language, but for code that's more complex or code that you'll want to run more than once, you'll need to create program files and run them. But, before we go any further, let's test to see whether Ruby is installed. Bring up a fresh command prompt, and type this:

```
$ ruby --version
ruby 3.3.0dev (2023-11-01T17:47:26Z master 909afcb4fc) [arm64-darwin23]
```

Technically, you can run Ruby interactively by typing `ruby` at the shell prompt. You'll get a blank ...

Creating Ruby Programs

The most common way to write Ruby programs is to put the Ruby code in one or more text files. You'll use a text editor or an Integrated Development Environment (IDE) to create and maintain these files—many popular editors, including Visual Studio Code, vim, Sublime Text, and RubyMine, feature Ruby support. You'll then run the files either from within the editor or from the command line. Both techniques are useful. You might run single-file programs from within the editor and more complex programs from the command line.

Let's create a short Ruby program and run it. Open a terminal window and create an empty directory somewhere, perhaps you could call it `pickaxe`.

Then, using your editor of choice, create the file `myprog.rb ...`

Getting More Information about Ruby

As the volume of the Ruby libraries has grown, it has become impossible to document them all in one book; the standard library that comes with Ruby now contains more than 9,000 methods. The official Ruby documentation is at https://docs.ruby-lang.org, with official pages for the different versions of the core and standard library located there. irb will also give you documentation of standard method names as you type.

Much of this documentation is generated from comments in the source code using a tool called RubyDoc, which we'll look at in Chapter 19, *Documenting Ruby*. The RubyDoc site at https://www.rubydoc.info contains documentation for Ruby projects that use RubyDoc. Third-party libraries in the Ruby ...

What’s Next

Now that you’re up and running, it’s time to learn how Ruby works. First, we’ll do a quick overview of the main features of the language.

Footnotes

[2] https://docs.microsoft.com/en-us/windows/wsl/install

[3] https://www.docker.com

[4] https://replit.com

[5] https://github.com/rbenv/rbenv

[6] https://rvm.io

[7] https://github.com/postmodern/chruby

[8] https://asdf-vm.com

[9] https://www.jruby.org

[10] https://github.com/oracle/truffleruby

[11] https://docs.microsoft.com/en-us/windows/wsl

[12] https://rubyinstaller.org

[13] https://docs.microsoft.com/en-us/windows/wsl

[14] https://rubyinstaller.org

Chapter 2
Ruby.new

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 ...

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, ...

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 4, *Ruby Language Reference*, 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 ...
```

Arrays and Hashes

Ruby provides a few different ways to combine objects into collections. Most of the time, you'll use two of them: Arrays and Hashes. An `Array` is a linear list of objects, you retrieve them via their index, which is the number of their place in the array, starting at zero for the first slot. A `Hash` is an association, meaning it's a key/value store where each value has an arbitrary key, and you retrieve the value via that key. Both arrays and hashes grow as needed to hold new elements. Any particular array or hash can hold objects of differing types; you can have an array containing an integer, then a string, then a floating-point number, as we'll see in a minute.

You can create and initialize a new array object using an *array* ...

Symbols

Often, when programming, you need to use the same string over and over. Perhaps the string is a key in a Hash, or maybe the string is the name of a method. In that case, you'd probably want the access to that string to be immutable so its value can't change, and you'd also want accessing the string to be as fast and use as little memory as possible.

This brings us to Ruby's *symbols*. Symbols aren't exactly optimized strings, but for most purposes, you can think of them as special strings that are immutable, are only created once, and are fast to look up. Symbols are meant to be used as keys and identifiers, while strings are meant to be used for data.

A symbol literal starts with a colon and is followed by some kind of name:

```
walk( ...
```

Control Structures

Ruby has all the usual control structures, such as `if` statements and `while` loops. Java or JavaScript programmers may be surprised by the lack of braces around the bodies of these statements. Instead, Ruby uses the keyword `end` to signify the end of a body of a control structure:

intro/weekdays.rb

```
today = Time.now

if today.saturday?
  puts "Do chores around the house"
elsif today.sunday?
  puts "Relax"
else
  puts "Go to work"
end
```

Produces:

```
Go to work
```

One thing you might find unusual is that in the second clause Ruby uses the keyword `elsif`—one word, missing an “e”—to indicate “else if”. Breaking that keyword up into `else if` would be a syntax error.

Similarly, `while` statements are terminated with

Regular Expressions

Most of Ruby's built-in types will be familiar to all programmers. A majority of languages have strings, integers, floats, arrays, and so on. But not all languages have built-in support for regular expressions the way that Ruby or JavaScript do. This is a shame because regular expressions, although cryptic, are a powerful tool for working with text. And having them built in rather than tacked on through a library interface, makes a big difference.

Entire books have been written about regular expressions (for example, *Mastering Regular Expressions* by Jeffrey Friedl), so we won't try to cover everything in this short section. Instead, we'll look at a few examples of regular expressions in action. You'll find more coverage ...

Blocks

This section briefly describes one of Ruby's particular strengths—*blocks*. A code block is a chunk of code you can pass to a method, as if the block were another parameter. This is an incredibly powerful feature, allowing Ruby methods to be extremely flexible. One of this book's early reviewers commented at this point: "This is pretty interesting and important, so if you weren't paying attention before, you should probably start now." We still agree.

Syntactically, code blocks are chunks of code that can be delimited one of two ways: between braces or between `do` and `end`. This is a code block at the end of a message call:

```
foo.each { puts "Hello" }
```

This is also a code block at the end of a message call:

```
foo.each do
  club.enroll ...
```

Reading and ‘Riting

Ruby comes with a comprehensive library to manage input and output (I/O). But, in most of the examples in this book, we’ll stick to a few simple methods. We’ve already come across methods that write output: `puts` writes its arguments with a newline after each; `p` also writes its arguments but will produce more debuggable output. Both can be used to write to any I/O object, but, by default, they write to the standard output stream.

You can read input into your program in many ways. Probably the most traditional one is to use the method `gets`—short for “get string”—which returns the next line from your program’s standard input stream:

```
line = gets
print line
```

Because `gets` returns `nil` when it reaches the end of input, you can ...

Command-Line Arguments

When you run a Ruby program from the command line, you can pass in arguments. These are accessible from your Ruby code in two different ways.

First, the global array `ARGV` contains each of the arguments passed to the running program. Create a file called `cmd_line.rb` that contains the following:

```
puts "You gave #{ARGV.size} arguments"
p ARGV
```

When we run it with arguments, we can see that they get passed in:

```
$ ruby cmd_line.rb ant bee cat dog
You gave 4 arguments
["ant", "bee", "cat", "dog"]
```

Often, the arguments to a program are the names of files that you want to process. In this case, you can use a second technique: the variable `ARGF` is a special kind of I/O object that acts like ...

Commenting Ruby

Ruby has two ways of adding comments to source code, one of which you'll use, and the other you'll almost certainly not use. The common one is the `#` symbol—anything after that symbol until the end of the line is a comment and is ignored by the interpreter. If the next line continues the comment, it needs its own `#` symbol.

Ruby also has a rarely used multiline comment, where the first line starts with `=begin` and everything is a comment until the code reaches `=end`. Both the `=begin` and `=end` must be at the very beginning of the line, they cannot be indented.

While we did just say that Ruby ignores comments, Ruby uses a small number of "magic comments" for configuration options on a per-file basis. These comments have the form of ...

What’s Next

We finished our lightning-fast tour of some of the basic features of Ruby. We took a look at objects, methods, strings, containers, and regular expressions. We saw some simple control structures and looked at some rather nifty iterators. We hope this chapter has given you enough ammunition to be able to attack the rest of this book.

It’s time to move on and move up—up to a higher level. Next, we’ll be looking at classes and objects, things that are at the same time both the highest-level constructs in Ruby and the essential underpinnings of the entire language.

Chapter 3
Classes, Objects, and Variables

From the examples we've shown so far, you may be wondering about our earlier assertion that Ruby is an object-oriented language. Well, here is where we justify that claim. We're going to be looking at how you create classes and objects in Ruby and at some of the ways that Ruby is more flexible than other object-oriented languages.

In *Ruby Is an Object-Oriented Language*, we state that everything we manipulate in Ruby is an object. And every object in Ruby was instantiated either directly or indirectly from a class. In this chapter, we'll look in more depth at creating and manipulating those classes.

Defining Classes

Let’s give ourselves a simple problem to solve. Suppose we’re running a secondhand bookstore. Every week, we do stock control. A gang of clerks uses portable bar-code scanners to record every book on our shelves. Each scanner generates a comma-separated value (CSV) file containing one row for each book scanned. The row contains (among other things) the book’s ISBN and price. An extract from one of these files looks something like this:

tut_classes/stock_stats/data.csv

```
"Date","ISBN","Price"
"2013-04-12","978-1-9343561-0-4",39.45
"2013-04-13","978-1-9343561-6-6",45.67
"2013-04-14","978-1-9343560-7-4",36.95
```

Our job is to take all the CSV files and work out how many of each title we have, as well as the total list price of the ...

Objects and Attributes

The `BookInStock` objects we've created so far have an internal state (the ISBN and price). That state is private to those objects—no other object can access an object's instance variables. In general, this is a Good Thing. It means that the object is solely responsible for maintaining its own consistency. (We feel obligated to note here that there's no such thing as perfect privacy in Ruby, and you shouldn't depend on Ruby's language privacy for security purposes.)

A totally secretive object is pretty useless—you can create it, but then you can't do anything with it. You'll normally define methods that let you access and manipulate the state of an object, allowing the outside world to interact with the object. These externally ...

Classes Working with Other Classes

Our original challenge was to read in data from multiple CSV files and produce various simple reports. So far, all we have is `BookInStock`, a class that represents the data for one book.

During object-oriented design, you identify external things and make them classes in your code. But there's another source of classes in your designs—the classes that correspond to things inside your code itself. For example, we know that the program we're writing will need to consolidate and summarize CSV data feeds. But that's a passive statement. Let's turn it into a design by asking ourselves *what* does the summarizing and consolidating. And the answer (in our case) is a *CSV reader*. Let's make it into a class as follows: ...

Specifying Access Control

When designing the interface for a class, it’s important to consider how much of your class you’ll expose to the outside world. Allow too much access into your class, and you risk increasing the amount that different classes depend on each other’s internal implementation, which is called *coupling*. Users of your class will be tempted to rely on details of your class’s implementation rather than on its logical interface. The good news is that the only easy way to change an object’s state in Ruby is by calling one of its methods. If you control access to the methods, you control access to the object. A good rule of thumb is never to expose methods that could leave an object in an invalid state.

Ruby gives you three levels ...

Variables

Now that we've gone through the trouble of creating all these objects, let's make sure we don't lose them. Variables are used to keep track of objects; each variable holds a reference to an object. Let's confirm this with some code:

```
person = "Tim"
puts "The object in 'person' is a #{person.class}"
puts "The object has an id of #{person.object_id}"
puts "and a value of '#{person}'"
```

Produces:

```
The object in 'person' is a String
The object has an id of 60
and a value of 'Tim'
```

On the first line, Ruby creates a new string object with the value `Tim`. A reference to this object is placed in the local variable `person`. A quick check shows that the variable has indeed taken on the personality of a string, with ...

Reopening Classes

While we're talking about classes in Ruby, we feel like we should at least mention one of the most unique features of Ruby's class structure: the ability to reopen a class definition and add new methods or variables to it at any time, even classes that are part of the third-party tools or the standard library.

In other words, if you have something like this in Ruby:

```
class Book
  attr_accessor :title

  # and a bunch of other stuff
end
```

Later, you can do this:

```
class Book
  def uppercase_title
    title.upcase
  end
end
```

If you declare `class Book` and a `Book` class already exists, Ruby won't give an error, and the new definitions in the second declaration will be added to the existing class. This is true even ...

What’s Next

There’s more to say about classes and objects in Ruby. We still have to look at class methods and concepts such as mixins and inheritance. We’ll do that in Chapter 6, *Sharing Functionality: Inheritance, Modules, and Mixins*. But, for now, know that everything you manipulate in Ruby is an object and that objects start their lives as instances of classes. And one of the most common things we do with objects is to create collections of them. But that’s the subject of our next chapter.

Chapter 4
Collections, Blocks, and Iterators

Most real programs have to manage collections of data: the people in a course, the songs in your playlist, the books in the store, and so on. Ruby comes with two classes that are commonly used to handle these collections: *arrays* and *hashes*. A Ruby array is an ordered collection of data. A Ruby hash is a key/value pair, equivalent to a Python dictionary, a Java Map, or a JavaScript object. Mastery of these two classes, and their large interfaces, is an important part of being an effective Ruby programmer.

But it isn’t only these two classes that give Ruby its power when dealing with collections. Ruby also has a block syntax that lets you encapsulate chunks of code. When paired with collections, these ...

Arrays

The class `Array` holds a collection of object references. Each object reference occupies a position in the array, identified by an integer index. You can create arrays by using literals or by explicitly creating an `Array` object. A literal array is a comma-delimited list of objects between square brackets:

```
a = [3.14159, "pie", 99]
a.class   # => Array
a.length  # => 3
a[0]      # => 3.14159
a[1]      # => "pie"
a[2]      # => 99
a[3]      # => nil
```

You can create an empty array with either `[]` or by directly calling `Array.new`:

```
b = Array.new
b.class   # => Array
b.length  # => 0
b[0] = "second"
b[1] = "array"
b         # => ["second", "array"]
```

As the example shows, array indices start at zero. Index an array with a non-negative ...

Hashes

Hashes (sometimes known as *associative arrays*, *maps*, or *dictionaries*) are similar to arrays in that they are indexed collections of object references. But, while you index arrays with integers, you index a hash with objects of any type, most often symbols and strings but also regular expressions or anything else in Ruby. When you store a value in a hash, you actually supply two objects: the index, which is called the *key*, and the *value*, or entry, to be stored with that key. You can subsequently retrieve the entry by indexing the hash with the same key value that you used to store it.

Why Are They Called Hashes?

The data structure that ...

Digging

Often data isn't simply a single hash or array but comes in a complex package that combines hashes and arrays. Accessing data in a complicated structure can be a pain, but Ruby provides a shortcut with the `dig` method.

The `dig` method, which is defined for `Array`, `Hash`, and `Struct`, allows you to "dig" through a complicated data structure in a single command.

```
data = {
  mcu: [
    {name: "Iron Man", year: 2010, actors: ["Robert Downey Jr.", "Gwyneth
Paltrow"]}
  ],
  starwars: [
    {name: "A New Hope", year: 1977, actors: ["Mark Hamill", "Carrie Fisher"]}
  ]
}
data[:mcu][0][:actors][1]     # => "Gwyneth Paltrow"
data.dig(:mcu, 0, :actors, 1) # => "Gwyneth Paltrow"
```

The biggest advantage of using `dig` ...

Word Frequency: Using Hashes and Arrays

Let’s round out this discussion of hashes and arrays with a program that calculates the number of times each word occurs in some text. (So, for example, in this sentence, the word *the* occurs two times.)

The problem breaks down into two parts. First, given some text as a string, return a list of words. That sounds like an array. Then, build a count for each distinct word. That sounds like a use for a hash—we can index it with the word and use the corresponding entry to keep a count.

Let’s start with the method that splits a string into words:

tut_containers/word_freq/words_from_string.rb

```
def words_from_string(string)
  string.downcase.scan(/[\w']+/)
end
```

This method uses two useful string ...

Blocks and Enumeration

In our program that wrote out the results of our word frequency analysis, we had the following loop:

```
top_five.reverse_each do |word, count|
  puts "#{word}:  #{count}"
end
```

The method `reverse_each` is an example of an *iterator*—a general term for a method that invokes a block of code repeatedly. Ruby also uses the term *enumerator* for such a method.

The most general iterator in Ruby is `each`, which takes a block and invokes the block once for each element in the collection. In this case, we're using `reverse_each`, a shortcut method that invokes the block once for each element of the list, but in reverse order.

Enumerator methods can have different behaviors beyond just executing the block of code. A Ruby ...

What’s Next

Collections, blocks, and iterators are core concepts in Ruby. The more you write in Ruby, the more you’ll find yourself moving away from conventional looping constructs. Instead, you’ll write classes that support iteration over their contents. And you’ll find that this code is compact, easy to read, and a joy to maintain. If this all seems too weird, don’t worry. After a while, it’ll start to come naturally. And you’ll have plenty of time to practice as you use Ruby libraries and frameworks. Now, let’s talk more about how Ruby lets you define and call methods.

Chapter 5
More about Methods

So far in this book, we've been defining and using methods without much thought. Now it's time to get into the details.

Defining a Method

As we've seen, a method is defined using the keyword `def`.

The keyword `def` creates a method and returns the name of the method as a symbol, which, as we saw in *Specifying Access Control*, allows us to put decorator methods like `private` before the declaration.

The body of a method contains normal Ruby expressions. The return value of a method is the value of the last expression executed or the argument of an explicit `return` expression.

An important fact about `def` is that if you define a method a second time, Ruby won't raise an error, it'll print a warning, and then it'll redefine the method using the second definition:

```
class Batman
  def who_is_robin
    puts "Dick Grayson"
  end

  def who_is_robin
 puts  ...
```

Calling a Method

You call a method by optionally specifying a receiver, giving the name of the method, and optionally passing some arguments and an optional block. Here's a code fragment that shows us calling a method with a receiver, a positional argument, a keyword argument, and a block:

