Contents

1 About this book **1**

2 Ruby **5**
2.1 What is Ruby? . . . 5
2.2 Philosophy and Principles . . . 6
2.3 Practical Characteristics . . . 7
2.4 Installing Ruby . . . 7
2.4.1 Ubuntu . . . 7
2.4.2 OSX . . . 8
2.4.3 Windows . . . 8
2.4.4 RVM . . . 8
Installing a Ruby interpreter . . . 11

3 Language Basics **15**
3.1 Dynamic typing . . . 17
3.2 Strong typing . . . 21
3.3 Basic types . . . 23
3.3.1 Integers . . . 24
Fixnums . . . 24
Bignums . . . 27
3.3.2 Floating point . . . 30
3.3.3 BigDecimals . . . 34
3.3.4 Dates . . . 36
3.3.5 Times . . . 38
3.3.6 Rationals . . . 42
3.3.7 Booleans . . . 44
3.3.8 Nil . . . 44
3.3.9 Strings . . . 45
3.3.10 Substrings . . . 50
3.3.11 Concatenating Strings . . . 52
3.3.12 Encodings . . . 52
3.3.13 Variables are references in memory . . . 54
3.3.14 Freezing objects explicitly . . . 54
3.3.15 Some String methods and tricks . . . 58
3.3.16 Symbols . . . 60

The philosophy of symbols . . . 61
3.3.17 Regular expressions . . . 63
Groups . . . 66
Named groups . . . 68
Accented characters . . . 69
Timeout . . . 70
3.3.18 Arrays . . . 72
3.3.19 Duck Typing . . . 81
3.3.20 Sets . . . 83
3.3.21 Ranges . . . 87
3.3.22 Hashes . . . 90
3.3.23 Code blocks . . . 95
3.3.24 Type conversions . . . 97
3.3.25 Base conversions . . . 98
3.3.26 Exception handling . . . 99
Raising exceptions . . . 103
Discovering the previous exception . . . 104
Creating our own exceptions . . . 105
Comparing exceptions . . . 105
Using catch and throw . . . 107
3.4 Control structures . . . 109
3.4.1 Conditionals . . . 109
if . . . 109
unless . . . 111
case . . . 112
3.4.2 Pattern matching . . . 114
3.4.3 Loops . . . 119
Magic comments . . . 120
while . . . 121
for . . . 122
until . . . 124
3.4.4 Operators . . . 126
Arithmetic operators . . . 126
Assignment operators . . . 126
Unary operators . . . 127
Logical operators . . . 128
Splat . . . 132
Bitwise operators . . . 134
3.5 Procs and lambdas . . . 141
3.6 Iterators . . . 146
3.6.1 Selecting elements . . . 149
3.6.2 Selecting elements that do not meet a condition . . . 151
3.6.3 Processing and modifying elements . . . 151
3.6.4 Detecting a condition in all elements . . . 153
3.6.5 Detecting if any element meets a condition . . . 153
3.6.6 Detecting and returning the first element that meets a condition . . . 154
3.6.7 Detecting maximum and minimum values . . . 154

3.6.8 Accumulating elements 157
3.6.9 Splitting the collection into two Arrays based on a condition 160
3.6.10 Traversing elements with their indices 161
3.6.11 Sorting a collection 162
3.6.12 Combining elements 162
3.6.13 Traversing values upward and downward 164
3.6.14 Filtering with grep 164
3.6.15 Chaining iterators 166
3.6.16 Random numbers 169
3.7 Methods 170
3.7.1 Returning values 171
3.7.2 Sending values 172
3.7.3 Sending and processing blocks and Procs 178
3.7.4 Values are passed by reference 183
3.7.5 Intercepting exceptions directly in the method 184
3.7.6 Destructive and predicate methods 185
3.8 Sending options on the command line 189

4 Classes and objects 195
4.1 Open classes 208
4.2 Aliases 210
4.3 Inserting and removing methods 211
4.4 Inheritance 213
4.5 Shallow and deep copying 220
4.6 Metaclasses 227
4.7 Class variables 236
4.8 Fluent interfaces 238
4.9 DSLs 239
4.10 Class instance variables 242
4.11 Playing with dynamic methods and hooks 244
4.12 Delegation 247
4.13 Operator-like methods 247
4.14 Closures 260
4.15 Static typing support 261
4.15.1 Generating the RBS file 262
4.15.2 Checking the types 264
4.16 Prism 268