```
connection.download_mp3("jitterbug", speed: :slow) { |p| show_progress(p) }
```

In this example, the object `connection` is the receiver, `download_mp3` is the name of the method, the string `"jitterbug"` is the positional parameter, the key/value pair `speed: :slow` is a keyword parameter, and the code between the braces is the associated block argument. When the method is called, Ruby invokes the method in that object, and inside that method, `self` is set to ...

What’s Next

A well-written Ruby program will typically contain many methods, each quite small, so it’s worth getting familiar with the options available when defining and using them. At some point, you’ll probably want to read Chapter 25, *Language Reference: Objects and Classes*, to see exactly how arguments in a method call get mapped to the method’s formal parameters when you have combinations of default parameters and splat parameters.

Now that we have methods, we need to talk about how different classes can share functionality defined by their methods, so it’s time to talk about inheritance and modules.

Chapter 6
Sharing Functionality: Inheritance, Modules, and Mixins

One of the principles of good software design is the elimination of unnecessary duplication. We work hard to make sure that each concept in our application is expressed only once in our code. Why? Because the world changes. And when you adapt your application to each change, you want to know that you've changed exactly the code you need to change. If each real-world concept is implemented at a single point in the code, this becomes vastly easier.

We've already seen how classes help reduce duplication. All the methods in a class are automatically accessible to instances of that class. But we want to do other, more general types of sharing. Maybe we're dealing with an application ...

Inheritance and Messages

In a previous chapter, we saw that when the `puts` method needs to convert an object to a string, it calls that object's `to_s` method. But we've also written our own classes that don't explicitly implement `to_s`. Despite this, instances of these classes respond successfully when we call `to_s` on them. How this works has to do with inheritance and how Ruby uses it to determine what method to run when you send a message to an object.

Inheritance allows you to create a class that's a specialization of another class. This specialized class is called a *subclass* of the original, and the original is a *superclass* of the subclass. People also refer to this relationship as *child* and *parent* classes.

The basic mechanism of subclassing ...

Modules

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits:

- Modules provide a namespace and prevent name clashes.
- Modules can be included in other classes, a facility known as a *mixin*.

Namespaces

As you start to write bigger Ruby programs, you'll find yourself producing chunks of reusable code—libraries of related routines that are applicable in many different contexts. You'll want to break this code into separate files so the contents can be shared among different Ruby programs.

Often this code will be organized into classes, so you'll probably stick a class into each file. But sometimes you want to group things together that don't naturally form a class—for example, the methods that you ...

Inheritance, Mixins, and Design

Inheritance and mixed-in modules both allow you to write code in one place and use that code in multiple classes. So, when do you use each?

As with most questions of design, the answer is, well…it depends. But over the years, developers have come up with some general guidelines to help us decide.

First, let's look at subclassing. Classes in Ruby are related to the idea of types. It would be natural to say that `"cat"` is a string and `[1, 2]` is an array. And that's another way of saying that the class of `"cat"` is `String` and the class of `[1, 2]` is `Array`. When we create our own classes, you can think of it as adding new types to the language. And when we subclass either a built-in class or our own class, we're creating ...

What’s Next

In this chapter, we looked at using Ruby modules to encapsulate code into namespaces and to share code by using the `include` method to mix modules into classes. We also talked about how module inclusion affects method lookup and when to use mixins versus inheritance.

Now that we’ve learned some of Ruby’s class and object structure, let’s look at some of the classes that Ruby uses for standard types.

Footnotes

[15] http://www.rubyonrails.org

Chapter 7
Basic Types: Numbers, Strings, and Ranges

We've been having fun implementing programs using arrays, hashes, and procs, but we haven't yet covered the most basic types in Ruby: numbers, strings, and ranges. Let's spend a few pages on these basic building blocks now.

Numbers

Ruby supports integers, floating-point, rational, and complex numbers. Integers can be of any length (up to a maximum determined by the amount of free memory on your system) and are of type `Integer`.

Integers are assumed to be decimal base 10, but, you can write integers using a leading sign as an optional base indicator—`0` for octal, `0x` for hex, or `0b` for binary (and `0d` for decimal)—followed by a string of digits in the appropriate base.

Underscore characters are ignored in the digit string, you'll see them used in place of commas in larger numbers.

```
123456                  => 123456   # base 10
0d123456                => 123456   # base 10
123_456                 => 123456   # underscore ignored
-543                    => -543     # negative number
0xaabb                  => 43707    # hexadecimal
0377 => 255  ...
```

Strings

Ruby strings are sequences of characters. They normally hold printable characters, but that isn’t a requirement; a string can also hold binary data. Strings are instances of class `String` and are often created using string literals —sequences of characters between delimiters.

Ruby has a lot of different ways to create string literals, which differ in how much processing is done on the characters in the string. One kind of processing is an *escape sequence*. An escape sequence allows you to represent data that is otherwise impossible to represent in the string. Escape sequences in Ruby start with a backslash (`\`).

The simplest literal in Ruby is the single-quoted string. Inside a single-quoted string, only two escape sequences are recognized. ...

Ranges

Ranges occur everywhere: January to December, 0 to 9, rare to well done, lines 50 through 67, and so on. If Ruby is to help us model reality, it seems natural for it to support these ranges. In fact, Ruby goes one better: it uses ranges to implement sequences and intervals.

Ranges as Sequences

The first and perhaps most natural use of ranges is to express a sequence of values. Sequences have a start point, an end point, and a way to produce successive values in the sequence. In Ruby, these sequences are created using the `..` and `...` range operators. The two-dot form creates an inclusive range, and the three-dot form creates a range that excludes the specified high value:

```
1..10
"a".."z"
0...3
```

If you’re looking for a way to remember ...

What’s Next

In this chapter, we covered some of Ruby’s most commonly used types: numbers, strings, and ranges. We showed how to create literals and how to use the functionality of these types. Now let’s look at one of Ruby’s most powerful standard types—regular expressions.

Footnotes

[16] https://home.unicode.org

Chapter 8
Regular Expressions

We spend much of our time in Ruby working with strings, so it seems reasonable for Ruby to have great tools for working with those strings. As we've seen, the `String` class itself is no slouch—it has more than 100 methods. But it still can't do everything on its own. For example, we might want to see whether a string contains two or more repeated characters, or we might want to replace every word longer than fifteen characters with its first five characters and an ellipsis. This is when we turn to the power of regular expressions.

Regular expressions are powerful and are used in many languages besides Ruby. In this chapter, we'll cover the basics of what regular expressions can do in Ruby; later, in *Regular Expressions ...*

What Regular Expressions Let You Do

A regular expression is a pattern that can be matched against a string. It can be a simple pattern, such as *the string must contain the sequence of letters "cat,"* or the pattern can be complex, such as *the string must start with a protocol identifier, followed by two literal forward slashes, followed by…*, and so on. This is cool in theory. But what makes regular expressions so powerful is what you can do with them in practice:

- You can test a string to see whether it matches a pattern.
- You can extract from a string the sections that match all or part of a pattern.
- You can change the string, replacing the parts that match a pattern.

Ruby provides built-in support that makes pattern matching, extraction, and substitution ...

Creating and Using Regular Expressions

Ruby has many ways of creating a regular expression pattern. By far, the most common is to write the pattern between forward slashes. Thus, the pattern `/cat/` is a regular expression literal in the same way that `"cat"` is a string literal.

`/cat/` is an example of a simple pattern. It matches any string that contains the substring `cat`. In fact, inside a regular expression pattern, all characters except `.`, `|`, `(`, `)`, `[`, `]`, `{`, `}`, `+`, `\`, `^`, `$`, `*`, and `?` match themselves. So, at the risk of creating something that sounds like a logic puzzle, here are some patterns and examples of strings they match and don't match:

`/cat/`	Matches `"dog and cat"` and `"catch"` but not `"Cat"` or `"c.a.t."`
`/123/`	Matches `"86512312"` and

Regular Expression Patterns

Like most things in Ruby, regular expressions are just objects—they are instances of the class `Regexp`. This means you can assign them to variables, pass them to methods, and so on:

```
str = "dog and cat"
pattern = /nd/
pattern.match?(str) # => true
str.match?(pattern) # => true
```

You can also create regular expression objects by calling the `Regexp` class's `new` method or by using the arbitrary delimiter `%r{...}` syntax. The `%r` syntax is particularly useful when creating patterns that contain forward slashes:

```
/mm\/dd/            # => /mm\/dd/
Regexp.new("mm/dd") # => /mm\/dd/
%r{mm/dd}           # => /mm\/dd/
```

Playing with Regular Expressions

If you're like us, you'll sometimes get confused by regular expressions. ...

Regular Expression Syntax

We said earlier that, within a pattern, all characters match themselves except for `. | ( ) [ ] { } + \ ^ $ *` and `?`. Those characters all have special meanings in regular expression patterns. First, always remember that you need to escape any of these characters with a backslash if you want them to be treated as regular characters to match:

```
show_regexp('yes | no', /\|/)       # => yes ->|<- no
show_regexp('yes (no)', /\(no\)/)   # => yes ->(no)<-
show_regexp('are you sure?', /e\?/) # => are you sur->e?<-
```

Now let’s see what some of these characters mean if you use them without escaping them.

Anchors

By default, a regular expression will try to find the first match for the pattern in a string. Match `/iss/` ...

What’s Next

So that’s it! If you’ve made it this far, consider yourself a regular expression ninja. Get out there and match some strings. Now we’ll take a more general look at Ruby expressions.

Chapter 9
Expressions

So far, we've been fairly cavalier in our use of expressions in Ruby. After all, `a = b + c` is pretty standard stuff. That said, Ruby expressions are different than what you might see in JavaScript, Python, or Java, and there's a lot of power and flexibility there. You could write a whole lot of Ruby code without reading any of this chapter, but it wouldn't be as much fun.

One of the first differences in Ruby is that anything that can reasonably return a value does: just about everything is an expression. What does this mean in practice?

Well, for one thing, we have the ability to chain statements together:

```
a = b = c = 0
[3, 1, 7, 0].sort.reverse # => [7, 3, 1, 0]
```

Code structures that are statements in languages ...

Operator Expressions

Ruby has the basic set of operators (`+`, `-`, `*`, `/`, and so on) as well as a few surprises. A complete list of the operators, and their precedences, is given in Table 19, *Ruby operators (high to low precedence)*.

In Ruby, many binary operators are implemented as method calls. For example, when you write `a * b + c`, you're actually asking the object referenced by `a` to execute the method `*`, passing in the parameter `b`. You then ask the object that results from that calculation to execute the `+` method, passing `c` as a parameter. This is the same as writing the following (perfectly valid) Ruby:

```
a, b, c = 1, 2, 3
a * b + c     # => 5
(a.*(b)).+(c) # => 5
```

Because everything is an object and because you can redefine instance ...

Command Expressions

If you enclose a string in backquotes (sometimes called *backticks*) or use the delimited form `%x{...}`, the string will (by default) be executed as a command by your underlying operating system. The value returned is the standard output of that command. Newlines will not be stripped, so the value you get back will likely have a trailing return or linefeed character.

```
`date`                # => "Thu Nov  2 17:16:02 CDT 2023\n"
`ls`.split[34]        # => "irb.md"
%x{echo "hello there"} # => "hello there\n"
```

You can use expression expansion and all the usual escape sequences in the command string:

```
0..3.each do |i|
  status = `dbmanager status id=#{i}`
  # ...
end
```

The exit status of the command is available in the global ...

Assignment

Almost every example we've given so far in this book has featured assignment. It's about time we said something about it.

An assignment statement sets the variable or attribute on its left side (the *lvalue*) to refer to the value on the right (the *rvalue*). It then returns that rvalue as the result of the assignment expression. This means you can chain assignments, and you can perform assignments in some unexpected places:

```
a = b = 1 + 2 + 3
a         # => 6
b         # => 6
a = (b = 1 + 2) + 3
a         # => 6
b         # => 3

File.open(name = gets.chomp)
```

Ruby has two basic forms of assignment. The first assigns an object reference to a variable or constant. This form of assignment is hardwired into the language:

```
instrument = "piano"
MIDDLE_A ...
```

Conditional Execution

Ruby has several different mechanisms for the conditional execution of code; they should feel similar to other programming languages, but many have some neat twists. Before we get into them, we need to spend a short time looking at boolean expressions.

Boolean Expressions

Ruby has a simple definition of truth. Any value that isn't `nil` or the constant `false` is `true`—`"cat"`, `99`, `0`, and `:a_song`—are all considered true. An empty string `""`, an empty array `[]`, and an empty hash `{}` are all true in Ruby. (You'll sometimes see Rubyists refer to the set of all false values as "falsey" and the set of all true values as "truthy".)

In this book, when we want to talk about a general true or false value, we use regular Roman type: true ...

Loops and Iterators

We discussed Ruby blocks and iterators back in Chapter 4, *Collections, Blocks, and Iterators*. In this section, we'll talk about all of Ruby's looping constructs in more depth.

Loops

Ruby has primitive built-in looping constructs, separate from the iterator constructs we've already seen.

The most basic loop of all that Ruby provides is a built-in iterator method called `loop`:

```
loop do
  # block ...
end
```

The `loop` iterator calls the associated block forever (or at least until you break out of the loop, but you'll have to read ahead to find out how to do that).

The `while` loop executes its body zero or more times as long as its condition is true. For example, this idiom reads until the input is exhausted, assigning each ...

Pattern Matching

When you're dealing with a complicated data structure, but you only need to use part of the structure, it can be awkward to access the structure through regular `[]` methods, with something like `movies[:mcu][1][:actors][1][:first_name]`. Not only is the access complicated but validating that the data has the shape you're looking for can also be difficult.

In Ruby, *pattern matching* is designed to make both these tasks easier by allowing you to specify the structure of the data as a pattern, and assign values to the parts of the data that match. Please note that many programming languages have features they call "pattern matching," but Ruby's implementation is somewhat different from many of these. The most similar seems to be in ...

What’s Next

In this chapter, we went through a lot of different Ruby expressions, from assignment to math to logic to loops to patterns. Next, we’ll look at Ruby’s exception handling and see what to do when things go wrong.

Chapter 10
Exceptions

So far, we've been developing code in Pleasantville, a wonderful place where nothing ever, ever goes wrong. Every library call succeeds, users never enter incorrect data, and resources are plentiful and cheap. Well, that's about to change.

In the real world, errors happen. Good programs (and programmers) anticipate them and arrange to handle them gracefully. This isn't always as easy as it may sound. Often the code that detects an error doesn't have the context to know what to do about it. For example, attempting to open a file that doesn't exist is acceptable in some circumstances and is a fatal error at other times. What's your file-handling module to do?

One approach is to use return codes to signal errors (for example, ...

The Exception Class

Information about an exception is encapsulated in an object of class `Exception` or in one of class `Exception`'s children. Ruby predefines a tidy hierarchy of exceptions, see https://docs.ruby-lang.org/en/master/Exception.xhtml for the full list. As we'll see later, this hierarchy makes handling exceptions considerably easier.

The most important subclass of `Exception` is `StandardError`. The `StandardError` exception and its subclasses represent the exceptional conditions that you're going to want to capture in your code. Other subclasses of `Exception` are raised by Ruby internals or system-level problems. Almost all of the time, if you want to capture exceptions, you capture `StandardError` or one of its children.

When you need to ...

Handling Exceptions

Here's some simple code that uses the `open-uri` library to download the contents of a web page and write it to a file, line by line:

tut_exceptions/fetch_web_page/fetch1.rb

```
require "open-uri"
URI.open("https://pragprog.com/news/index.xhtml") do |web_page|
  output = File.open("index.xhtml", "w")
  while (line = web_page.gets)
    output.puts line
  end
  output.close
end
```

What happens if we get a fatal error halfway through? We certainly don't want to store an incomplete page to the output file.

Let's add some exception-handling code and see how it helps. To start exception handling, we enclose the code that could raise an exception in a `begin`/`end` block and use one or more `rescue` clauses to tell Ruby ...

Raising Exceptions

So far, we've been on the defensive, handling exceptions raised by others. It's time to turn the tables and go on the offensive. It's time to raise some… exceptions.

You can raise exceptions in your code with the `raise` method (or its judgmental and less commonly used synonym, `fail`):

```
raise
raise "bad mp3 encoding"
raise InterfaceException, "Keyboard failure"
```

The first form simply reraises the current exception (or raises a `RuntimeError` if no current exception exists). This is used in exception handlers that intercept an exception before passing it on.

The second form creates a new `RuntimeError` exception, setting its message to the given string. This exception is then raised up the call stack.

The third form uses ...

Using Catch and Throw

Although the exception mechanism of `raise` and `rescue` is great for abandoning execution when things go wrong, it's sometimes nice to be able to jump out of some deeply nested construct during normal processing. This is where the rarely used `catch` and `throw` come in handy.

Here's a trivial example. The following code reads a list of words one at a time and adds them to an array. When done, it prints the array in reverse order. But, if any of the lines in the file don't contain a valid word, we want it to abandon the whole process.

tut_exceptions/catch_1.rb

```
word_list = File.open("wordlist")
catch(:done) do
  result = []
  while (line = word_list.gets)
    word = line.chomp
    throw :done unless /^\w+$/ ...
```

What’s Next

In this chapter, we looked at how to make our Ruby code more error-proof by catching and raising exceptions, and you saw how to create your own exception classes that might have their own data. Next up, we’re going to talk about a leading cause of exceptions in code: managing input and output.

Chapter 11
Basic Input and Output

Ruby provides what looks at first sight like two separate sets of input and output (I/O) routines. The first is the simple interface we've been using a lot so far:

```
print "Enter your name: "
name = gets
```

This whole set of I/O-related methods is implemented in the `Kernel` module, including `gets`, `open`, `print`, `printf`, `putc`, `puts`, `readline`, `readlines`, and `test`. The I/O methods are available to all objects, and they make it simple and convenient to write straightforward Ruby programs. These methods typically do I/O to standard input and standard output, which makes them useful for writing simple tasks.

The other way to do I/O, which gives you more control, is to use Ruby's dedicated `IO` classes.

What Is an I/O Object?

Ruby defines a single base class, `IO`, to handle input and output. This base class is subclassed by classes `File` and `BasicSocket` to provide more specialized behavior, but the principles are the same. An `IO` object is a bidirectional channel between a Ruby program and some external resource.

In this chapter, we'll focus on class `IO` and its most commonly used subclass, class `File`.

Opening and Closing Files

You can create a new file object using `File.new`:

```
file = File.new("testfile", "r")
 # ... process the file
file.close
```

The first parameter to the method is the filename. The second is called the *mode string*, which lets you declare whether you're opening the file for reading, writing, or both. Here we opened `testfile` for reading with an `"r"`. We could also have used `"w"` for write or `"r+"` for read-write. The full list of allowed modes appears in Table 28, *Mode values*.

You can also optionally specify file permissions when creating a file. After opening the file, we can write and/or read data as needed and as specified by the mode string. When we're done, as responsible software citizens, we close the file, ...

Reading and Writing Files

The same methods that we've been using for "simple" I/O from standard input and output are available for File objects. So, where `Kernel#gets` reads a line from standard input (or from any files specified on the command line when the script was invoked), `File#gets` reads a line from the file object.

For example, we could create a program called `copy.rb`:

tut_io/copy.rb

```
while (line = gets)
  puts line
end
```

If we run this program with no arguments, it'll read lines from the console and copy them back to the console. Note that each line is echoed once the Return key is pressed. (In this and later examples, we show user input in a bold font.) The `^D` is the end-of-file character on Unix systems.

```
$ ruby copy.rb
```

Talking to Networks

Ruby is fluent in most of the Internet's protocols, both low-level and high-level.

For those who enjoy groveling around at the network level, Ruby comes with a set of classes in the socket library (https://docs.ruby-lang.org/en/master/Socket.xhtml). These give you access to TCP, UDP, SOCKS, and Unix domain sockets, as well as any additional socket types supported on your architecture. The library also provides helper classes to make writing servers easier. Here's a simple program that gets information about our user website on a local web server using the HTTP OPTIONS request:

tut_io/socket.rb

```
require "socket"

client = TCPSocket.open("127.0.0.1", "www")
client.send("OPTIONS /~dave/ HTTP/1.0\n\n", 0)
```

What’s Next

In this chapter, we’ve seen both Ruby’s simple I/O library, implemented as a series of methods in the `Kernel` module, and the more complicated I/O methods in the class `IO` and its children. We’ve also seen how to read and write data.

A common problem with I/O is that it’s slow and blocks programs. A common workaround is to use threading to allow the program to do multiple things at once. Let’s take a look at some of Ruby’s threading options.

Chapter 12
Threads, Fibers, and Ractors

Being able to do more than one thing at the same time is pretty useful. When a computer program has to wait for one task to finish, like an API call to a slow server, multitasking allows it to turn control over to another task and do other useful work while it waits. When a computer has more than one CPU—which, these days, means a computer—the program can split tasks across multiple CPUs. You can achieve tremendous speed boosts this way.

Being able to multitask is also pretty complicated. When a program multitasks, a task can change the state of the data another task is using, so the other task's understanding of the data may no longer be correct. When a program multitasks, its tasks may fight for access ...

Multithreading with Threads

The lowest-level mechanism in Ruby for doing two things at once is to use the `Thread` class. Although threads can in theory take advantage of multiple processors or multiple cores in a single processor, there's a major catch. Many Ruby extension libraries aren't thread-safe because they expect to be protected by the GIL. So, Ruby uses native operating system threads but operates only a single thread at a time. Unless you use the Ractor library, you'll never see two threads in the same application running Ruby code truly concurrently. You'll instead see threads that are busy executing Ruby code while another thread waits on an I/O operation.

Creating Ruby Threads

The code that follows is a simple example. It downloads ...

Running Multiple External Processes

Sometimes you may want to split a task into several process-sized chunks—maybe to take advantage of all those cores in your shiny new processor. Or perhaps you need to run a separate process that was not written in Ruby. Not a problem: Ruby has a number of methods by which you may spawn and manage separate processes.

Spawning New Processes

You have several ways to spawn a separate process. The easiest is to run some command and wait for it to complete. You may find yourself doing this to run a system command or retrieve data from the host system. Ruby lets you spawn a process with the `system` or by using backquote (or backtick) methods:

```
system("tar xzf test.tgz") # => true
spawn("date")              # => 38483\nThu ...
```

Creating Fibers

Although the name “fibers” suggests some kind of lightweight thread, Ruby’s fibers are a mechanism for denoting a block of code that can be stopped and restarted, which is sometimes called a *coroutine*. Fibers in Ruby are *cooperatively multitasked*, meaning that the responsibility for yielding control rests with the individual fibers and not the operating system. Fibers can explicitly yield control, or be set to automatically yield control when its operations are blocked.

Fibers let you write programs that share control without incurring all of the complexity inherent in low-level threading. Let’s look at a simple example. We’d like to analyze a text file, counting the occurrence of each word. We could do this (without using fibers) ...

Understanding Ractors

Ruby 3.0 introduced *ractors*, a Ruby implementation of the Actor pattern for multithreaded behavior. (Experimental support for the feature was originally developed under the name “Guilds.”)

How Ractors Work

Ractors allow true parallelism within a single Ruby interpreter: each ractor maintains its own GIL, allowing for potentially better performance. In order for this to work, ractors have only limited ability to access variables outside their scope and can communicate with each other in only specific, pre-defined ways. (Also, if for some reason you’re running multiple threads inside a single ractor—probably you shouldn’t do this—those threads are subject to the equivalent of a global lock on the ractor and won’t run in parallel.) ...

What’s Next

That covers the basics of threading in Ruby. We’ve talked about basic threads, using system processes, fibers, and ractors. Now let’s look at how we can use testing to help ensure that our code does what we expect.

Footnotes

[17] https://www.freebsd.org/cgi/man.cgi?query=fork

[18] https://github.com/bruno-/fiber_scheduler

Chapter 13
Testing Ruby Code

Automated testing has long been an important part of how Ruby developers validate their code. Not only does testing ensure that the code behaves as expected, but the process of writing tests can also expose weaknesses in the structure of the code. Ruby provides a core library called *minitest* to make it easy to write automated tests. A more complex and fully-featured library, *RSpec*, is also commonly used. The two tools have different terminologies and a slightly different focus. In this chapter, we'll look at how these tools are used for *unit testing*, which is testing that focuses on small chunks of code, typically individual methods or branches within methods.

Why Unit Test?

It’s important to be able to test individual units for many reasons, one of which is that being able to isolate code into testable units is useful for ongoing changes and maintenance. Code in one unit often relies on the correct operation of the code in other units. If one unit turns out to contain bugs, then all the code that depends on that unit is potentially affected. This is a big problem.

When you unit test this code as you write it, two things can happen. First, you’re more likely to find the bug while the code was still fresh in your mind. Second, because the unit test was only interacting with the code you just wrote, when a bug does appear, you only have to look through a handful of lines of code to find it, rather than ...

Testing with Minitest

If all that seems a little abstract, let's look at an example of how you use the minitest library to write automated testing. We'll start with a Roman numeral class. Our first pass at the code is pretty simple; it lets us create an object representing a certain number and display that object in Roman numerals:

unittesting/romanbug.rb

```
# This code has bugs
class Roman
  MAX_ROMAN = 4999

  def initialize(value)
    if value <= 0 || value > MAX_ROMAN
      fail "Roman values must be > 0 and <= #{MAX_ROMAN}"
    end
    @value = value
  end

  FACTORS = [
    ["m", 1000], ["cm", 900], ["d", 500], ["cd", 400],
    ["c", 100], ["xc", 90], ["l", 50], ["xl", 40],
    ["x", 10], ["ix", 9], ["v", 5], ...
```

Structuring Tests

Earlier we asked you to ignore the scaffolding around our tests. Now it's time to look at it.

You include the testing framework facilities in your unit by including `minitest/autorun`.

```
require "minitest/autorun"
```

The `minitest/autorun` module includes `minitest` itself, which has most of the features we've talked about so far. It also includes an alternate `minitest/spec` syntax that's more like RSpec and the `minitest/mock` mock object package. (We're not going to talk about `minitest/spec` syntax in this book. If you want that style of syntax, we recommend actually using RSpec.) Finally, it calls `Minitest.autorun`, which starts the test runner. This is why our test files have been executing tests when invoked just as plain Ruby files. ...

Creating Mock Objects in Minitest

Minitest allows you to create *mock objects*, which are objects that simulate the API of an existing object in the system, typically providing a canned response instead of a more expensive or fragile real response. A minitest mock object can be *verified*, meaning it’ll raise a failure if the methods you expected to be called were not called during the test.

Using these mock object expectations allows for a style of testing where, instead of testing the result of a method by verifying its output, you test the behavior of the method by verifying that it makes expected calls to other methods.

In minitest, a mock object is created like any other Ruby object. Then you add the methods you wish the mock to respond ...

Organizing and Running Tests

The test cases we've seen so far are all runnable Ruby programs. If, for example, the test case for the Roman class was in a file called `test_roman.rb`, we could run the tests from the command line using this:

```
$ ruby test_roman.rb
Run options: --seed 29842
# Running:
..
Finished in 0.000407s, 4914.0040 runs/s, 17199.0141 assertions/s.
2 runs, 7 assertions, 0 failures, 0 errors, 0 skips
```

Minitest is clever enough to run the tests even though there's no main program. It collects all the test case classes and runs each in turn.

If we want, we can ask it to run a particular set of test methods based on a naming pattern:

```
$ ruby test_roman.rb -n test_range
Run options: -n test_range --seed ...
```

Testing with RSpec

The minitest framework has a lot going for it. It’s simple and compatible in style with frameworks from other languages (such as JUnit for Java and pytest for Python).

RSpec has different things going for it. It’s feature-rich (or “complicated,” as some would say), and it has a different vocabulary for discussing testing. It also has a different syntax. Even so, that syntax has influenced the design of other testing tools including the Jasmine and Jest JavaScript testing frameworks.

In RSpec, the focus isn’t on assertions. Instead, you write *expectations*. RSpec is very much concerned with driving the design side of things. As a result, the vocabulary words of RSpec (expectation and specification) are associated with ways ...

What’s Next

In this chapter, we covered Ruby’s two most commonly used test frameworks: minitest and RSpec. Which should you use? Well, if you’re working on a project that already uses one of them, we recommend sticking with that one. There’s not so much difference between the two that it’s worth re-writing all your tests.

If you’re starting a new project, consider whether you like RSpec’s syntax. RSpec is probably more widely used, but some prominent Ruby projects still use minitest, including Ruby on Rails. RSpec has a higher initial complexity but is also more flexible and has more available functionality out of the box. Ultimately, though, it’s a question of which syntax you like better and will get you to write more tests.

We’ve finished ...

Part 2Ruby in Its Setting

Ruby isn't just a programming language. It's an entire ecosystem of tools that enables you to leverage the language and make it valuable for a variety of tasks in a range of different contexts. These tools include the Ruby command-line program itself, the Ruby gems tool for including libraries, and tools for interacting with and debugging Ruby. Ruby also has support for automated documentation, can be used in various editors and different operating systems, and has runtime versions that are optimized for performance in different settings.

Chapter 14
Ruby from the Command Line

If you're using Ruby as a scripting language, you'll be starting it from the command line. In this chapter, we'll look at how to use Ruby as a command-line tool and how to interact with your operating system environment. The two most common ways for a Ruby program to kick off from the command line are with the Ruby interpreter itself and with Rake, a utility that makes it easy to define a series of interrelated tasks. You also might want to create your own command-line programs, and Ruby can help with that as well.

Please note that some of the details of this chapter only apply to Unix-based systems like Linux, MacOS, and WSL.

Calling the Ruby Command

The most direct way to start the Ruby interpreter and run a Ruby program is by calling the `ruby` command from the command line. Regardless of the system in which Ruby is deployed, you have to start the Ruby interpreter somehow, and doing so gives us the opportunity to pass in command-line arguments both to Ruby itself and to the script being run.

A Ruby command-line call consists of three parts, none of which are required: options for the Ruby interpreter itself, the name of a program to run, and arguments for that program.

```
ruby <options> <--> <programfile> <arguments>*
```

You only need the double-dash if you're separating options to Ruby itself from options being passed to the program being run. The simplest Ruby command ...

Ruby Command-Line Options

Following is a complete list of Ruby's command-line options roughly organized by functionality.

Options That Determine What Ruby Runs

-0[octal]
: The 0 flag (the digit zero) specifies the record separator character (\0, if no digit follows). `-00` indicates paragraph mode: records are separated by two successive default record separator characters. `0777` reads the entire file at once (because it's an illegal character). Sets `$/`.

-a
: Autosplit mode when used with `-n` or `-p`; equivalent to executing `$F = $_.split` at the top of each loop iteration.

-c
: Checks syntax only; does not execute the program.

--copyright
: Prints the copyright notice and exits.

-e 'command'
: Executes *command* as one line of Ruby source. Several `-e`s are allowed, ...

Making Your Code an Executable Program

It's a little clunky to have to call `ruby my_code.rb` when you want to run your code; it'd be easier if you could use `my_code.rb`. This is more of an operating system tip than a Ruby tip, but on Unix systems, this is doable with just a couple of steps.

First, you need to make the file executable by changing the mode of the file. To oversimplify slightly, the mode of the file is metadata that determines if the current user can read from, write to, or execute a given file. Typically, on a Unix-based system, the command to make a file executable is `chmod +x <FILENAME>`. The `chmod` command is Unix-speak for "change the mode of the file," `+x` means "make it executable," and FILENAME is the filename. For more on ...

Processing Command-Line Arguments to Your Code

Just as you can pass arguments to methods in your Ruby code, you can pass arguments from the command line to the Ruby script itself. Ruby provides mechanisms for capturing arguments passed to the script and allowing you to read and parse them as part of your script.

ARGV

Any command-line arguments after the program filename are available to your Ruby program in the global array `ARGV`. For instance, assume `test.rb` contains the following program:

```
ARGV.each { |arg| p arg }
```

If you invoke it with the following command line:

```
$ ruby -w test.rb "Hello World" a1 1.6180
```

It'll generate the following output:

```
"Hello World"
"a1"
"1.6180"
```

There's a gotcha here for all you C programmers. In Ruby, ...

Accessing Environment Variables

You can access operating system environment variables using the predefined variable `ENV`. It responds to the same methods as `Hash`. Technically, `ENV` isn't actually a hash—it's a separate class—but if you need to, you can convert it into a `Hash` using `ENV#to_h`.

```
ENV['SHELL']
ENV['HOME']
ENV['USER']
ENV.keys.size
ENV.keys[0, 4]
```

Standard Environment Variables

The values of some environment variables are read by Ruby when it first starts. These variables modify the behavior of the interpreter. Some of the environment variables used by Ruby are listed in the table shown.

Table 4. Ruby environment variables

Variable Name	Description
`DLN_LIBRARY_PATH`	Specifies the search path for dynamically loaded modules. ...

Where Ruby Finds Its Libraries

You use `require` to bring a library into your Ruby program. Some of these libraries are supplied with Ruby, some may have been packaged as RubyGems, and some you may have written yourself. How does Ruby find them?

Let's start with the basics. When Ruby is built for your particular machine, it predefines a set of standard directories to hold library stuff. Where these are depends on the machine in question. You can determine this from the command line with something like this:

```
$ ruby -e 'puts $LOAD_PATH'
```

On our MacOs box, with rbenv installed, this produces the following list:

```
/opt/homebrew/Cellar/rbenv/1.2.0/rbenv.d/exec/gem-rehash
/Users/noel/.rbenv/versions/3.3.0-dev/lib/ruby/site_ruby/3.3.0+0
```

Using the Rake Build Tool

Another way to structure code that can be easily invoked from the command line is with a useful utility program called Rake. Written by Jim Weirich, Rake was initially implemented as a Ruby version of Make, the Unix build utility. However, calling Rake a build utility is to miss its true power. Really, Rake is an automation tool—it's a way of putting all those tasks that you perform in a project into one neat and tidy place.

Rake gives you a convenient way to define small tasks and task dependencies, allowing you to say that a particular task requires a different task to run first. Rake also allows you to automate transitions between files based on file extensions, for example, converting all your `.csv` files to `.json` ...

The Build Environment

When Ruby is compiled for a particular architecture, all the relevant settings used to build it (including the architecture of the machine on which it was compiled, compiler options, source code directory, and so on) are written to the module `RbConfig` within the library file `rbconfig.rb`. After installation, any Ruby program can use this module to get details on how Ruby was compiled:

```
require "rbconfig"
include RbConfig
CONFIG["host"]   # => "arm64-apple-darwin23"
CONFIG["libdir"] # => "/Users/noel/.rbenv/versions/3.3.0-dev/lib"
```

Extension libraries use this configuration file to compile and link properly on any given architecture.

What’s Next

Now that we’ve seen how to run our Ruby files from the command line and use command-line and environment options within Ruby, we’re ready to look at the entire Ruby gems system for packaging tools, managing dependencies, and organizing code.

Footnotes

[20] http://whatisthor.com

[21] http://martinfowler.com/articles/rake.xhtml

Chapter 15
Ruby Gems

One of the tremendous benefits of being a Ruby developer is being able to take advantage of the entire ecosystem of other Ruby developers who have written and shared useful tools with the community. These tools are called *gems*, a term that can refer to the individual tools and the library of code that manages packaging the tools and distributing them. Ruby comes standard with RubyGems, a command-line tool for installing and managing Ruby gems, and Bundler, a tool for creating manifests of gem versions so that developers all use the same set of dependencies.

Not only can you use existing gems, but you can also write your own. Even if you don’t intend on sharing your gem, the basic structure of a Ruby gem is a good skeleton ...

Installing and Managing Gems

All the RubyGems tooling comes standard as a part of Ruby, including the command-line application `gem`, which you can use for many of your RubyGem-related needs. Ruby gems conform to a standardized format that provides metadata about the gems and most importantly about any other gems that this gem might depend on. The `gem` command-line tool knows to look in the central repository for gems, but you can also point it to look at other sources. Using other sources allows you to maintain a private gem repository, for example, as a place to keep internal tooling that you don't want to be made public.

In this section, we'll talk about managing gems directly from the command line, which is useful, but please keep in mind that ...

Using Bundler to Manage Groups of Gems

Gems are pretty great, and any reasonably sized Ruby program will likely depend on many of them, each of which may have its own gem dependencies. This can easily become a management problem in and of itself. How can you have multiple programs on one machine that might use different versions of a gem? Or, because gems change all the time, how can you guarantee that everybody working on the application will be using the same set of gems?

The answer to both of these questions and many more is Bundler.[22] Bundler is itself a Ruby gem, and it manages a manifest file of the gems and versions in use on a project. Bundler allows you to specify the versions in use and limit your Ruby programs to only find the specific ...

Writing and Packaging Your Own Code into Gems

RubyGems isn’t only for downloading gems; you can also write and distribute your own gems. Even if you don’t plan to distribute a gem, the default packaging for RubyGems can help you plan the basic structure of your Ruby code. There used to be multiple sources for how to structure a Ruby gem, but Bundler also provides a default template that has become the basic standard.

As your programs grow (and they all seem to grow over time), you’ll find that you’ll need to start organizing your code. Simply putting everything into a single huge file becomes unworkable and makes it hard to reuse chunks of code in other projects. So, we need to find a way to split our project into multiple files and then knit ...

Organizing Your Source Code

We have two related problems to solve: how do we split our source code into separate files, and where in the file system do we put those files? Some languages, such as Java, make this easy. They dictate that each outer-level class should be in its own file and that file should be named according to the name of the class. Other languages, such as Ruby, have no rules that relate source files and their content. In Ruby, you're free to organize your code as you like.

That said, you'll find that some kind of consistency helps. It'll make it easier for you to navigate your own projects, and it'll also help when you read (or incorporate) other people's code.

The Ruby community has largely adopted a de facto standard. In ...

Distributing and Installing Your Code

There are a few parts of using RubyGems we haven't discussed. We need to consider the metadata that allows the RubyGems ecosystem to know things about our gem. In particular, we want to see how we can inject executables into code that uses our gem and also how our gem declares its dependencies.

RubyGems needs to know information about your project that isn't contained in the directory structure. Instead, you have to write a short RubyGems specification. Our gem creation tool has already created one with most of the boilerplate at `aaagmnr.gemspec`. This comes with some lines marked `TODO` which must be changed before the gem can be used. Here's the completed file, after the `TODO` lines have been changed:

gems/aaagmnr/aaagmnr.gemspec ...

What’s Next

In this chapter, we looked at how to build Ruby programs and package them as gems. Now it’s time to dig a little deeper into working with Ruby itself. In the next chapter, we’ll talk about interacting with Ruby using irb, the interactive Ruby shell.

Footnotes

[22] http://bundler.io

[23] https://github.com/fxn/zeitwerk

[24] https://github.com/testdouble/standard

[25] http://rubygems.org

Chapter 16
Interactive Ruby

If you want to play with Ruby, we recommend Interactive Ruby—irb, for short. irb is a Ruby command-line "shell" similar in concept to an operating system shell (complete with job control). It's sometimes called a REPL (which is an abbreviation for "Read, Evaluate, Print Loop") and provides an environment where you can play around with the language in real time.

You launch irb at the command prompt:

```
$ irb
```

irb displays the value of each expression as you complete it.