5 Modules 271
5.1 Mixins 272
5.1.1 Composition versus inheritance 274
5.1.2 Where methods are inserted 277
5.1.3 Modules extending themselves 285
5.1.4 Implementing singletons 287
5.1.5 Refinements 288
5.2 Namespaces 291
5.3 TracePoint 293

5.4 Ruby::Box 296

6 RubyGems 301
6.1 Installing manually 301
6.2 Commands 302
6.3 Using the installed gem 303
6.4 Using memoization with the gem 306
6.5 Managing with a Gemfile 311
6.6 Tail call optimization 315

7 Threads 319
7.1 Threads 319
7.1.1 Sending Procs 322
7.1.2 Mutexes 325
7.1.3 Queues 330
7.1.4 Canceling 332
7.1.5 Intercepting signals 334
7.2 Fibers 338
7.3 Continuations 348
7.4 Parallel processes 350
7.5 Benchmarks 357
7.5.1 Measuring time 357
7.5.2 Seeing where time is spent 360
7.6 Ractors 364
7.6.1 Understanding the Actor Model 365
7.6.2 Getting to know Ractor::Port 372
7.6.3 Reusing Ports 374
7.6.4 Isolation 385
7.6.5 Waiting 385
7.6.6 Copying and moving 387
7.6.7 Creating pools 390
7.6.8 Checking parallelism 393
7.6.9 Sharing Procs and lambdas 396
7.6.10 Error handling 398
7.6.11 Recommendations 400
7.6.12 Choosing between Threads, Fibers and Ractors 401

8 JIT 405
8.1 MJIT 405
8.2 RJIT 406
8.3 Enabling YJIT 406
8.4 ZJIT 408

9 Input and output 409
9.1 Files 409
9.1.1 FileUtils 416
9.1.2 Zip files 417
9.1.3 CSV 419

9.1.4 Creating . . . 419
9.1.5 Reading . . . 420
9.1.6 XML . . . 424
9.1.7 XSLT . . . 430
9.1.8 JSON . . . 433
9.1.9 YAML . . . 435
9.2 Network protocols . . . 440
9.2.1 TCP . . . 440
9.2.2 UDP . . . 444
9.2.3 SMTP . . . 446
9.2.4 POP3 . . . 448
9.2.5 FTP . . . 449
9.2.6 HTTP . . . 450
9.2.7 HTTPS . . . 458
9.2.8 SSH . . . 459
9.3 Operating system processes . . . 461
9.3.1 Backticks . . . 461
9.3.2 System . . . 461
9.3.3 Exec . . . 462
9.3.4 IO.popen . . . 462
9.3.5 Open3 . . . 463
9.4 XML-RPC . . . 465
9.4.1 Python . . . 467
9.4.2 PHP . . . 468
9.4.3 Java . . . 470

10 JRuby **473**
10.1 Using Java classes from inside Ruby . . . 474
10.2 Using Ruby classes inside Java . . . 478

11 Databases **479**
11.1 Installing the necessary gems . . . 480
11.2 Opening the connection . . . 480
11.3 Executing queries . . . 481
11.4 Queries that do not return data . . . 482
11.5 Updating a record . . . 483
11.6 Deleting a record . . . 484
11.7 Queries that return data . . . 484
11.8 Prepared statements . . . 487
11.9 Metadata . . . 487
11.10 ActiveRecord . . . 488

12 C extensions **491**
12.1 Using external libs . . . 497
12.1.1 Writing the C code of the lib . . . 497
12.2 Using the shared lib . . . 499

13 Garbage collector **501**

13.1 Phase 1: Initial State . . . 503
13.2 Phase 2: Unreachable Object . . . 503
13.3 Phase 3: Mark Phase . . . 504
13.4 Phase 4: Sweep Phase . . . 504
13.5 Heap Structure . . . 508
13.5.1 Young Space . . . 508
13.5.2 Old Space . . . 508
13.6 Types of collection . . . 508
13.6.1 Minor GC . . . 508
13.6.2 Major GC . . . 509
13.6.3 Object promotion . . . 509
13.7 This is not a C book, but 510
13.8 This is still not a C book, but 511
13.8.1 A small detail: not every String uses malloc/free . . . 513

14 Testing 519
14.1 Starting with the classic API . . . 523
14.2 Failing . . . 525
14.3 Pending . . . 527
14.4 Omitted . . . 528
14.5 Notifications . . . 530
14.6 Assertions . . . 532
14.7 Modernizing the tests . . . 534
14.7.1 Randomizing the tests . . . 536
14.7.2 Testing with specs . . . 537
14.7.3 Benchmarks . . . 539
14.8 Mocks . . . 541
14.9 Stubs . . . 543
14.10 Expectations . . . 544
14.11 Different output formats . . . 547
14.12 Debugging . . . 550
14.13 Continuous testing . . . 552
14.13.1 Running tests automatically . . . 554
14.13.2 Running only selected tests . . . 555

15 Building gems 559
15.1 Creating the gem . . . 560
15.2 Testing the gem . . . 565
15.3 Building the gem . . . 566
15.4 Publishing the gem . . . 566
15.4.1 Publishing locally . . . 566
15.4.2 Publishing to the official repository . . . 569
15.5 Extracting a gem . . . 571
15.6 Signing a gem . . . 571
15.6.1 Creating a certificate . . . 571
15.6.2 Adapting the gem to use the certificate . . . 572
15.6.3 Building and publishing the signed gem . . . 573

15.6.4 Using the signed gem . . . 573
15.6.5 Always using signed gems . . . 574

16 Rake **577**
16.1 Defining a task . . . 577
16.2 Namespaces . . . 579
16.3 Dependent tasks . . . 580
16.4 Running tasks in other programs . . . 582
16.5 Different files . . . 582
16.6 Tasks with file names . . . 584
16.7 Tasks with file lists . . . 586
16.8 Rules . . . 588
16.9 Extending . . . 590

17 AI **593**
17.1 The RubyLLM gem . . . 595
17.1.1 Creating a chat . . . 595
Text chat . . . 596
Interpreting an image . . . 601
Interpreting a video . . . 602
Interpreting a document . . . 603
Interpreting code . . . 603
Interpreting audio . . . 604
Using static methods . . . 605
Creating images . . . 606

18 Generating documentation **609**
18.1 Rdoc . . . 609
18.2 YARD . . . 615

19 Challenges **621**
19.1 Challenge 1 . . . 621
19.2 Challenge 2 . . . 621
19.3 Challenge 3 . . . 622
19.4 Challenge 4 . . . 622
19.5 Challenge 5 . . . 622
19.6 Challenge 6 . . . 623

20 Companies **625**

7.6 Ractors

Starting from version 3.0.0, we have the `Ractor` class available (previously called `Guild`), which despite starting as experimental, has awakened great interest in the developer community.

As we saw earlier, we have situations where code can run in a parallel manner (multiple tasks executing simultaneously) or concurrently (multiple tasks being managed, but only one executing at a time, especially when competing for the same resource).

Before version 3.0.0, the Ruby language did not support true parallelism because of the GIL ("Global Interpreter Lock"), that synchronization mechanism that works like a global semaphore already discussed earlier, allowing only one `Thread` to execute Ruby code at a time.

With `Ractors`, each one has its own GIL and its own isolated execution context. This means that multiple `Ractors` can truly execute Ruby code in parallel, each on its own CPU core, keeping objects completely isolated from each other. Communication between `Ractors` happens exclusively through a specific message-passing protocol.

As a curiosity, the name `Ractor` comes from "Ruby Actor" (based on the `Actor Model` of concurrent computing).

Objects that can be shared between `Ractors` are called *shareable* and include many, such as numbers (`Integer`, `Float`), symbols (`Symbol`), immutable `strings`, frozen objects (freeze), and the `Ractor` object itself, as well as other objects for which a deep copy (*deep copy*) can be made.