```
$ irb
irb(main):001:0* a = 1 +
irb(main):002:0* 2 * 3 /
irb(main):003:0> 4 % 5
=> 2
irb(main):004:0> 2 + 2
=> 4
irb(main):005:0> x = _
=> 4
irb(main):006:0> x
=> 4
irb(main):007:1* def test
irb(main):008:1*   puts "Hello, world!"
```

Using irb

irb is run with this command:

```
irb <irb-options> <ruby_script> <program arguments>
```

The command-line options for irb are listed in the following table. Typically, you'll run irb with no options, but if you want to run a script and watch the step-by-step description as it runs, you can provide the name of the Ruby script and any options for that script.

Table 6. irb command-line options

Option	Description
`--autocomplete`, `--noautocomplete`	Uses or doesn't use the autocomplete feature. The default is `--autocomplete`.
`--back-trace-limit` *n*	Displays backtrace information using the top *n* and last *n* entries. The default value is 16.
`--colorize`, `--nocolorize`	Uses or doesn't use color syntax highlighting. The default is `--colorize`.
`--context-mode ...`	

Navigating irb

From the irb prompt, if you’re not in the middle of typing a multiline expression, pressing the up arrow will move you through your command history, showing the previous command, then the one before it and so on. The down arrow will move you forward in the history once you’ve started backward. The command history is persisted between sessions.

Unlike the way your shell might be set up, the up arrow doesn’t act as a partial search. If you type a character and then press the up arrow, your typing will be replaced by the entire previous command regardless of whether the character you typed is the beginning of the previous command. By default, irb stores 1000 commands in its history, but this amount can be configured.

If you’re in ...

Configuring irb

irb is remarkably configurable. You can set configuration options with command-line options, from within an initialization file, and while you're inside irb itself.

irb uses an initialization file in which you can set commonly used options or execute any required Ruby statements. When irb is run, it'll try to load an initialization file from one of the following sources in order: `~/.irbrc`, `.irbrc`, `irb.rc`, `_irbrc`, and `$irbrc`.

Within the initialization file, you may run any arbitrary Ruby code. For example, you can use `require` for any gem that you might want included in an irb session (such as irbtools[26] or awesome-print[27]).

You can also set configuration values. The list of configuration variables is given in *irb Configuration ...*

What’s Next

In this chapter, we looked at how to use Ruby interactively with irb. Debugging is a common use case for irb. In the next chapter, we’ll dive into the official Ruby debugging tools.

Footnotes

[26] https://irb.tools

[27] https://github.com/awesome-print/awesome_print

Chapter 17
Debugging Ruby

Sometimes code doesn’t work as you expect. Ruby provides different ways for you to see what’s happening as you execute your code, which better enables you in understanding and debugging it. From humble print statements to elaborate inline debugging tools, you can get the visibility you need into your code.

Printing Things

If you want to debug code and you don't want to use any fancy tools, then printing things out to the console is the way to go. We don't mean to make fun—we extensively use this method. It's quick and lends itself to faster cycle times than using a debugger to step through code.

We've seen a few options for this already, like `puts`, which uses `to_s` to convert its argument to a string, and `p`, which does the same thing but uses `inspect`. Ruby also provides `print` and pretty-print via the `pp` method. Here's a table of what they all look like for `nil`, a string, a symbol, an array, a hash, and an array with a hash element:

Table 8. print method

print	p	puts	pp
	nil		nil
test	"test"	test	"test"
test	:test	test	:test
[1, 2, 3]	[1, 2, 3]	1	

The Ruby Debugger

Debuggers can also be quite powerful, and Ruby 3.1 came with a big improvement to Ruby's standard debugger. The new debugger is distributed as a separate Ruby gem rather than part of the standard library:

```
$ gem install debug
```

Note that the gem is named *debug* and not *debugger*. The *debugger* gem is a separate tool that hasn't been maintained since 2015. Similarly, if you're using Bundler, specify a version constraint of `">= 1.0.0"` or else you'll get a different, older gem. Global, permanent namespaces are fun!

After you have installed the debugger gem, there are a few different ways to start the debugger.

You can start your script with the command `rdbg` rather than `ruby`:

```
rdbg code/trouble/profileeg.rb
```

The script ...

Pry

In addition to the official Ruby debugger and the official irb client, there is a third-party alternative called Pry.[29]. Pry predates the Ruby 3.1 debugger and contains many overlapping features, but with a slightly different command-line interface.

To install Pry, you need the Pry gem. Pry also has a plugin system that allows extensions, and we'll be adding the pry-byebug gem that gives step-by-step debugging control to Pry. You can add them to a `Gemfile`:

```
gem "pry"
gem "pry-byebug"
```

The main way to use Pry is by adding the method call `binding.pry` into a code base. If we do that with the same code we used for the debugger earlier, just by replacing `require "debug"` with `require "pry"` and the `binding.break` call with `binding.pry` we ...

Debugging Performance Issues with Benchmark

Ruby is an interpreted, high-level language, and as such it may not perform as fast as a lower-level language such as C. In the following sections, we'll list some basic things you can do to inspect its performance.

Typically, slow-running programs have one or two performance graveyards —places where execution time goes to die. Find and improve them, and suddenly your whole program springs back to life. The trick is finding them —developers are notoriously bad at guessing where performance hangups actually are. The `Benchmark` module can help.

You can use the `Benchmark` module to time sections of code. For example, we may wonder what the overhead of method invocation is. You can use `Benchmark.bm` or `Benchmark.bmbm ...`

What’s Next

In this chapter, we talked about different ways to debug Ruby, including printing information to the console, using the official Ruby debugger and the popular third-party program, Pry. We also looked at how to use Ruby’s benchmarking tools to identify slow spots in your code.

Type errors are a common source of bugs in Ruby, and there are some attempts to allow developers to add type information to their Ruby code so that tooling can identify type problems before they can become bugs. Let’s check this out.

Footnotes

[28] https://marketplace.visualstudio.com/items?itemName=KoichiSasada.vscode-rdbg

[29] http://pry.github.io

Chapter 18
Typed Ruby

When you see a variable in your code, it's useful to know what values can be assigned to that variable without the code breaking. For example, you might have the following Ruby method:

```
def mystery_method(x)
  x * 3
end
```

You'd likely expect that `x` should be a number. But it's also completely valid Ruby for `x` to be a string (`"a" * 3` resolves to "aaa") or an array (`[:a] * 3` resolves to `[:a, :a, :a]`).

Let's say that this method is in your code, and over time people call `mystery_method` with strings, integers, floating-point numbers, and so on, until somebody changes the method and inadvertently changes what variables it'll accept. Here's an example:

```
def mystery_method(x)
  x.abs * 3
end
```

Now all of those string ...

What’s a Type?

The terminology around types in programming languages can be confusing because each language community uses the terms slightly differently.

Most generally, setting the *type* of a variable, attribute, or method argument limits the set of values that can be assigned to that variable, attribute, or method argument. The type also determines the behavior of the variable within the program. For example, the result of `x/y` depends on the type of `x` and `y`. In many languages, the result will be different if the numbers are integers than if they are floating-point types.

Many programming languages have a set of “basic types” that can be used, often including strings, boolean values, different kinds of numerical values, and so on. Ruby doesn’t ...

Official Ruby Typing with RBS

The official Ruby typing system is called RBS (short for Ruby Signature). With RBS, you create a separate file that contains type signature information for all or part of your code.

Writing RBS

To take a look at how RBS works, we'll use the gem we created in *Writing and Packaging Your Own Code into Gems*, and augment it with RBS typing. If you look at the Aaagmnr gem code, you'll see that it contains a directory named `sig` that we didn't talk much about. That directory is where you're supposed to put the type information, and right now it contains one file:

gems/aaagmnr/sig/aaagmnr.rbs

```
module Aaagmnr
  VERSION: String
  # See the writing guide of rbs: https://github.com/ruby/rbs#guides
end
```

The only thing this file ...

Ruby Typing with Sorbet

You may have noticed that while RBS is an interesting way to add type hints to your application, the actual usage of it is still a little light. A third-party tool called Sorbet that's managed by a team at Stripe also provides type checking. At the moment Sorbet has more powerful analysis tools.

As with some other third-party tools, we're covering Sorbet here because there's a good chance you'll see it out in the world, but third-party tools often change quickly, so we recommend https://sorbet.org/docs for the full scoop of Sorbet, so to speak.

Sorbet is different from RBS in that the type annotations generally go in the Ruby file, though there is also an external file format. The type annotations are plain Ruby, and ...

What’s Next

To be honest, we’re a little conflicted about types in Ruby. We like the potential performance benefit, and the tooling advantages are promising. There’s definitely a communication benefit to being explicit with types. But we worry that some of Ruby’s dynamic power and flexibility is being traded for static typing, and for developers who came to Ruby for that flexibility, that can be a hard trade-off.

Some of that communication benefit can also come from documentation. In the next chapter, we’ll discuss RDoc, the official Ruby documentation solution, and YARD, a commonly used third-party extension.

Chapter 19
Documenting Ruby

Documentation is a critical part of communicating across teams. Code comments can help share the intent of the developer or can be a way to explain constraints on the code that might not be clear from just reading code. It's not enough to add comments to the code; it's also useful to be able to publish those comments onto the web and to make them consumable by your editor or command-line tools like `ri` or `irb`.

Two tools in the Ruby ecosystem are used for converting code comments into external documentation: RDoc and YARD. Ruby comes bundled with RDoc, which is used by Ruby itself to document the built-in Ruby classes and modules. Those who like a more formal, tag-based scheme might want to look at YARD (http://yardoc.org ...

Documenting with RDoc

RDoc does two jobs. Its first job is to analyze source files. Ruby files, of course, but it will also analyze C files and Markdown files. Within those files, RDoc looks for information to document. Its second job is to take this information and convert it into something readable—usually HTML or Ruby's `ri` documentation format.

Let's look at an example.

rdoc/example/counter.rb

```
class Counter
  attr_reader :counter

  def initialize(initial_value = 0)
    @counter = initial_value
  end

  def inc
    @counter += 1
  end
end
```

Going into that directory and running `rdoc` will create an entire `doc` directory with HTML files.

Here's what one of the files looks like:

Even though the source contains no internal documentation, ...

Adding RDoc to Ruby Code

RDoc parses Ruby source files to extract the major elements (such as classes, modules, methods, attributes, and so on). You can choose to associate additional documentation with these by simply adding a comment block before the element in the file.

One of the design goals of RDoc was to leave the source code looking totally natural. In most cases, you don’t need any special markup in your code to get RDoc to produce decent looking documentation. For example, comment blocks can be written fairly naturally:

```
# Calculate the minimal-cost path though the graph using Debrinkski's
algorithm,
# with optimized inverse pruning of isolated leaf nodes.
def calculate_path
  . . .
end
```

You can also use Ruby’s block-comments ...

Running RDoc

RDoc can be run from the command line, like this:

```
$ rdoc OPTIONS FILENAMES
```

Type `rdoc --help` for an up-to-date option summary.

Files are parsed, and the information they contain is collected before any output is produced. This allows cross-references between all files to be resolved. If a name is that of a directory, it is traversed. If no names are specified, all Ruby files in the current directory (and subdirectories) are processed.

A typical use may be to generate documentation for a package of Ruby source (such as RDoc itself):

```
$ rdoc
```

This command generates HTML documentation for the files in and below the current directory. These will be stored in a documentation tree starting in the subdirectory `doc/`.

Documenting with YARD

YARD (http://yardoc.org) is an extension of RDoc that uses tags to allow you to add metadata to your comments. It then uses the metadata to create more interesting documentation.

Yet Another Aside

While YARD claims to be short for “Yay! A Ruby Documentation Tool,” it likely was named in reference to the long-standing open-source tradition of starting acronyms with YA for “yet another.” We think the first tool to this was the Yacc parser-generator (for "Yet Another Compiler Compiler"), but there’s also YAML (originally "Yet Another Markup Language," now styled as "YAML Ain’t Markup Language," which is a whole other open-source ...

What’s Next

In this chapter, we talked about the two most commonly used tools for documenting Ruby: RDoc and YARD. If you look around the web at documentation for Ruby gems, you’ll see both of their distinctive HTML styles in regular use. Now let’s take a bit of a turn and talk about Ruby’s tools for working with the web itself.

Part 3Ruby Crystallized

Ruby is a sophisticated and flexible object-oriented language, and using it effectively can mean exploring reflection, metaprogramming, and other related ideas.

Chapter 20
Ruby and the Web

There’s a good chance that if you’re reading this book, you’re intending to use Ruby in the context of some kind of Web application. Ruby is used as the language for a lot of web tools, not only for Ruby on Rails,[30] but also for many other web frameworks and third-party tools.

While the Ruby ecosystem is full of web tools, most of those tools are third-party gems and not part of the core Ruby distribution. Core Ruby does provide an implementation of the Common Gateway Interface (CGI), which was the original dynamic web standard, but as we write this, you’re unlikely to be writing CGI scripts directly, so we’re not going to spend time on CGI scripts in this book. That said, for historical reasons, a lot of Ruby’s ...

Ruby's Web Utilities

The Ruby standard distribution includes some core utilities as part of the `CGI` class and `CGI::Util` module.

CGI Encoding

When dealing with URLs and HTML code, you must be careful to quote certain characters. For instance, a slash character (`/`) has special meaning in a URL, so it must be "escaped" if it's not part of the path name. That is, any `/` in the query portion of the URL is translated to the string `%2F`, and then it must be translated back to a `/` for you to use it. Space and ampersand are also special characters.

To handle this, `CGI` provides the methods `escape` and `unescape`:

```
require "cgi"
puts CGI.escape("Nicholas Payton/Trumpet & Flugel Horn")
```

Produces:

```
Nicholas+Payton%2FTrumpet+%26+Flugel+Horn
```

More frequently, ...

Templating with ERB

Templating systems let you separate the presentation and logic of your application.

So far, we've looked at using Ruby to create HTML output, but we can turn the problem inside out: we can actually embed Ruby in an HTML document. The embedded Ruby (or ERB) library is included with Ruby's standard distribution.

Embedding Ruby in HTML is a powerful concept—it gives us the equivalent of a scripting tool such as PHP, but with the full power of Ruby.

Using ERB

ERB is a filter. Input text is passed through untouched, with the following exceptions:

Expression	Description
<% *ruby code* %>	This executes the Ruby code between the delimiters. Any resulting value isn't sent to the output.
<%= *ruby expression* %>	This evaluates the Ruby expression ...

Serving Ruby Code to the Web

Ruby is commonly used as the back end of a web application. In this pattern, a request is made to a web server, which executes Ruby code. The Ruby code returns HTML (usually), which is sent back to the web browser as the response.

While you could write all this in plain Ruby yourself, there's no need for you to do all that work. The Ruby ecosystem has multiple web servers, including Puma,[32] Unicorn,[33] Thin,[34] and Falcon.[35] You can also use multiple web frameworks, including Ruby on Rails,[36] Sinatra,[37] Roda,[38] and Hanami.[39]

You might look at that incomplete list of web servers and frameworks and think that compatibility between the two sets might be a nightmare of continually having to adjust the ...

Ruby in the Browser with Web Assembly

Web Assembly (Wasm) is a virtual machine runtime engine that runs in a web browser, allowing any Wasm-compliant programming language to be used as a scripting language in that browser.

Ruby 3.2 added support for Wasm as a compilation target of Ruby. You can see full instructions for creating your own Wasm build at https://github.com/ruby/ruby/blob/master/wasm/README.md, but in most cases you'll likely either pull in the Wasm build as a module in Node Package Manager (NPM) or point to a source for the Wasm build in your web page.

So, you can do this in an HTML file:

web/wasm.xhtml

```
<html>
<script
  src="https://cdn.jsdelivr.net/npm/ruby-head-wasm-
wasi@0.5.0/dist/browser.script.iife.js"
></script>
<script ...
```

What's Next

We've covered many different ways Ruby interacts with the web. We've seen Ruby's own utilities, the Rack framework for interactions between web servers and application frameworks, and the Sinatra framework for building basic web applications. Next we'll look at Ruby style.

Footnotes

[30] http://www.rubyonrails.org

[31] https://docs.ruby-lang.org/en/master/ERB.xhtml

[32] https://puma.io

[33] https://yhbt.net/unicorn

[34] https://github.com/macournoyer/thin

[35] https://github.com/socketry/falcon

[36] https://rubyonrails.org

[37] https://sinatrarb.com

[38] https://github.com/jeremyevans/roda

[39] https://hanamirb.org

[40] https://github.com/rack/rack

[41] https://github.com/rack/rack/blob/main/SPEC.rdoc

[42] https://sinatrarb.com

Chapter 21
Ruby Style

As you become familiar with Ruby and the Ruby community, you will see references to “Ruby style” or “idiomatic Ruby.” These terms refer to the way in which developers who love Ruby tend to write Ruby code. Ruby provides a broad range of allowed syntax and different ways of performing similar tasks and interacting with existing code. Ruby is such a flexible language that style and community standards of practice go a long way toward keeping Ruby code readable from project to project.

There are two distinct kinds of Ruby style. The first is the kind that’s syntax-based, governs how you write individual lines and small blocks of code, and can (to some extent) be evaluated by a linting program. In this chapter, we’ll discuss ...

Written Ruby Style

The goal of having a written coding style is to make the intent and functionality of the code clear and easy to maintain. When the physical layout of the file matches its logical layout and the constructs are presented consistently, it’s easier for a reader of the code to understand what the code is doing.

Programming languages tend to offer flexibility in how the code is actually laid out in the file as written by the developer. Some languages are stricter in arranging the physical layout of their syntax. For example, Python is known for enforcing the use of indentation to denote logical blocks of code. Ruby, for better or worse, isn’t like that. Ruby offers the developer tremendous flexibility.

In the face of that flexibility, ...

Using RuboCop

When you write Ruby code, you often don't want to have to think about style. What you want to do is write your code and solve your problem. But what usually happens is that once you have your team review your code, people on your team start to nitpick your minor style choices, sometimes contradicting each other. This is, to say the least, not compatible with great team morale.

Enter RuboCop.[43] RuboCop is a *linter*, which means that it automatically checks your code for style and then either flags discrepancies or optionally autocorrects them. (The name "linter" comes from the first such program, `lint`, written in 1978 by Stephen C. Johnson for C code and named by analogy to a lint trap as a thing that catches small issues.)

Using Standard

RuboCop is powerful, but also complicated. If we're being honest, we don't like all the defaults. As a result, our typical project winds up with a long configuration file that's a bit unwieldy and continually subject to argument. If only somebody would come up with a more consensus-based Ruby style-checker.

Enter Standard,[46] a default configuration of RuboCop created by Justin Searls that is minimally configurable and conforms to a small, but commonly used, set of rules.

What rules? You can see the entire list in the Standard RuboCop's setup, but the gist is similar to the common rules we set out in the first part of this chapter:

- Two-space indentation
- Double-quoted strings
- No hash rockets for symbol hash keys
- Spaces inside ...

Ruby Style in the Large

Ruby style isn’t just a matter of how you code line-by-line, it also manifests in how you approach problems and how you build solutions. The goal of all of these style recommendations is to allow the code to clearly reflect the intent of the programmer. Taking advantage of what Ruby does best and most simply will make it easier to read and modify your code going forward.

Ruby code is often written with shorter methods and smaller classes than more strongly typed languages. Not only do smaller methods give you more chances to give blocks of code meaningful names, but also having small pieces of functionality made into methods makes it easier to build functionality by combining methods using Ruby syntax such as blocks. ...

Duck Typing

You may have noticed that in Ruby you don’t explicitly declare the types of variables or methods. Whether the particular value of a variable is correct for the messages being passed to it’s evaluated at run time when the message is sent.

Folks tend to react to this in one of two ways. Some like this flexibility and feel comfortable writing code with dynamically typed variables and methods. Others get nervous when they think about all those objects floating around unconstrained. If you’ve come to Ruby from a language such as C#, Java, or TypeScript, where you’re used to giving all your variables and methods an explicit type, you may feel that Ruby is just too permissive for writing “real” applications.

It isn’t.

We’d like to spend ...

What’s Next

In this chapter, we talked about Ruby style in terms of the decisions you make both when writing individual lines and also when writing methods and classes.

But we’ve only talked about part of Ruby’s dynamic toolkit. Ruby has a rich set of options that make metaprogramming easier. These are often referred to as “magic,” so let’s take a look behind the curtain and see how the magic is done.

Footnotes

[43] https://rubocop.org

[44] https://docs.rubocop.org/rubocop/1.38/index.xhtml

[45] https://docs.rubocop.org

[46] https://github.com/testdouble/standard

Chapter 22
The Ruby Object Model and Metaprogramming

The Jacquard loom, invented more than 200 years ago, was the first device controlled by punched cards—rows of holes in each card were used to control the pattern woven into the cloth. But imagine if instead of churning out fabric, the loom could punch more cards, and those cards could be fed back into the mechanism. The machine could be used to create new programming that it could then execute. And that would be metaprogramming—writing code that writes code.

Programming is all about building layers of abstractions. As you solve problems, you're building bridges from the unrelenting and mechanical world of silicon to the more ambiguous and fluid world we inhabit. Some programming languages—such ...

Understanding Objects and Classes

Classes and objects are central to Ruby, but at first sight, they can be confusing. It seems like there are a lot of concepts: classes, objects, class objects, instance methods, class methods, singleton classes, and virtual classes. In reality, all these Ruby constructs are part of the same underlying class and object structure.

Internally, a Ruby object has three components: a set of flags, some instance variables, and an associated class. A Ruby class contains all the things an object has plus a set of method definitions and a reference to a superclass (which is itself another class). A Ruby class is itself an instance of the class `Class`. Let’s look at how that structure lends itself to metaprogramming in ...

Defining Singleton Methods

Ruby lets you define methods that are specific to a particular object. These are called *singleton methods*.

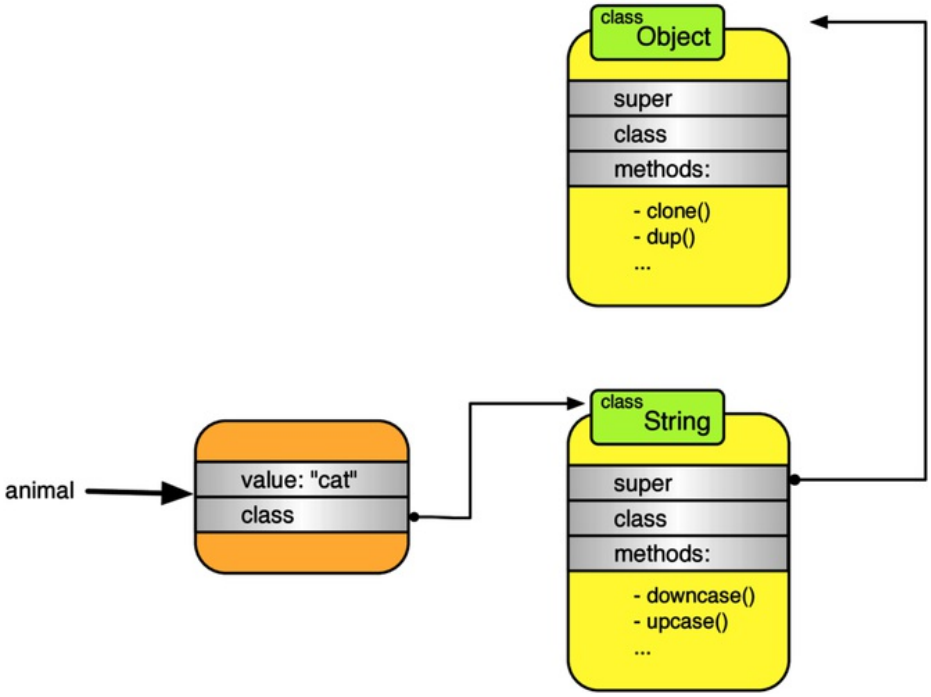
Here's a simple string object and a regular, non-singleton method call:

```
animal = "cat"
puts animal.upcase
```

Produces:

```
CAT
```

This call results in the object structure shown in the following illustration:



The `animal` variable points to an object containing (among other things) the value of the string (`"cat"`) and a pointer to the object's class, `String`.

When we call `animal.upcase`, Ruby checks the object referenced by the `animal` variable and then looks up the method `upcase` in the class object referenced

from the ...

Inheritance and Visibility

Method definition and class inheritance have a wrinkle, but it's fairly obscure. Within a class definition, you can change the visibility of a method in an ancestor class. For example, you can do something like this:

```
class Base
  def a_method
    puts "Got here"
  end
  private :a_method
end

class MakeItPublic < Base
  public :a_method
end

class KeepItPrivate < Base
end
```

In this example, you could invoke `a_method` in instances of class `MakeItPublic`, but not via instances of `Base` or `KeepItPrivate`.

So, how does Ruby pull off this feat of having one method with two different visibilities? Simply put, it cheats.

If a subclass changes the visibility of a method in a parent, Ruby effectively ...

Modules and Mixins

As we saw in *Mixins*, when you include a module into a Ruby class, the instance methods in that module become available as instance methods of the class, like this:

```
module Logger
  def log(msg)
    STDERR.puts Time.now.strftime("%H:%M:%S: ") + "#{self} (#{msg})"
  end
end

class Song
  include Logger
end

s = Song.new
s.log("created")
```

Produces:

```
17:16:19: #<Song:0x0000000104717708> (created)
```

Ruby implements `include` very simply. The module you include is added as an ancestor of the class being defined. It's as if the module is the parent of the class that it's mixed into. And that would be the end of the description except for one small wrinkle. Because the module is injected ...

Metaprogramming Class-Level Macros

If you've used Ruby for any time at all, you're likely to have used `attr_accessor`, the method that defines reader and writer methods for instance variables:

```
class Song
  attr_accessor :duration
end
```

If you've written a Ruby on Rails application, you've probably used `has_many`:

```
class Album < ActiveRecord::Base
  has_many :tracks
end
```

These are both examples of class-level methods that generate code behind the scenes. Because of the way they expand into something bigger, folks sometimes call these kinds of methods *macros*.

Let's create a trivial example and then build it up into something realistic. We'll start by implementing a simple method that adds logging capabilities to instances of a class. ...

Using instance_eval and class_eval

No matter where you are in a Ruby program, `self` always has a value determined by your location in the code. Sometimes it's useful to be able to manage that relationship and change the value of `self` for a while.

The methods `Module#instance_eval`, `Module#class_eval`, and `Module#module_eval` let you set `self` to be an arbitrary object, evaluate the code in a block with that object as `self`, and then reset `self`:

```
"cat".instance_eval do
  puts "Upper case = #{upcase}"
  puts "Length is #{self.length}"
end
```

Produces:

```
Upper case = CAT
Length is 3
```

Inside the `instance_eval` block, the variable `self` is temporarily given the value of the object that received the `instance_eval` message.

All the

Using Hook Methods

In *Class Methods and Modules*, we defined a method called `included` in our `GeneralLogger` module. When this module was included in a class, Ruby automatically invoked this `included` method, allowing our module to add class methods to the host class.

`included` is an example of a *hook method* (sometimes called a *callback*). A hook method is a method that you write but that Ruby calls from within the interpreter when some particular event occurs. The interpreter looks for these methods by name. If you define a method in the right context with an appropriate name, Ruby will call it when the corresponding event happens.

The methods that can be invoked from within the interpreter are:

Method-related hooks

: `method_added`, `method_missing` ...

A Metaprogramming Example

Let’s bring together all the metaprogramming topics we’ve discussed in a final example by writing a module that allows us to trace the execution of methods in any class that mixes the module in. This would let us write the following:

metaprogramming/trace_calls_example.rb

```
require_relative "trace_calls"

class Example
  def one(arg)
    puts "One called with #{arg}"
  end
end

ex1 = Example.new
ex1.one("Hello") # no tracing from this call

class Example
  include TraceCalls
  def two(arg1, arg2)
    arg1 + arg2
  end
end

ex1.one("Goodbye") # but we see tracing from these two
puts ex1.two(4, 5)
```

Produces:

```
One called with Hello
==> calling one with ["Goodbye"]
```

Top-Level Execution Environment

Finally, there's one small detail we have to cover to complete the metaprogramming environment. Many times in this book we've claimed that everything in Ruby is an object. But we've used one thing time and time again that appears to contradict this—the top-level Ruby execution environment:

```
puts "Hello, World"
```

Not an object in sight. We may as well be writing some variant of Fortran or Basic. But dig deeper, and you'll come across objects and classes lurking in even the simplest code.

We know that the literal `"Hello, World"` generates a Ruby `String`, so that's one object. We also know that the bare method call to `puts` is effectively the same as `self.puts`. But what is `self`?

```
self       # => main
self.class # => ...
```

What’s Next

Metaprogramming is one of Ruby’s sharpest tools. Don’t be afraid to use it to raise up the level at which you program. But, at the same time, use it only when necessary—overly metaprogrammed applications can become pretty obscure pretty quickly.

There’s one more piece of the Ruby metaprogramming puzzle: reflection, which is how Ruby knows things about the runtime environment. Let’s take a look.

Chapter 23
Reflection and Object Space

One of the advantages of dynamic languages such as Ruby is the ability to *introspect*—to examine aspects of a program from within the program itself. This is also called *reflection*.

When people introspect, they think about their thoughts and feelings. This is interesting because we're using thought to analyze thought. It's the same when programs use introspection—a program can discover the following information about itself:

- What objects it contains
- Its class hierarchy
- The attributes and methods of objects
- Information on methods

Armed with this information, we can look at particular objects and decide which of their methods to call at runtime—even if the class of the object didn't exist when we first wrote ...

Looking at Objects

Have you ever craved the ability to traverse all the living objects in your program? We have! Ruby has a global object called `ObjectSpace` that lets you do some fun tricks with the set of objects Ruby is tracking (this means all the objects that have been created and not yet destroyed by garbage collection).

What Is Garbage Collection?

Ruby is a dynamic language, and it doesn’t require the programmer to manage the memory that the program uses during runtime. Instead, Ruby uses a process called garbage collection. Garbage collection looks for objects that have been allocated into memory but are no longer in scope or are otherwise ...

Looking at Classes

Knowing about objects is one aspect of reflection, but to get the whole picture, you also need to be able to look at classes—and the methods and constants that they contain.

Looking at the class hierarchy is easy. You can get the parent of any particular class using `Class#superclass` and its children using `Class#subclasses`. For classes *and* modules, the `Module#ancestors` method lists both superclasses and mixed-in modules:

ospace/relatives.rb

```
klass = Integer
print "Inheritance chain: "
begin
  print klass
  klass = klass.superclass
  print " < " if klass
end while klass
puts
p "Ancestors: #{Integer.ancestors}"
p "Subclasses: #{Integer.subclasses}"
```

Produces:

```
Inheritance chain: Integer < ...
```

Calling Methods Dynamically

The `Object#send` method lets you tell any object to invoke a method by name. The first argument is a symbol or a string representing the name, and any remaining arguments are passed along to the method of that name. Let's have a look at the following code:

```
"John Coltrane".send(:length)          # => 13
"Miles Davis".send("sub", /iles/, '.') # => "M. Davis"
```

There are two twists to `send`. First, your class might define its own `send` method if it wanted to send something somewhere. Ruby provides the `__send__` method, defined in `BasicObject`, which is identical to `send` and is meant to be used in cases where `send` might have been overwritten .

Second, the `send` method doesn't enforce method access, meaning that ...

System Hooks

A *hook* is a technique that lets you trap some Ruby event, such as object creation. Let's take a look at some common Ruby hook techniques.

Intercepting Method Calls

The simplest hook technique in Ruby is to intercept calls to methods in core classes. Perhaps you want to log all the operating system commands your program executes. You could simply use `alias_method` to rename the `system` method and replace it with a `system` method of your own that both logs the command and calls the original `Kernel#system` method.

For example:

```
class Object
  alias_method :old_system, :system

  def system(*args)
    old_system(*args).tap do |result|
      puts "system(#{args.join(', ')}) returned #{result.inspect}"
```

Tracing Your Program's Execution

While we're having fun reflecting on all the objects and classes in our programs, let's not forget about the humble expressions that make our code actually do things. It turns out that Ruby lets us look at these expressions, too.

First, you can watch the interpreter as it executes code, using the `TracePoint` class. `TracePoint` is used to execute a proc while adding all sorts of juicy debugging information whenever a new source line is executed, methods are called, objects are created, and so on.

Here's a bit of what `TracePoint` can do (run this in your own irb window, but be aware that it'll produce a *lot* of output):

```
class Test
  def test
    a = 1
  end
end

TracePoint.trace(:line) do |trace_point| ...
```

Behind the Curtain: The Ruby VM

If you'd like to know what Ruby is doing with all that code you're writing, you can ask the Ruby interpreter to show you the intermediate code that it's executing.

You can ask it to compile the Ruby code in a string or in a file and then disassemble it and even run it. You might wonder if it can dump the opcodes out and later reload them. The answer is no—the interpreter has the code to do this, but it's disabled because there isn't yet an intermediate code verifier for the Ruby interpreter.

Here's a trivial example of disassembly:

```
code = RubyVM::InstructionSequence.compile('a = 1; puts 1 + a')
puts code.disassemble
```

Produces:

```
== disasm: #<ISeq:<compiled>@<compiled>:1 (1,0)-(1,17)>
local table (size: ...
```

Marshaling and Distributed Ruby

Ruby features the ability to *serialize* objects, letting you store them somewhere and reconstitute them when needed. You can use this facility, for instance, to save a tree of objects that represent some portion of the application state—a document, a CAD drawing, a piece of music, and so on.

Ruby calls this kind of serialization *marshaling* (think of railroad marshaling yards where individual cars are assembled in sequence into a complete train, which is then dispatched somewhere). Saving an object and some or all of its components is done using the method `dump`. Typically, you'll dump an entire object tree starting with some given object. Later, you can reconstitute the object using `load`.

Here's a short example. ...

What’s Next

The important thing to remember about Ruby is that there’s no big difference between “compile time” and “runtime.” It’s all the same. You can add code to a running process. You can redefine methods on the fly, change their scope from public to private, and so on. You can even alter basic types, such as `Class` and `Object`. Once you get used to this flexibility, it’s hard to go back to a static language such as C++ or even to a half-static language such as Java.

But then, why would you want to do that?

Now it’s time to take a look at Ruby’s syntax in a more structured way.

Footnotes

[47] http://www.yaml.org

[48] https://www.json.org/json-en.xhtml

Part 4Ruby Language Reference

This part looks at the Ruby language from the bottom up. Most of what appears here is the syntax and semantics of the Ruby language. The extensive library of classes and modules will mostly be covered in Part V. However, Ruby's syntax and the library are closely entangled—literal values are part of syntax, but they create objects in the library, so we'll cover parts of the library as needed to explain the syntax.

Chapter 24
Language Reference: Literal Types and Expressions

So far, we've given a narrative look at how Ruby works. In this chapter, we're going to follow more of a reference structure to discuss Ruby's syntax as it concerns literal types and expressions. In the next chapter, we'll cover syntax relating to objects and classes.

Source Layout

Ruby is a line-oriented language. Ruby expressions and statements are terminated at the end of a line unless the parser can determine that the statement is incomplete. Some examples are when the last token on a line is an operator or comma, or when an open delimiter, such as a parenthesis, square bracket, or curly brace has not been closed. A backslash at the end of a line also tells Ruby to continue the expression onto the next line:

```
# no backslash '\' needed -- ends with an operator
d = 4 + 5 +
    6 + 7

# no backslash '\' needed -- has an unclosed parenthesis
e = (4 + 5
     + 6 + 7)

# backslash '\' needed -- ends with a number
f = 8 + 9   \
    + 10
```

A semicolon can be used to separate multiple expressions on a line: ...

Ruby Literals

Ruby has special syntax for scalar values that are booleans, numbers, lambdas, ranges, regular expressions, strings, symbols, and array and hash collections. Although all types in Ruby are implemented as classes, these types have special syntax for creating literal values of them. Lambda literals will be covered in *Proc Objects*; we're going to discuss the rest here.

Boolean Literals

Ruby provides the literal values `true` and `false`. The `true` value is the instance of the singleton class `TrueClass` and represents a true value in logical expressions. The `false` value is the instance of the singleton class `FalseClass` and represents a false value in logical expressions. The literal value `nil` is the only instance of the singleton `NilClass` ...

Regular Expressions

Regular expression literals are objects of type `Regexp`. They are created explicitly by calling `Regexp.new` or implicitly by using the literal forms, `/_pattern_/` and `%r{_pattern_}`. The `%r` construct is a form of general delimited input as described in *General Delimited Input*.