We can test whether an object is *shareable* through the `shareable?` method:

```
Ractor.shareable?(1)            # true
Ractor.shareable?([])           # false
Ractor.shareable?([].freeze)    # true
```

When the object is not *"shareable"*, an automatic deep copy (*deep copy*) of the object is made, however some objects do not support deep copy for use with `Ractors` (such as `Thread`), so be careful!

7.6.1 Understanding the Actor Model

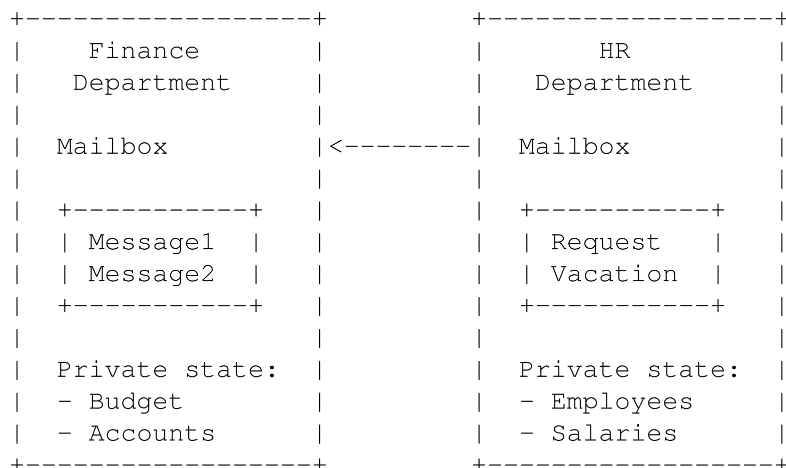
Before diving into the Ruby 4.0 `Ractors`, it is important to understand the `Actor Model`, which is the concurrency model that inspired `Ractors`.

The `Actor Model` is a mathematical model of concurrent computing created by Carl Hewitt in 1973. The central idea is simple and powerful: everything is an actor.

An actor is an entity that:

- Has private state - no one outside can access it
- Has a mailbox - where it receives messages
- Processes one message at a time - sequentially
- Can send messages to other actors
- Can create new actors
- Can decide how to respond to the next message

Think of a company:



Where:

- Each department has its private state (internal data)
- Each department has a mailbox
- Departments do not access each other's data directly
- They communicate only through messages
- Each department processes one request at a time

We have several advantages in this model:

No Race Conditions

Since each actor processes one message at a time, there is no simultaneous access to state:

```ruby
# Threads can have race conditions:

@counter = 0

threads = 100.times.map do
  Thread.new { @counter += 1 }  # DANGER!
end
threads.map(&:join)

puts @counter

# @counter may not be 100!

# Actors have no race conditions!
# Each actor has ITS OWN private counter
# Increment happens sequentially
```

Fault Isolation

If an actor fails, the others keep working. If actor A crashes, actors B and C continue normally:

```
Actor A [CRASH]
Actor B [OK]
Actor C [OK]
```

Scalability

Actors can be in different `Threads`, different processes, or even different machines! Same API, but different contexts:

```
Local Actor       -> Thread
Parallel Actor    -> Ractor (Ruby 4.0)
Distributed Actor -> Different machines
```

Conceptual Simplicity

You think in terms of "who does what" instead of locks, mutexes, semaphores. Instead of thinking "I need to lock this mutex before accessing...", you think "I will send a message to the responsible actor".

Let us look at the `Actor Model` versus shared memory:

```
Shared Memory (Traditional Threads)
-----------------------------------
Thread A --+
           |-> [Shared Memory] <- DANGER!
Thread B --+       (needs locks)

Actor Model (Ractors)
---------------------
Actor A -+
         +- [Message] -> Actor B
Actor C -+                    |
                        [Private state]
                          (no locks!)
```

Famous Implementations

- Erlang/Elixir: Languages built heavily on the Actor Model
- Akka (JVM): Actor framework for Java/Scala, which became a commercial option
- Apache Pekko: open-source fork of Akka
- Orleans (.NET): Microsoft's Virtual Actors
- Ruby Ractors: Ruby's implementation!