```
/pattern/ /pattern/options %r{pattern} %r{pattern}options
Regexp.new("pattern" <, options>)
```

`options` is one or more of `i` (case insensitive), `o` (substitute once), `m` (matches newline), and `x` (allow spaces and comments). You can additionally override the default encoding of the pattern with `n` (no encoding-ASCII), `e` (EUC), `s` (Shift_JIS), or `u` (UTF-8).

Within a regular expression, each entry in the following table matches the characters described in its description. ...

Names

Ruby names are used to refer to constants, variables, methods, classes, and modules. The first character of a name helps Ruby determine its intended use. Certain names, listed in the following table, are keywords and shouldn't be used as variable, method, class, or module names. (Technically, many of these names are legal method or variable names; it's just very confusing to use them in that way.) Method names are described later in *Method Definition*.

Table 18. Reserved words

__ENCODING__	__FILE__	__LINE__	BEGIN	END	alias	and	begin	
break	case	class	def	defined?	do	else	elsif	end
ensure	false	for	if	in	module	next	nil	not
or	redo	rescue	retry	return	self	super	then	true
undef	unless	until	when	while	yield			

In these descriptions, an *uppercase letter* is a capital letter from any ...

Values, Variables, and Constants

Ruby variables and constants hold references to objects. Variables themselves don't have an intrinsic type. Instead, the type of a variable is defined solely by the messages to which the object referenced by the variable responds. (When we say that a variable is not typed, we mean that any given variable can at different times hold references to objects of different types.)

A Ruby *constant* is also a reference to an object. Constants are created when they are first assigned to (normally in a class or module definition). Ruby, unlike other less flexible languages, lets you alter the value of a constant, although this will generate a warning message, which gets sent to `$stderr`:

```
MY_CONST = 1
puts "First MY_CONST ...
```

Expressions, Conditionals, and Loops

Single terms in an expression may be any of the following:

- *Literal*. Ruby literals are boolean, numbers, strings, arrays, hashes, ranges, symbols, and regular expressions. These are described in *Ruby Literals*.

- *Shell command*. A shell command is a string enclosed in backquotes or in a general delimited string starting with `%x`. The string is executed using the host operating system's standard shell, and the resulting standard output stream is returned as the value of the expression. The execution also sets the `$?` variable with the command's exit status:

```
filter = "*.c"
files = `ls #{filter}`
files = %x{ls #{filter}}
```

- *Variable reference* or *constant reference*. A variable is referenced by citing ...

Chapter 25
Language Reference: Objects and Classes

In this second chapter of language reference, we'll cover the syntax of Ruby objects and classes.

Method Definition

> def *defname* <(param) > *body* end def *defname*
>
> <(param)> = expression *defname* ← *methodname* | *expr.methodname*

The *defname* contains the name of the method and optionally an expression defining the context in which the method is valid, the most common expression here is `self`, as in `def self.method_name`, but the expression can be any Ruby object.

A *methodname* is either a redefinable operator (see Table 19, *Ruby operators (high to low precedence)*) or a name. If *methodname* is a name, it starts with a letter or underscore optionally followed by uppercase and lowercase letters, underscores, and digits. A method can start with an uppercase letter, but that's normally only done for the conversion methods in `Kernel`. A *methodname* ...

Invoking a Method

> <receiver.>*name* < arguments > < {block} >
>
> <receiver::>*name* < arguments > < {block} > *arguments* ←
>
> (<arg>* <, hashlist> <*array> <&a_proc>) *block* ← {
> *blockbody* } *or* do *blockbody* end

When invoking a method, the parentheses around the arguments may be omitted if the expression is otherwise unambiguous. So, `foo 3, 4` as a line by itself is a legal call to `foo` with two arguments. Usually, the ambiguity happens if there are method calls in the arguments. If you had both `foo` and `bar` as methods, then `foo 3, bar 4, 5` would trigger an error because the parser would attempt to resolve `bar 4` as the second argument to `foo`. Fully parenthesizing as `foo(3, bar(4, 5))` is preferred, but `foo 3, (bar 4, 5)` would also be legal.

Aliasing

> alias *new_name old_name*

This creates a new name that refers to an existing method, operator, global variable, or regular expression backreference (`$&`, `$“`, `$'`, and `$+`). Local variables, instance variables, class variables, and constants may not be aliased. The parameters to `alias` may be names or symbols.

object_ref/alias_1.rb

```
class Integer
  alias plus +
end
1.plus(3)        # => 4

alias $prematch $`
"string" =~ /i/  # => 3
$prematch        # => "str"

alias :cmd :`
cmd "date"       # => "Thu Nov  2 17:16:30 CDT 2023\n"
```

When a method is aliased, the new name refers to a copy of the original method’s body. If the original method is subsequently redefined, the aliased name will still invoke the original implementation. ...

Defining Classes

> class <scope::> *classname* << superexpr> *body* end class << *obj* *body* end

A Ruby class definition creates or extends an object of class **Class** by executing the code in **body**. In the first form, a named class is created or extended. The resulting **Class** object is assigned to a constant named **classname** (keep reading for scoping rules). This name should start with an uppercase letter. In the second form, an anonymous (singleton) class is associated with the specific object.

If present, **superexpr** should be an expression that evaluates to a **Class** object that will be the superclass of the class being defined. If omitted, the superclass defaults to class **Object**.

Within **body**, most Ruby expressions are executed as the definition ...

Defining Modules

```
module name     body end
```

A module is a collection of behaviors that are grouped together. The grouping can be used to include all the behaviors in a different module or class as one unit, or it can be used for name spacing. Like a class, the module body is executed during definition, and the resulting `Module` object is stored in a constant. A module may contain both class and instance methods and may define constants and class variables. (A module can also reference instance variables, but those are dependent on the module being added to a class; see the next section.)

As with classes, a module's own methods (called *module methods*) are invoked using the `Module` object as a receiver, and constants are accessed using the `::` scope ...

Access Control

```
private <symbol>* protected <symbol>* public <symbol>*
```

Ruby defines three levels of access protection for module and class constants and methods:

- *Public*—Accessible to anyone.
- *Protected*—Can be invoked only by objects of the defining class and its subclasses.
- *Private*—Can be called only with `self` as the receiver, including both implicit and explicit uses of `self`. Private methods therefore can be called in the defining class and by that class's descendants and ancestors, but only within the same object. (See *Specifying Access Control* for examples.)

These levels are invoked with the methods `public`, `protected`, and `private`, which are defined in class `Module`. Each access control function can be used in three different ways:

- If used ...

Blocks, Closures, and Proc Objects

A code block is a set of Ruby statements and expressions inside braces or a `do`/`end` pair. The block may start with an argument list between vertical bars. A code block may appear only immediately after a method invocation. The start of the block (the brace or the `do`) must be on the same logical source line as the end of the invocation:

> *invocation* do | a1, a2, ... | end
> *invocation* { | a1, a2, ... | }

Braces have a high precedence; `do` has a low precedence. If the method invocation has parameters that aren't enclosed in parentheses, the brace form of a block will bind to the last parameter, not to the overall invocation. The `do` form will bind to the entire invocation.

Within the body of the invoked method, the ...

Exceptions

Ruby exceptions are objects of class `Exception` and its descendants.

Raising Exceptions

The `raise` method raises an exception:

> raise raise *string cause: $!* raise *thing* <, string stack tracecause: $!>

When an exception is raised, Ruby places a reference to the `Exception` object in the global variable `$!`. The first form reraises the exception in `$!` or creates a new `RuntimeError` if `$!` is `nil`. The second form creates a new `RuntimeError` exception, setting its message to the given string. The third form creates an exception object by invoking the method `exception` on its first argument, setting this exception's message and backtrace to its second and third arguments. Class `Exception` and objects of class `Exception` contain a factory method called ...

Catch and Throw

The method `catch` executes its associated block:

```
catch ( object ) do     code... end
```

The method `throw` interrupts the normal processing of statements:

```
throw( object <, obj> )
```

When a `throw` is executed, Ruby searches up the call stack for the first `catch` block with a matching `object`. If it's found, the search stops, and execution resumes past the end of the `catch`'s block. If the `throw` is passed a second parameter, that value is returned as the value of the `catch`. Ruby honors the `ensure` clauses of any block expressions it traverses while looking for a corresponding `catch`.

If no `catch` block matches the `throw`, Ruby raises an `ArgumentError` exception at the location of the `throw`.

Typed Ruby

RBS has its own syntax for defining Ruby types. The definitive source for the syntax reference is found at https://github.com/ruby/rbs/blob/master/docs/syntax.md. We'll go over the structure here.

RBS Declarations and File Structure

RBS syntax has a few main parts (we're adopting the naming convention from the official syntax file):

- *Declarations* are class, module, and interface structures that can be used as types in other parts of the file.
- *Members* are things that are inside declarations that might have type information, including instance variables, attributes, and methods.
- *Types* are the things that RBS uses to specify actual type information, including class names, union of class names, or literal types, as well as a few RBS-specific ...

Part 5Ruby Library Reference

Ruby gets much of its functionality from its extensive library. That library is sometimes described as having two parts: the "core," which is part of Ruby and is included as part of every Ruby program, and the "standard library," which is shipped with Ruby but must be explictly required in code to be used.

In this part, we cover a curated list of the most important classes and their most useful methods in both the core and the standard library. We didn't separate the two. If a class in this list needs to be explicitly required, we note that as part of the description of that class. Note that we tried to keep related functionality together, so when you browse for one method, you might find another one that fits ...

Chapter 26
Library Reference: Core Data Types

In this chapter, we'll take a closer look at Ruby's core data types. The goal is to give you more information about what you can do with these classes and also to discuss related functions together so that you can browse and perhaps find a new feature that might help. We're presenting the topics in alphabetical order for easier browsing, and we'll cross-reference between topics as needed.

This isn't intended to be a complete listing of every class, method, or option. For that, please refer to the official Ruby documentation at https://docs.ruby-lang.org.

In this chapter, when a method is mentioned for the first time, we provide its complete name and signature. The notation `Foo.bar` indicates a class ...

Dates and Times

Ruby has three separate classes to represent date and time data: `Time`, `Date`, and `DateTime`.

- `Time` represents a specific moment in time, and you can retrieve both date and time information based on that. The `Time` class is based on a library that is common to Unix systems and is used by many programming languages.

- `Date` represents a date only with no time information attached. It's useful for calendar arithmetic that doesn't depend on the time of day. You need `require "date"` to use the `Date` class.

- `DateTime` also represents a specific moment in time but uses a different internal representation than `Time`. `DateTime` is now considered deprecated—at one point it had a more complete API than `Time`, but that is no longer true. Currently, ...

Math

The Numeric class and its subclasses handle a lot of the basic arithmetic in Ruby. (For more see *Numbers*.) But sometimes you need to do more advanced math. The Math module provides a couple dozen math functions, largely trigonometry, but it also handles other branches of advanced math. The BigMath model recreates a subset of those methods for BigDecimal arguments and values.

All of these methods are module methods, so they are all called with module syntax, as in Math.cos(value).

The Math module contains multitudes, especially if you like trigonometry. Specifically, it contains:

- The constant values Math::Pi and Math::E.
- Math.sqrt(x) and Math.cbrt(x), which return the square and cube roots of the argument, respectively.
- Logarithm functions, ...

Numbers

We talked about numeric literals in *Integer and Floating-Point Numbers*. Here, let's talk about the library methods for numbers.

Numeric Class

All number classes inherit from the class `Numeric`. If you want to define your own number subclass, it's recommended that you also inherit from `Numeric` because `Numeric` does some internal things about storing numbers in memory that are useful to have.

It might go without saying—but it's our job to say it—that Numeric implements `Numeric#<=>` and includes `Comparable`. As with other `<=>`, the return value is `-1` if the left side is smaller, `1` if the left side is greater, and `0` if the two are equal. The equality test is also aliased as `eql?`.

It's also our job to say that the actual arithmetic operators— ...

Random and SecureRandom

Random numbers are an important part of games and cryptographic security, and Ruby has a few different ways to get randomness. The easiest is the method `Kernel#rand(max = 0)`. Because it's available in `Kernel`, you can call `rand` anytime. If you call `rand` with no arguments, you get a pseudo-random float greater than or equal to zero and less than one. If you call `rand` with an argument and the argument is an integer one or greater, then you get a pseudo-random integer greater than or equal to zero and less than the argument. Note this means that `rand(1)` will always equal `0`.

Non-integer arguments are converted using `to_i.abs`, meaning that negative arguments will return positive values and floating-point arguments will be ...

Regexp

The `Regexp` class is the Ruby representation of a regular expression. We discussed the basic syntax of regular expressions at length in Chapter 8, *Regular Expressions*. Here we focus on the API for the `Regexp` class itself and then cover some advanced regular expression syntax.

You can create a regular expression using the literal syntax, which is two foreword slashes with the regular expression in the middle, such as `/.*rb/`. (All the escape sequences and whatnot inside the slashes are discussed in Chapter 8, *Regular Expressions*. The alternate delimiter `%r{...}` will also create regular expressions.

The method, `Regexp.new(string, options = 0, timeout: nil)` creates a new regular expression with the `string` argument as the pattern. The ...

Strings

Strings are probably the most commonly used data type in Ruby, and they have a powerful and wide-ranging API to prove it. Here are some of the most useful and most interesting String methods.

Finding Information about a String

The length of a string is accessible with the method `String#length` or `String#size`, which are aliases of each other. The length is in characters and is determined by the current encoding. You can get the length in bytes with the method `String#bytesize`. The method `String#empty?` returns `true` if the length of the string is zero.

If you want to know how many times a given character is used in a string, you can use `String#count(*selectors)`. This works, as you probably expect, if you pass in a single character, but it ...

Symbols

Symbols don’t have a lot of methods in Ruby. We’ve seen `Symbol#to_proc` as a shortcut for creating a block. The `to_proc` method creates a block that is effectively equivalent to `{ |receiver, ...| receiver.send(symbol, ...)}`. You usually see it used implicitly with `&`, but it can be used explicitly, too:

```
proc = :split.to_proc
proc.call("The pickaxe book")       # => ["The", "pickaxe", "book"]
proc.call("The pickaxe book", "e")  # => ["Th", " pickax", " book"]
```

You can also use `Symbol#to_s` to convert a symbol to a string; the `Symbol#name` and `Symbol#inspect` methods perform basically the same conversion.

Symbols respond to the following string-like methods, which are effectively shortcuts for calling `to_s` so that you don’t have ...

Chapter 27
Library Reference: Ruby's Object Model

In this chapter, we'll take a closer look at the classes that make up Ruby's object model. The goal is to give you more information about what you can do with these classes and also to discuss related functions together so that you can browse and perhaps find a new feature that might help.

This isn't intended to be a complete listing of every class, method, or option. For that, please refer to the official Ruby documentation at https://docs.ruby-lang.org.

In this chapter, when a method is mentioned for the first time, we provide its complete name and signature. The notation `Foo.bar` indicates a class or module method, while `Foo#bar` indicates an instance method. Optional arguments are indicated ...

BasicObject

For most purposes in Ruby, you can consider the `Object` class to be the root of Ruby's class hierarchy, and the `Kernel` module to be mixed into all objects. But in some specialized classes, you might want a Ruby object that doesn't have the basic functionality contained in `Object` and `Kernel`. For example, you might want a very minimal data object, or you might want to experiment with your own metaprogramming tools or object semantics.

For those cases, you want the class `BasicObject` which is the *real* root of Ruby's class hierarchy. `BasicObject` deliberately has just a few methods, allowing it to be conveniently used as the basis for a number of metaprogramming techniques.

If you write code in a direct descendent of `BasicObject`, you won't ...

Class

Classes in Ruby are first-class objects—each is an instance of class `Class`. Since `Class` is a subclass of `Module`, most of the behavior of `Class` is actually defined in the class `Module` (see *Module*). The `Class` class itself adds a small number of new methods.

You can create an anonymous class with `Class.new` and assign it to a variable and use it like a regularly defined class. If you assign it to a variable whose name starts with a capital letter, it's treated as a constant and behaves exactly like a regularly defined class.

When a new class is defined (typically using `class SomeName ... end`), an object of type `Class` is created and assigned to a constant (`SomeName`, in this case). When `SomeName.new` is called to create a new object, the ...

Comparable

If you want to be able to compare two objects of the same class in general, all you need to do is implement the `<=>` operator and include the `Comparable` module:

built_in_data/team.rb

```
class Team
  include Comparable
  attr_accessor :wins, :losses, :name

  def initialize(name, wins, losses)
    @name = name
    @wins = wins
    @losses = losses
  end

  def percentage = (wins * 1.0) / (wins + losses)

  def <=>(other)
    raise ArgumentError unless other.is_a?(Team)
    percentage <=> other.percentage
  end

  def to_s = name
end

brewers = Team.new("Brewers", 73, 89)
cardinals = Team.new("Cardinals", 86, 76)
cubs = Team.new("Cubs", 103, 58)
pirates = Team.new("Pirates", 78, 83) ...
```

Kernel

The `Kernel` module is included by class `Object`, so its methods are available in every Ruby object. One of the reasons for the `Kernel` module is to allow methods like `puts` and `gets` to be available everywhere and even to look like global commands. `Kernel` methods allow Ruby to still maintain an “everything is an object” semantics. `Kernel` methods actually use the implicit receivers and could be written as `self.puts` and `self.gets`. As a result, `Kernel` methods are both available everywhere and resolve like any other method.

Since `Kernel` is mixed into `Object`, there is no practical difference between a method defined in `Kernel` and a method defined in `Object`. Logically, the distinction is that the methods in `Object` manage the object-oriented semantics ...

Method

A Ruby `Method` object represents a method that is attached (the technical term is “bound”) to a specific receiver. You create `Method` objects via `Object#method(name)`, which returns the method object for a given name.

Once you have a method object, you can call the method with `call(*, **, &)`, which forwards the arguments to the method and invokes it with the method object’s receiver. The `call` method is aliased as `[]` and also as `===`.

You can compose Ruby methods in a functional programming style with `Method#<<(other_proc)`, which takes a proc or callable object as the right-hand side and returns a new proc. The new proc takes arguments, calls `other_proc`, and then calls the given method with the result of the call to `other_proc`. The flip ...

Module

The `Module` class is the class of any module you declare with the `module` keyword. Each module is an instance of the class `Module`. The class `Class` is a subclass of the class `Module`, and so it inherits all the functionality described here.

You can create an anonymous module with `Module.new [{block}]`, the block body is the body of the module. You can assign the module to a variable. If that variable name starts with a capital letter, then it's a constant and you can treat it exactly like a module that's created in the more common way.

Information about Modules

The `Module` class has a number of methods that allow you to dynamically access information about a module or class.

Module (and Class) Hierarchy

You can compare two modules to determine ...

Object

`Object` is the parent class of (almost) all classes in Ruby unless a class explicitly inherits from `BasicObject`. So, its methods are available to all objects unless explicitly overridden.

`Object` mixes in the `Kernel` module, making the built-in kernel functions globally accessible (see *Kernel*). The methods discussed here for the `Object` class mostly pertain to Ruby's object-oriented semantics.

An interesting fact about Ruby's actual implementation is that even these methods that are documented here (and in Ruby's official documentation) as being part of `Object` are actually all internally defined in `Kernel`. You can prove this by calling `owner` on any `Method` object in `Object`, as in `Object.instance_method(:itself).owner`. The official documentation ...

Chapter 28
Library Reference: Enumerators and Containers

In this chapter, we'll take a closer look at Ruby's collection classes, especially those features that are based on the `Enumerable` module, which is the basis for the functionality of all container classes in Ruby. The goal is to give you more information about what you can do with these classes and also to discuss related functions together so that you can browse and perhaps find a new feature that might help.

This isn't intended to be a complete listing of every class, method, or option. For that, please refer to the official Ruby documentation at https://docs.ruby-lang.org.

In this chapter, when a method is mentioned for the first time, we provide its complete name and signature. The ...

Array

Arrays are ordered, integer-indexed collections that may contain any Ruby object. The objects in the array do not need to be of the same type. They can be created using the literal square bracket syntax discussed in Chapter 4, *Collections, Blocks, and Iterators*. The `%w` delimiter with a space-delimited list can create an array of strings, and `%i` can similarly create an array of symbols. The `Array.new(size, default = nil)` method creates a new array with the given size and populated with the default object. The `Kernel#Array(object)` method converts its argument into an array if the argument isn't already an array.

Arrays implement `Array#each [{ |element| block}]` and mix in the `Enumerable` module, so all methods of `Enumerable` described in ...

Enumerable

Ruby's `Enumerable` module is the basis for the functionality of all container classes in Ruby. The most common container classes in use are `Array`, `Hash`, and sometimes `Set`, but this functionality applies to any other class that defines an `each` method and includes the `Enumerable` module.

In this section, we'll be talking about features common to all Enumerables. Other sections in this chapter will talk about how the core implementations of `Array`, `Hash`, and `Set` add their own features.

Iterating

The `Enumerable` module looks for a method called `each` as the building block for basically all of its functionality. The `Enumerable` module doesn't define `each`; instead, it depends on any class that includes `Enumerable` to define `each`. The basic contract ...

Enumerator

Nearly every `Enumerable` method that takes a block argument can also be called without a block, in which case the method returns an `Enumerator`. (There are a few subclasses of `Enumerator` that you wouldn't create by hand but which implement some specialized logic.) In addition to those `Enumerable` methods, you can create an enumerator in a few other ways.

Creating Enumerators

Any object can be converted into an `Enumerator` with the `Object#to_enum(method = :each, *args)` method (aliased as `enum_for`). The first argument to `to_enum` is a symbol that's the name of a method that converts the object to something enumerable. Any further arguments to `to_enum` are passed to the method named in the first argument. You can then treat that enumerator ...

Hash

The `Hash`, which associates arbitrary indexes to arbitrary values, is the most flexible basic class in Ruby. Although `Hash` does implement `each` and `Enumerable`, hashes behave slightly differently than arrays and sets.

Creating Hashes

Hashes have a literal syntax that uses curly braces to associate keys with values. The original form of separating keys from values uses the `=>` symbol, often called a *hash rocket*:

```
hash = {"a" => 1, "b" => 2, "c" => 3}
```

In general, you want the hash keys to be immutable values, and symbols are commonly used. If the key is a symbol, then you can use a colon to separate the key from a value. The colon will also convert strings to symbols. The keys in this literal are `:a`, `:b`, and `:c`, all symbols:

```
hash = { ...
```

Set

A Ruby `Set` is somewhere between an `Array` and a `Hash`—it's a collection of unique items. The `Set` class is a subclass of `Object`, and it defines `each` and includes `Enumerable`, so all `Enumerable` methods described in this chapter apply to sets. The elements of a `Set` are ordered the way that `Hash` keys are ordered—they preserve the sequence in which the elements were added to the `Set`. In other words, iterating over a set multiple times will always result in the same ordering, but you can't access arbitrary elements of the set via an index.

To use `Set` in versions of Ruby before 3.2, you need to explicitly call `require "set"`.

Creating Sets

There are several ways to create a `Set`. The `Enumerable` class defines `Enumerable#to_set`, which converts the collection ...

Chapter 29
Library Reference: Input, Output, Files, and Formats

In this chapter, we'll take a closer look at Ruby's input and output (I/O) classes, including reading and writing from files, manipulating files, and managing file formats. We'll investigate their API and functionality in somewhat more detail than we did in Part I of this book. The goal of this chapter is to give you more information about what you can do with these classes and also to discuss related functions together so that you can browse and perhaps find a new feature that might help.

This isn't intended to be a complete listing of every class, method, or option. For that, please refer to the official Ruby documentation at https://docs.ruby-lang.org.

In this chapter, when a ...

CSV

Comma-separated data files are often used to transfer tabular information, especially for importing and exporting spreadsheet and database information. Ruby's current CSV library is based on James Edward Gray II's FasterCSV gem. The CSV object has possibly the best official documentation in the entire Ruby library, and it goes beyond what's discussed here.

Ruby's CSV library deals with arrays (corresponding to the rows in the CSV file) and strings (corresponding to the elements in a row). If an element in a row is missing, it'll be represented as `nil` in Ruby.

The generic CSV parsing method is `CSV.parse(string_or_io, headers: nil, **options) [{|row|}]` and takes an optional block. The main argument is either a string or an `IO` object and ...

Dir

The `Dir` class is used to interact with directories in the file system. Like many of Ruby's file manipulation classes, it has both a lot of class methods and instance methods, in some cases duplicating functionality.

You create a `Dir` instance with `Dir.new(path)`, where the path is a string or something that can be implicitly converted to a string because it implements `to_str` (so `Pathname` objects can be used here). The path is relative to the current system working directory. There's an optional keyword argument, `encoding:`, which specifies the encoding of the directory as you look at it.

Directories can technically be opened and closed in Ruby; when open, they stream a list of their children files. The class method `Dir.open(path) {|dir| ...`

File

The `File` class in Ruby does two things. First, it's a subclass of `IO`, meaning it handles reading data from and writing data to files. Second, it adds a number of methods for manipulating files as objects, similar to the `Dir` class. Since the reading and writing behavior is mostly managed by methods defined in `IO`, we'll discuss it when we look at `IO` in *IO*. Here, we'll look at file-specific manipulations.

Most `File` manipulation methods are class methods, while most of the read/write methods are instance methods.

Opening a File

You create a `File` instance with `File.new(filename, mode = "r", permissions = 0666, *options)` or `File.open(filename, mode = "r", permissions = 0666, *options) [{block}]`. (Note that this doesn't necessarily create the ...

FileUtils

In addition to the functionality in `File`, Ruby has an entire module called `FileUtils` that defines many module level methods that are basically wrappers around operating system features or `Dir` and `File` features.

To use these methods, you need to use `require "fileutils"`. All the methods here are defined as module methods and as instance methods, though you would typically use them as module methods, as in `FileUtils.mkdir`.

Methods in `FileUtils` that take paths expect either a string, an object with a `to_path` method, or an object with a `to_str` method. Methods in `FileUtils` that are described as working recursively can take directories as arguments and act on all files in the directory.

Let’s take a quick tour of the `FileUtils`.

Common Arguments ...

IO

Class `IO` is the basis for all input and output in Ruby. An I/O stream may be *duplexed* (that is, bidirectional) and so may use more than one native operating system stream. Many of the examples in this section use class `File`, which is the only standard subclass of `IO`. The two classes are closely associated.

As used in this section, *portname* may take any of the following forms:

- A plain string represents a filename suitable for the underlying operating system.
- A string starting with `|` indicates a subprocess. The remainder of the string following `|` is invoked as a process with appropriate input/output channels connected to it.
- A string equal to `|-` will create another Ruby instance as a subprocess.

The `IO` class uses the Unix abstraction of ...

JSON

JSON[52] is a language-independent data interchange format based on key/value pairs (hashes in Ruby) and sequences of values (arrays in Ruby). JSON is frequently used to exchange data between JavaScript running in browsers and server-based applications. JSON isn’t a general-purpose, object marshaling format.

Ruby makes JSON methods available with `require "json"`.

Parsing JSON

The general Ruby method for parsing JSON is `JSON.parse(source, options = {})`. The source is a JSON string. The output is a Ruby object. If the JSON is an object, you get a Ruby hash. If the JSON string is an array, you get a Ruby array. If the JSON is a scalar, you get a Ruby object of the matching type, and that’s true recursively of sub-objects as well. The commonly ...

Pathname

A `Pathname` represents the absolute or relative name of a file. It has two distinct uses. First, it allows manipulation of the parts of a file path (extracting components, building new paths, and so on). Second, it acts as a facade for some methods in the `Dir` and `File` classes and the `FileTest` module, forwarding on calls for the file named by the `Pathname` object.

The class `Pathname` is part of the Ruby Standard Library, meaning it ships with Ruby but is only available to code that explicitly requires it using `require "pathname"`.

You create a pathname with `Pathname.new(path)`, which takes a string argument. The method `Pathname.pwd` returns the current working directory as a path, and the method `Pathname.glob` is essentially a wrapper around ...

StringIO

In some ways, the distinction between strings and file contents is artificial: the contents of a file are basically a string that happens to live on disk, not in memory. The `StringIO` class, available by using `require "stringio"`, aims to unify the two concepts, making strings act as if they were opened `IO` objects. Once a string is wrapped in a `StringIO` object, it can be read from and written to as if it were an open file. This can make unit testing a lot easier.

The `StringIO` class isn't a subclass of `IO`, it just implements many of the same read/write methods. Using `StringIO` lets you pass strings into classes and methods that were originally written to work with files. `StringIO` objects take their encoding from the string you pass in—if ...

Tempfile

Class Tempfile creates managed temporary files. Although they behave the same as any other IO objects, temporary files are automatically deleted when the Ruby program terminates. Once a Tempfile object has been created, the underlying file may be opened and closed a number of times in succession.

Tempfile doesn't directly inherit from IO. Instead, it delegates calls to a File object. From the programmer's perspective, apart from the unusual new, open, and close semantics, a Tempfile object behaves as if it were an IO object.

If you don't specify a directory to hold temporary files when you create them, the Dir.tmpdir location will be used to find a system-dependent location. Here's an example:

ref_io/tempfile.rb

```
require "tempfile" ...
```

URI

`URI` encapsulates the concept of a Uniform Resource Identifier (URI), a way of specifying some kind of (potentially networked) resource. URIs are a superset of URLs. URLs (such as the addresses of web pages) allow specification of addresses by location, and URIs also allow specification by name. The `URI` classes are available with `require "uri"`.

The `URI` class can be used to do the following:

- Parse URIs into component parts.
- Open a stream to the network location referred to by the URI.
- Manage encoding and decoding of strings to be safe for use in URLs.

URIs consist of a scheme (such as `http`, `mailto`, `ftp`, and so on), followed by structured data identifying the resource within the scheme.

Parsing is managed with the method `URI.parse(string)`, which ...

YAML

The YAML library, available with `require "yaml"`, serializes and deserializes Ruby object trees to and from an external, readable, plain-text format. YAML can be used as a portable object marshaling scheme, allowing objects to be passed in plain text between separate Ruby processes. In some cases, objects may also be exchanged between Ruby programs and programs in other languages that also have YAML support.

The `YAML` module in Ruby is an alias to `Psych`, which is the name of the YAML parser being used. We mention this because it may be easier to find further documentation searching for `Psych` rather than `YAML`.

Writing YAML

YAML can be used to store an object tree in a string. The API call is `YAML.dump(object, io = nil, options = {})`. The ...

Chapter 30
Library Reference: Ruby on Ruby

In this chapter, we'll take a closer look at some useful classes in Ruby that you might use for metaprogramming or observation. We'll investigate their API and functionality in somewhat more detail than we did in Part I of this book. The goal of this chapter is to give you more information about what you can do with these classes and also to discuss related functions together so that you can browse and perhaps find a new feature that might help.

This isn't intended to be a complete listing of every class, method, or option. For that, please refer to the official Ruby documentation at https://docs.ruby-lang.org.

In this chapter, when a method is mentioned for the first time, we provide its complete ...

Benchmark

The `Benchmark` module allows code execution to be timed and the results tabulated. `Benchmark` is easier to use if you include it in your top-level environment.

The most useful method of `Benchmark` is `Benchmark.bm(label_width = 0, *labels) { |report| ...}`. The `bm` method passes a report object to the block. Inside the block, you call `report(caption)` on that object one or more times, passing a block each time. Ruby will execute each block and emit a table with an entry for each block listing the time spent by the CPU executing code (user time), the CPU time spent by the system during the block (system time), the total of those two (total), and the amount of clock time that passed during the block.

This example compares the costs of four ...

Data

Ruby provides the `Data` class to be used as an immutable data object. The intent of `Data` is to create an object similar to a `Struct`, but whose attributes cannot be changed (see *Struct*).

You create new `Data` classes with the `define` method. As with `Struct`, you can then create new instances with either positional or keyword arguments, and you can read those arguments:

ref_meta_ruby/data_1.rb

```
Classroom = Data.define(:name, :capacity)
auditorium = Classroom.new("auditorium", 1000)
math = Classroom.new(name: "X 206", capacity: 30)

auditorium.capacity  # => 1000
```

Unlike `Struct`, you can't write the attributes of a `Data` object, but you can create new instances using `with`. The `with` method takes keyword arguments and returns ...

Delegator and SimpleDelegator

Object delegation is a way of *composing* objects—extending an object with the capabilities of another—at runtime. The Ruby `Delegator` class implements a simple but powerful delegation scheme, where requests are automatically forwarded from a master class to delegates or their ancestors and where the delegate can be changed at runtime with a single method call. The class `SimpleDelegator` is an implementation of `Delegator` that's good enough for most purposes.

The typical use of `SimpleDelegator` is as a decorator. You create a class as a subclass of `SimpleDelegator`. You create new instances of the simple delegator class by passing it an existing instance of another class. When you call a method on the delegator, it'll ...

Logger

Ruby has a `Logger` class that's accessible with `require "logger"`. It writes log messages to a file or stream and supports automatic time- or size-based rolling of log files. Messages can be assigned severities, and only those messages at or above the logger's current reporting level will be logged.

A new logger is created with `Logger.new(location, shift_age = 0, shift_size = 1048576, **options)`.

The `location` is one of the following:

- A string, which is interpreted as a filename; log entries are appended to the file.
- An `IO` stream, in which case log entries are written to the stream. The stream can be an open `File` object or any of Ruby's global streams, like `$stdout`, but any stream will work.
- `nil` (or `File::NULL`), in which case log entries ...

ObjectSpace

The `ObjectSpace` module contains a number of routines that interact with the garbage collection facility and allow you to traverse all living objects with an iterator.

`ObjectSpace` also provides support for object finalizers. These are procs that will be called when a specific object is about to be destroyed by garbage collection. Typically, you either call `ObjectSpace` methods as module methods as in `ObjectSpace.count_objects` or you include `ObjectSpace` as a module in another class and call the methods directly.

This is just a glance at what `ObjectSpace` can do; there's more in the official documentation.

The method `ObjectSpace.define_finalizer(object, proc = proc())` adds `proc` as a finalizer, called automatically when `object` is about ...

Observable

The Observer pattern, also known as Publish/Subscribe, provides a simple mechanism for one object (the source) to inform a set of interested third-party objects when its state changes. In the Ruby implementation, the notifying class mixes in the module `Observable`, which provides the methods for managing the associated observer objects. The observers must implement the `update` method to receive notifications.

The way this works is that the class that's sending the notifications adds `include Observable`. To add subscribers to the notifications, you call `Observable#add_observer(observer, method = :update)`. The observer is an object that receives a notification, and `method` is the method that's automatically called when a notification is ...

OpenStruct

If `Data` is the most immutable way to get a small object, `OpenStruct` is the most flexible. An `OpenStruct` isn't a class generator, rather, it's more a way to allow you to have hash-like data with attribute-like syntax.

You create an `OpenStruct` with `new` method, taking either a hash argument or an arbitrary set of keyword arguments. After that, you can read, write, and create attributes just by using them, and you can also use hash syntax:

```
require "ostruct"

bulb = OpenStruct.new(brightness: 1600, watts: 15, color: 2500)
bulb.color   # => 2500
bulb[:watts] # => 15
bulb.shape = "A19"
bulb.to_h    # => {:brightness=>1600, :watts=>15, :color=>2500, :shape=>"A19"}
```

Internally `OpenStruct` uses `method_missing`, which ...

PP

`PP` uses the `PrettyPrint` library to format the results of inspecting Ruby objects. In addition to the methods in the class, it defines a global function, `pp`, which works like the existing `p` method but formats its output.

`PP` has a default layout for all Ruby objects. But you can override the way it handles a class by defining the method `pretty_print`, which takes a `PP` object as a parameter. It should use that `PP` object's methods (`text`, `breakable`, `nest`, `group`, and `pp`) to format its output:

ref_meta_ruby/pretty_print.rb

```
require 'pp'

Customer = Struct.new(:first_name, :last_name, :dob, :country)
cust = Customer.new("Walter", "Wall", "12/25/1960", "Niue")

puts "Regular print"
p cust

puts "\nPretty print" ...
```

Prism

Prism is a new parsing gem in Ruby 3.3 that's likely to be the future parsing library for Ruby and Ruby tooling. Prism can be used as a stand-alone gem; it is also included with the Ruby standard library. You can use it with `require "prism"`.

Prism is designed to be a common parser across Ruby implementations, and as of this writing, it's also integrated into Ruby and TruffleRuby, among others. It's also used by the ruby-lsp language server.

Prism is designed to be tolerant to errors. For example, if you're typing a file in your editor, Prism tries not to let syntax errors in one part of the file affect the parsing of the rest of the file, making for a much better development experience while typing.

Prism includes an API that you can ...

Ripper

The ripper library, available with `require "ripper"`, gives you access to Ruby's parser. It can tokenize input, meaning it can convert a string of Ruby code into a series of semantic elements called tokens. It can return a lexical analysis of those tokens and what they mean to Ruby. And it can return a nested S-expression, which is a tree-like structure that represents the relationship between the tokens in the code. Ripper also supports event-based parsing.

Here's an example that shows the possibilities on a single string of Ruby code:

ref_meta_ruby/ripper_1.rb

```
require "ripper"

content = "a=1;b=2;puts a+b"

p "Tokens"
p Ripper.tokenize(content)
puts
p "Lexical analysis"
pp Ripper.lex(content)[0,5]
puts
p "S-Expressions" ...
```

Singleton

The Singleton design pattern ensures that only one instance of a particular class may be created for the lifetime of a program.

The `Singleton` module makes this simple to implement. Mix the `Singleton` module into each class that's to be a singleton, and that class's `new` method will be made private. In its place, users of the class call the method `instance`, which returns a singleton instance of that class.

Ruby overrides a few other methods of the class when `Singleton` is mixed in: `inherited`, `clone`, `_load`, and `dup`, all of which are changed to prevent multiple instances of the class from existing.

In this example, the two instances of `MyClass` are the same object:

ref_meta_ruby/singleton.rb

```
require "singleton"

class MyClass
 attr_accessor ...
```

Struct

Sometimes you want to create a small object to hold data that has little to no behavior of its own, and a Ruby class seems like too much structure to bother with.

Ruby has a few lightweight ways to create classes that have little to no behavior.

The most commonly used is probably `Struct`. You can use `Struct` to create instance-like objects that have attributes and can respond to messages that you define.

Using `Struct` creates a class that you then create instances of. Let's say you want to represent a classroom that has a name and a capacity. You use `Struct.new` to create a `Classroom` class, with the desired attribute names as arguments. You can then use `Classroom.new` to create new classrooms. The positional arguments in `new` match the order ...

Unbound Method

UnboundMethod is a method that's not currently attached to an instance, which means it can't yet be called. UnboundMethod instances are created by Module methods such as instance_method and can also be created by calling unbind on a Method object.

To use an UnboundMethod, you must bind it to an object using UnboundMethod#bind(object), which returns a Method that can be called. The object must be a member of the class that the unbound method came from or a subclass of that class. You can get that class with UnboundMethod#owner. It's pretty common to call the method immediately so the shortcut method UnboundMethod#bind_call(object, ...) is equivalent to bind(object).call(...) but with a performance improvement because it doesn't ...

Part 6 Appendixes

Included in the appendixes is information on troubleshooting misbehaving Ruby code, a collection of Ruby symbols and their meanings that might be hard to look up, a more detailed introduction to using a command line, information about Ruby runtimes, and a list of significant changes made to Ruby in each version since 2.0.

Appendix 1
Troubleshooting Ruby

You've read through this entire book, you start to write your very own Ruby program, and…it doesn't work. Here's a list of common gotchas and other tips to help get you back up and running.

Common Issues

- In Ruby, unlike in JavaScript and Python, a method name with no parentheses calls the method with no arguments. It doesn't return the method as an object to be used later. To get the method object, use the method named `method`.

- In Ruby, calling a class as if it were a method, as in `Classname()`, is an error. To create a new instance, you need to call `Classname.new()`.

- If you happen to forget a comma (,) in an argument list—especially to print—you can produce some very odd error messages.

- Ruby allows you to have a trailing comma at the end of an array or hash literal, method call, or block method list, but not at the end of the parameter list of a method definition.

- If Ruby is telling you that the number of arguments being passed ...

Debugging Tips

- Read the error message! Ruby error messages have a lot of information, including the type of error, the location of the error, and the entire sequence of method calls that lead to the error. If the error is because a method name wasn't found, Ruby will suggest similarly named methods that actually exist. If the error is potentially at several points along a line, Ruby will attempt to show you where along the line the error happened.

- Running your scripts with warnings enabled (the `-w` command-line option) can give you insight into potential problems.

- If you cannot figure out where a method is defined, you can access the source location with `obj.method(:method_name).source_location`. This will return a two-element array with the ...

Appendix 2
I Can’t Look It Up!

Ruby has a lot of notation and typography that is called by a name that isn’t necessarily obvious, making it hard to search for the meaning of a particular line of code. Here are a few particularly important symbols:

Symbol	Name	Functionality
`||=`	Or-equals	Is like other Ruby operate and assign operators. `x ||= y` is equivalent to `x = x || y`. Because of Ruby’s short circuit of boolean operators, the expression means that if x is nil, the new value is y, and if x isn’t nil, then x’s value remains the same. Is often used as a shortcut to set a default value.
`=~ !~`	Match operators	With a string on one side of the operator and a regular expression on the other, `=~` returns true if the string matches the regular expression, ...

Appendix 3
Command-Line Basics

Although great support exists for Ruby in IDEs, you'll probably still end up spending a lot of time at your system's command prompt, also known as a *shell prompt* or just plain *prompt*. The most popular IDEs also provide their own shell prompts in another window right next to your code.

The Command Prompt

If you're a Linux user, you're probably already familiar with the command prompt. If you don't already have a desktop icon for it, hunt around for an application called Terminal or xterm.

On macOS, run Applications → Utilities → Terminal.app. (We also recommend the excellent iTerm2[53] on macOS.)

On Windows, you can install Windows Subsystem for Linux[54] and have a shell that behaves like the Linux or MacOS shells, or you can use the default Windows Power Shell, which, as we'll see, behaves a little differently. On Windows, we recommend installing Windows Terminal (https://docs.microsoft.com/en-us/windows/terminal/install), which makes it easier to use other shell type.

When you run the application, a fairly empty window ...

Folders, Directories, and Navigation

If you're used to a GUI tool such as Explorer on Windows or Finder on MacOS for navigating to your files, then you'll be familiar with the idea of *folders*—locations on your hard drive that can hold files and other folders.

When you're at the command prompt, you have access to these same folders. But at the prompt, these folders are usually called *directories* (because they contain lists of other directories and files). These directories are organized into a strict hierarchy. On Unix-based systems (including macOS and WSL), there's one top-level directory, called `/` (a forward slash). On plain Windows, there is a top-level directory for each drive on your system, so you'll find the top level for your `C:` drive ...

Appendix 4
Ruby Runtimes

Ruby code is converted to executable code using an interpreter. The current default interpreter is called YARV (Yet Another Ruby VM) and has been the standard since Ruby 1.9, replacing the original interpreter, which was known as CRuby or MRI (Matz's Ruby Interpreter). You'll actually still see the names CRuby and MRI used interchangeably with YARV for the current version of the interpreter.

The interpreter makes dozens of choices about how to convert Ruby code to machine code, such as how to store objects, associate objects with their methods, and so on. Each of these choices has implications for the runtime performance of Ruby.

Not all uses of Ruby are equal. A one-off script could be optimized for a quick startup ...

Just-in-Time Compilers

Historically, computer languages are translated in one of two different ways. A language might use a *compiler* to convert the program code directly to machine language. This compilation happens in a separate step before the code is executed and produces machine-language artifacts. When it’s time to run the code, the machine-language version is run, and the original source code isn’t used.

Ruby typically uses a different tool called an *interpreter*. An interpreter converts the source code to machine language at runtime, generally without creating an intermediate machine-language artifact. In an interpreted language, you typically use the original source code at runtime.

That said, the line between compilers and interpreters ...

TruffleRuby

Now we move away from runtime engines that ship with the Ruby virtual machine and toward Ruby implementations that run in various other environments. In general, these implementations offer performance improvements at the cost of lagging behind new features in Ruby and also limiting access to the Ruby ecosystem, since not all gems can run in the other environments.

GraalVM[56] is a virtual machine environment implemented in Java and designed to be a high-performance, cross-language, virtual environment. TruffleRuby,[57] created by Chris Seaton, is a Ruby implementation built on top of GraalVM. TruffleRuby provides very high performance relative to the standard Ruby implementations, but it’s not completely compatible with standard ...

JRuby

JRuby is the older and more established Java runtime version of Ruby. After several years of falling behind core Ruby, the November 2022 release (JRuby 9.4) brought JRuby to parity with Ruby 3.1, with Ruby 3.2 support expected in the near future. There are some substantial limitations in JRuby's support as of this writing.

- Ruby's threading constructs aren't completely supported, specifically Ractors and the thread scheduler aren't supported yet. But you can interoperate with Java thread-safe data.

- The readme for this version says "Nearly all features from CRuby's NEWS file have been implemented." Again, we recommend making sure your tests continue to run if you're considering switching to JRuby.

- Rails support for most databases is incomplete. ...

mRuby

mRuby (Minimalist Ruby) is an offshoot of official Ruby and is led by many of the same developers. It implements a subset of Ruby that's designed for a minimal memory footprint and for use embedded inside C programs where memory might be tight, such as inside devices. The idea is to allow hardware developers access to Ruby as a scripting language.

You can install mRuby via your ruby version manager (`rbenv install mruby-3.1.0`), or you can install it standalone from the download site.[62] If you install it via a version manager, you can run it using `ruby`; if you install it standalone, the command is `mruby`.

Other Runtimes

Other attempts to create Ruby runtimes that are in progress, aren't used much, or have been abandoned.

Artichoke Ruby

Artichoke Ruby is an attempt to build a CRuby-compliant Ruby runtime in Rust. It's currently in pre-production.

Opal

Opal is a Ruby-to-JavaScript compiler.

MagLev

MagLev is a Ruby runtime built on top of the GemStone Smalltalk runtime. It appears to have had very little development since 2016.

Rubinius

Rubinius is an attempt to build a Ruby runtime in Ruby, partially for use as a reference implementation. It appears to have had little development since 2020.

Iron Ruby

Iron Ruby was a .NET implementation of Ruby that appears to have had little development since 2011.

Footnotes

[55] https://www.solnic.dev/p/benchmarking-ruby-32-with-yjit ...

Appendix 5
Ruby Changes

For most of this book, we assume you're using the most current Ruby version, which is version 3.3. With a couple of exceptions, we don't specify when particular features were added to Ruby, as we find that makes the main text more confusing.

This appendix covers changes to Ruby that involve features that are mentioned in this book. There are many more changes in each version, many having to do with core library and gem methods that aren't covered here. Ruby's documentation[63] contains a listing of what was added to each version since 1.8.7. The Ruby Evolution site,[64] maintained by Victor Shepelev, contains more details about the changes. This appendix is only here to tell you when major changes first appeared, starting ...

Version 2.0

- `Module#prepend` is introduced.
- Default source encoding changes to UTF-8.
- Refinements are added.
- Keyword arguments are added, but a default value is always required, and the internal implementation still overlaps with positional arguments.
- `%i` is added as a delimiter for a list of symbols.
- Lazy enumerators are added.
- `to_h` and `Kernel#Hash` are added as the convention to convert to `Hash` objects.
- `TracePoint` is added.
- Numeric values are frozen.

Version 2.1

- Keyword arguments without a default are now allowed.
- `def` now returns the symbol name of the method.
- Literal syntax for rational and complex numbers is added.
- `Array` and `Enumerable` get a default `to_h`.

Version 2.2

- The method `Object#itself` is added, returning the receiving object.
- Unusual symbols are allowed as `Hash` keys as a string with a trailing colon, as in `{"unusual symbol": 1}`.

Version 2.3

- The safe navigation operator, `&.`, is added.
- `dig` is added to `Array`, `Hash`, and `Struct`.
- Heredoc with `~` that removes leading spaces, allowing for indented text, is added.
- `Hash#to_proc` is added.

Version 2.4

- Using `return` at the top level exits the program.
- All integers are now of class `Integer`. Previously, smaller integers were `Fixnum` and bigger ones were `Bignum`.
- Boolean methods for regular expression matches, `match?` are added.
- Refinements can be used in `send` and `Symbol#to_proc`.

Version 2.5

- Structs can be initialized with keywords.
- Exception `rescue` is allowed inside a block.

Version 2.6

- `Object#then` is added to allow chained functions.
- Ranges without ending values are allowed.

Version 2.7

- Experimental support for pattern matching is added.
- Blocks now support numbered parameters to match positional arguments, as in `[1, 2, 3].map { _1 * _1}`.
- Safety concepts are deprecated. They aren't covered in this book, but they were covered in the previous edition.
- Private methods are now accessible with `self` as the receiver. Previously, this had been an error.
- Keyword and positional arguments are differentiated internally, but old semantics aren't removed.
- The ability to forward arguments with `...` is added.
- The `Enumerator#produce` method is added.
- Ranges without beginning values are allowed.

Version 3.0

- Class variables can no longer be overridden in subclasses or including modules.
- One-line “endless” syntax for method definition is added.
- Keyword and positional arguments are now completely separated. Previously, `def foo(*arg)` would capture keyword arguments in `arg`.
- RBS is added for type definitions.
- Rightward assignment `=>` is added via pattern matching. The `in` operator for pattern matching becomes a boolean check.
- The pattern matching find pattern, `[*start, pattern, *rest]`, is added.
- Argument forwarding with `...` now allows specific arguments before the `...`.
- Non-blocking `Fibers` and the Fiber scheduling API are added.

Version 3.1

- The pattern-matching pin operator `^` allows expressions and variables that have a sigil.
- Values in keywords and hashes where the key is already a name in the local binding can be omitted. `{x:}` if `x` has a value.
- Block arguments can be received anonymously with `&`.
- Ractors are added.
- Major updates are made to IRB.
- Major updates are made to the internal debugging tool.

Version 3.2

- `Set` is added to the core library.
- Anonymous positional and keyword arguments can be passed through with `*` and `**`.
- `Data` object is added to the core library.
- `Struct` no longer requires `keyword_init: true` to be used with keyword arguments.
- The Rust implementation of YJIT is considered production-ready.

Version 3.3

- The Prism parser is added as a way to work with parsing Ruby code.
- MJIT is replaced with RJIT.
- YJIT gets substantial performance improvements.
- In Ruby 3.4, the keyword `it` is expected to be enabled as a synonym for `_1`. In Ruby 3.3, a use of `it` that might conflict with this future usage will result in a warning.

Footnotes

[63] https://docs.ruby-lang.org/en

[64] https://rubyreferences.github.io/rubychanges/evolution.xhtml

Thank you!
We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

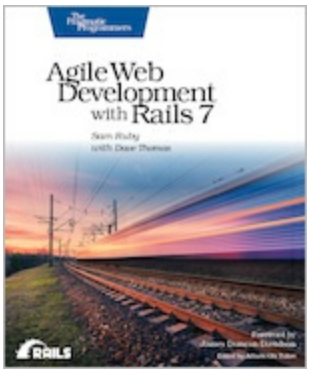
Head on over to https://pragprog.com right now, and use the coupon code BUYANOTHER2024 to save 30% on your next ebook. Offer is void where prohibited or restricted. This offer does not apply to any edition of the *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With up to a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit https://pragprog.com/become-an-author/ ...

You May Be Interested In…

Select a cover for more information

Agile Web Development with Rails 7



Rails 7 completely redefines what it means to produce fantastic user experiences and provides a way to achieve all the benefits of single-page applications – at a fraction of the complexity. Rails 7 integrates the Hotwire frameworks of Stimulus and Turbo directly as the new defaults, together with that hot newness of import maps. The result is a toolkit so powerful that it allows a single individual to create modern applications upon which they can build a competitive business. The way it used to be.

Sam Ruby

(474 pages) ISBN: 9781680509298 $59.95

Ruby ...