Erlang is a functional programming language created by Ericsson in the late 1980s to develop highly reliable and concurrent telecommunications systems. Its main differentiator is the actor-based concurrency model, in which thousands or millions of lightweight and isolated processes communicate through message passing, allowing the construction of fault-tolerant distributed systems. Erlang also incorporates concepts like "let it crash", automatic supervisors, and hot code reloading, making it especially suited for applications that require high availability, such as message servers, telecom systems, and real-time platforms.

Elixir is a modern functional programming language created by José Valim that runs on the Erlang virtual machine (BEAM). It maintains the same capabilities of massive concurrency, fault tolerance and distribution of the Erlang ecosystem, but offers a more expressive and productive syntax, strongly inspired by Ruby. Elixir is widely used to build scalable and highly available applications - such as APIs, real-time systems and web applications - especially through the Phoenix Framework.

In Ruby 4.0, `Ractors` implement the Actor Model quite faithfully:

Actor Model Concept	Ruby 4.0 Implementation
Actor	`Ractor.new { ... }`
Mailbox	`Ractor::Port.new`
Send message	`port « message`
Receive message	`port.receive`
Private state	Variables inside the `Ractor` block
Isolation	Only shareable objects (`shareable`) pass between `Ractors`

To better understand, let us implement a simple banking system that demonstrates all the concepts. We need to run this code with the *flag* `-W0`, since `Ractors` are still marked as experimental:

> Want to always run with `-W0`? Configure a variable in your *shell*'s initialization file:
>
> export RUBYOPT="-W0"

```ruby
def create_bank_account(name, start_balance, port)
  Ractor.new(name, start_balance, port) do |name, balance, port|
    loop do
      puts "Waiting in #{name} ..."
      msg = Ractor.receive # aqui pega do default_port do Ractor
      puts "-> Received #{msg[:operation]} in #{name}!"

      case msg[:operation]
      when :deposit
        balance += msg[:valor]
        port << { ok: true, balance: balance, account: name }
      when :withdraw
        if balance >= msg[:valor]
          balance -= msg[:valor]
          port << { ok: true, balance: balance, account: name }
        else
          port << { ok: false, erro: "Insufficient balance", account: name }
        end
      when :query
        port << { ok: true, balance: balance, account: name }
      when :stop
        puts "  Closing #{name}'s account..."
        break
      end
      puts "Processed in #{name}, returning to wait ..."
    end
  end
end

joao_answer, maria_response  = Ractor::Port.new, Ractor::Port.new
account_joao  = create_bank_account("João", 1_000, joao_answer)
account_maria = create_bank_account("Maria", 500, maria_response)

puts "Depositing $ 200 in João's account"
account_joao << { operation: :deposit, valor: 200 }
puts "   #{joao_answer.receive}\n\n"

account_joao << { operation: :query }
puts "João's balance: $ #{joao_answer.receive[:balance]}\n\n"

puts "Trying to withdraw $ 1000 from Maria (will fail)"
account_maria << { operation: :withdraw, valor: 1000 }
result = maria_response.receive
puts result[:ok] ? "Balance $ #{result[:balance]}\n\n" : "#{result[:erro]}\n\n"

account_maria << { operation: :withdraw, valor: 100 }
puts "Result of withdrawing $ 100 from Maria: #{maria_response.receive}\n\n"

account_maria << { operation: :query }
puts "Maria's final balance: $ #{maria_response.receive[:balance]}\n\n"

account_joao  << { operation: :stop }
account_maria << { operation: :stop }

account_joao.join
account_maria.join
```

Code 7.31: Bank accounts with Ractors

Running the code:

```
$ ruby -W0 ract_account.rb
Depositing $ 200 in João's account
Waiting in João ...
-> Received deposit in João!
Waiting in Maria ...
Processed in João, returning to wait ...
Waiting in João ...
   {ok: true, balance: 1200, account: "João"}

-> Received query in João!
Processed in João, returning to wait ...
João's balance: $ 1200

Waiting in João ...
Trying to withdraw $ 1000 from Maria (will fail)
-> Received withdrawal in Maria!
Insufficient balance

Processed in Maria, returning to wait ...
Waiting in Maria ...
-> Received withdrawal in Maria!
Processed in Maria, returning to wait ...
Result of withdrawing $ 100 from Maria: {ok: true, balance: 400, account: "M

Waiting in Maria ...
-> Received query in Maria!
Processed in Maria, returning to wait ...
Waiting in Maria ...
Maria's final balance: $ 400

-> Received stop in Maria!
-> Received stop in João!
  Closing João's account...
  Closing Maria's account...
```

The secret of `Ractors` there is the `default_port`. Each `Ractor` has an automatic `default_port`, which greatly simplifies communication:

- `Ractor.receive` inside the `Ractor`: reads messages from its `default_port`
- `ractor « message` outside the `Ractor`: sends to the `Ractor`'s `default_port`
- For responses, we create explicit `Ports` and pass them along with the message
- Golden rule: whoever creates the `Port` can `receive` from it, others can only `« (send)`

Internally, `Ractor.receive` is equivalent to `Ractor.current.default_port.receive` and `ractor « msg` is equivalent to `ractor.default_port « msg`.

We do not need to create a new `Port` for every operation - it is much more efficient to create one `Port` per context (per client, per thread, per session) and reuse it, as we saw in the code above. This makes the code cleaner, more efficient and easier to understand!

What happens in the code above:

- Each account is an actor with private state (balance)
- No *race condition* - even with 1000 threads making simultaneous deposits, the balance will be correct since each message is processed sequentially
- Isolation - if João's account *crashes*, Maria's continues working
- *Message passing* - all communication via messages (`Ports`)
- Request/Response - client sends a response port when creating the `Ractor`

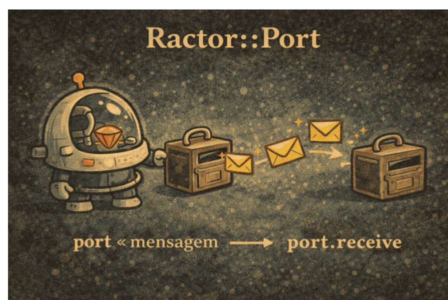
The `Actor Model` solves the classic concurrency problems in an elegant way:

- No *race conditions* - private state, sequential processing
- No *deadlocks* - there are no *locks*!
- Scalable - easy to distribute actors across CPUs/machines
- Fault tolerant - isolation between actors
- Easy to understand - think of "entities that exchange messages"

> **IMPORTANT**: The `Actor Model` (and `Ractors`) are ideal for *CPU-bound* tasks (calculations, processing). For *I/O-bound* tasks (HTTP requests, database, file reading), use `Threads` or `Async I/O`, as seen earlier. `Ractors` have *overhead* that is not worth it when the bottleneck is I/O!

Now that we have seen an overview of the `Actor Model`, `Ractors` in Ruby 4.0 make much more sense! Let us see more examples and explanations of how the new API with `Ports` implements this model perfectly.

7.6.2 Getting to know Ractor::Port



Starting from Ruby 4.0, the Ractors API has changed completely. The new approach uses `Ractor::Port` for communication between `Ractors`, which is much more intuitive and better represents the `Actor Model`. But what changed?

- Removed: `Ractor.yield` and `Ractor.take`
- Added: `Ractor::Port` for communication
- Added: `Ractor.join` and `Ractor.value` (similar to `Threads`)
- Modified: `Ractor.select` now only accepts `Ractors` and `Ports`

Now you can see why `Ractors` are still marked as experimental. The Ruby language has a tradition of not breaking things between one version and another, but experimental features are used at your own risk and may change significantly.

`Ports` are "mailboxes" that allow bidirectional communication between `Ractors`. Any `Ractor` can send messages to a `Port`, but only the `Ractor` that created the `Port` can receive messages from it. `Ports` are better than the old API because the Ruby 3.x API tried to do too much "magic" and ended up confusing:

```ruby
# Ruby 3.x

ractor = Ractor.new do
  # receive from where, from whom?
  msg = Ractor.receive

  # yield to where, to whom?
  Ractor.yield "processed: #{msg}"
end

# Sending happens "magically" through the ractor
ractor.send "message"

# Take is also "magical"
result = ractor.take
```

With `Ports`, everything is explicit and clear: