RUBY

Fundamentals

Getting Started · Variables & Data Types · Control Flow

Methods & Blocks · Arrays & Hashes Deep Dive

Classes & Objects · Modules & Mixins

Blocks, Procs & Lambdas

Philip Osztromok

Generated with Claude

Table of Contents

Ruby Fundamentals — 8 Chapters

💎 Getting Started with Ruby

Ruby is an interpreted, dynamically-typed, thoroughly object-oriented language created by Yukihiro "Matz" Matsumoto in the mid-1990s, explicitly designed around programmer happiness and readability rather than raw performance. This chapter covers what makes Ruby distinct from PHP and Python — the two languages you already know — and the absolute basics: running a script, the interactive REPL, and printing output.

💡 What Is Ruby, and Why Learn It?

Ruby is:

- **Interpreted and dynamically typed** — like PHP and Python, no compile step, types are checked at runtime
- **Purely object-oriented** — unlike PHP and Python, there are no primitives at all in Ruby. Every value, including numbers and `nil`, is an object you can call methods on
- **Designed for programmer happiness** — Matz has been explicit that Ruby prioritizes how enjoyable and readable the language feels to write, sometimes over strict consistency
- **The engine behind Ruby on Rails** — the web framework that made Ruby famous; this course covers the language itself, with Rails as a natural follow-up
- **Expression-oriented** — most Ruby constructs (including `if` and `case`) return a value, which shows up constantly once you start writing idiomatic Ruby

Ruby vs PHP vs Python, at a Glance

Trait	PHP	Python	Ruby
File extension	.php	.py	.rb
Primitives	int/string/bool are not objects	Mostly objects, some exceptions	Everything is an object, no exceptions
Run a script	php script.php	python script.py	ruby script.rb
Interactive shell	php -a (rarely used)	python (REPL)	irb (central to the workflow)
Statement terminator	Semicolon required	Newline (no semicolons)	Newline (semicolons optional)
Block delimiters	{ }	Indentation	do...end or { }, plus end keywords

▶️ Your First Ruby Program

Hello, Ruby

```
# hello.rb
puts "Hello, Ruby!"
```

Run it from the command line:

```
$ ruby hello.rb
Hello, Ruby!
```

No class wrapper, no `public static void main`-style boilerplate, no build step — a Ruby file is just a sequence of statements executed top to bottom, the same way a PHP or Python script runs. `puts` is a top-level method available everywhere by default, not something you import.

🖥️ irb — The Interactive Ruby Shell

Ruby ships with **irb** (Interactive Ruby), a REPL (read-eval-print loop) you launch by typing `irb` at the command line. It plays a much bigger role in everyday Ruby work than PHP's rarely-used `php -a`: it's the standard way to quickly test an expression, inspect an object's available methods, or experiment with syntax before writing it into a file — and it's the same underlying tool Rails' `rails console` is built on, so getting comfortable with it now pays off directly later.

```
$ irb
irb(main):001:0> 2 + 2
=> 4
irb(main):002:0> "Ruby".upcase
=> "RUBY"
irb(main):003:0> exit
```

📣 puts, print, and p — Three Ways to Output

Where PHP has `echo`/`print` and Python has a single `print()`, Ruby gives you three distinct output methods, each with a different job:

```
puts "Hello" # Hello\n  -- adds a newline automatically
print "Hello" # Hello    -- no trailing newline
p "Hello" # "Hello"  -- shows the inspect form, quotes included
puts [1, 2, 3]  # prints each element on its own line: 1 / 2 / 3
p [1, 2, 3]     # prints the array literally: [1, 2, 3]
```

puts — for humans

Adds a trailing newline, and if you give it an array, prints each element on its own line. This is the everyday "show the user something" method — closest to PHP's `echo` with a manual `"\n"` already built in.

p — for debugging

Prints the *inspect* form of an object — strings show their quotes, arrays show their brackets exactly as written. This is the method you reach for when you want to see the real shape of a value, similar in spirit to Python's `repr()`.

Comments

```ruby
# A single-line comment, same symbol as Python
=begin
This is a multi-line block comment.
Ruby is one of the few languages with a dedicated
block-comment syntax rather than reusing single-line comments.
=end
puts "Comments don't affect output" # inline comments work too
```

Ruby's single-line `#` matches Python exactly (PHP supports `#` too, though `//` is more common there). The `=begin`/`=end` block form is unique to Ruby — PHP and Python both reuse other syntax (`/* */`, triple-quoted strings) instead of having a true dedicated block-comment construct.

🧬 Everything Is an Object

This is the single biggest philosophical difference from PHP and Python, and it shows up immediately:

```
5.class        # => Integer
5.even?        # => true  -- calling a method directly on an integer literal
nil.class      # => NilClass  -- even nil is an object with a class
nil.to_s       # => ""     -- nil responds to methods like anything else
"ruby".class   # => String
```

⚠ In PHP/Python, this would look very different

In PHP, `5` is a scalar, not an object — you can't call a method directly on an integer literal at all. Python is closer (integers *are* objects, and `(5).bit_length()` works), but Python's `None` and Ruby's `nil` behave differently: calling an arbitrary method on `None` raises an `AttributeError`, while `nil` in Ruby happily responds to a defined set of its own methods, like `nil.to_s` above. Keep this distinction in mind — it'll matter again once method chaining shows up.

💻 Coding Challenges

Challenge 1: puts vs print vs p

Write a Ruby script that creates a string variable and an array variable, then prints each one using all three of `puts`, `print`, and `p` — six lines of output total. Add a comment above each line explaining, in your own words, what's different about the output.

Goal: Build a clear mental model of when to reach for each of Ruby's three output methods.

→ Solution

Challenge 2: Comments & Running a Script

Create a file called `profile.rb`. Using both single-line `#` comments and an `=begin`/`=end` block comment, document a short script that prints your name, a favorite number, and whether that number is even or odd (using `.even?`). Run it from the command line with `ruby profile.rb`.

Goal: Get comfortable writing, saving, and running a real `.rb` file rather than only experimenting in irb.

→ Solution

Challenge 3: Everything Is an Object

In irb, call at least four different methods directly on literals with no variable assignment — for example `10.class`, `"hello".length`, `nil.to_a`, `true.class`. Write down each result, then write one sentence explaining how this would need to be written differently in PHP.

Goal: Internalize that in Ruby, literals aren't a special case — they're objects like anything else.

→ Solution

💎 Why "Programmer Happiness" Isn't Just Marketing

You'll notice Ruby often gives you several ways to do the same thing (`puts`/`print`/`p` is the first example this chapter covers, but it's a pattern that recurs throughout the language). That's a deliberate design choice, not an inconsistency — Matz optimized for the syntax reading naturally in context over having exactly one rigid way to do something. Coming from PHP and Python's generally stricter "one obvious way" philosophy, expect Ruby to feel more flexible, and occasionally more ambiguous, by design.

🎯 What's Next

Next chapter: **Variables & Data Types** — symbols (a Ruby-specific type with no direct PHP/Python equivalent), strings, numbers, the basics of arrays and hashes, and how `nil` fits into all of it.

📦 Variables & Data Types

Chapter 1 showed that every Ruby value is an object. This chapter covers the everyday types you'll build those objects out of — strings, numbers, arrays, and hashes — plus two ideas with no clean PHP or Python equivalent: symbols, and how `nil` actually behaves in a boolean context.

✏️ Variables — No Declaration Keyword

Ruby variables need no declaration keyword and no sigil — just assign a name and Ruby infers the type from the value, exactly like Python. This is a step further than PHP, which requires the `$` sigil on every variable reference, not just at declaration:

```
age = 33 # no $ sigil, no let/var/val keyword
name = "Philip"
is_ready = true # snake_case is Ruby's naming convention, like Python
```

Ruby variable names use **snake_case** by convention (like Python; PHP's convention varies between snake_case and camelCase depending on the codebase). Variable names are entirely lowercase by convention too — a name starting with a capital letter means something different in Ruby, as you'll see once classes and constants show up in later chapters.

🔤 Strings — Single vs Double Quotes Actually Matter

Both single and double quotes create strings, but unlike PHP (where the same distinction exists) they aren't interchangeable — only double quotes support interpolation and escape sequences:

```ruby
name = "Philip"
puts "Hello, #{name}!" # Hello, Philip!  -- double quotes interpolate
puts 'Hello, #{name}!' # Hello, #{name}! -- single quotes are literal
puts "Line one\nLine two" # \n becomes a real newline in double quotes
puts 'Line one\nLine two' # \n stays as literal backslash-n in single quotes
```

String Interpolation: PHP vs Python vs Ruby

	PHP	Python	Ruby
Interpolating syntax	"Hi $name"	f"Hi {name}"	"Hi #{name}"
Requires special prefix?	No	Yes — f-string	No — plain double quotes
Non-interpolating quote	'...' (literal)	Any plain string	'...' (literal)

Common string methods: `"ruby".upcase` → `"RUBY"`, `"RUBY".downcase` → `"ruby"`, `"ruby".length` → `4`, `"ruby".reverse` → `"ybur"`, and concatenation with `+` or repetition with `*` (`"ab" * 3` → `"ababab"`).

🔢 Numbers — Watch Out for Integer Division

```
10 / 3 # => 3   -- Integer / Integer stays an Integer, truncated
10.0 / 3 # => 3.3333333333333335  -- a Float on either side forces Float division
10.fdiv(3)  # => 3.3333333333333335  -- explicit float division method
```

⚠ This gotcha is a real trap coming from PHP or Python 3

In PHP, `10 / 3` automatically produces a float (`3.3333...`) — there's no separate integer-division operator by default. In Python 3, `/` is always float division (`10 / 3` → `3.333...`), and you'd deliberately reach for `//` to truncate. Ruby does the opposite of both: plain `/` between two integers **truncates**, silently, with no warning. Forgetting this is one of the most common early Ruby bugs for anyone coming from either language.

Symbols — The New Concept

A **symbol** is a lightweight, immutable identifier written with a leading colon: `:name`, `:status`, `:pending`. Neither PHP nor Python has a direct equivalent — the closest PHP gets is a bare string constant, and the closest Python gets is an enum member, but neither carries Ruby's specific guarantee: **every instance of the same symbol in a running program is the exact same object in memory.**

```
:pending.object_id == :pending.object_id   # => true  -- always the same object
"pending".object_id == "pending".object_id # => false -- two different String objects
```

Why symbols exist

Because a symbol is always the same object, comparing two symbols for equality is a fast identity check rather than a character-by-character string comparison. That makes symbols the natural choice for things that name something — hash keys, method names, status values — rather than for text that gets displayed or manipulated.

Symbols vs strings, in practice

Use a **symbol** (`:active`) when the value is really an internal label or identifier. Use a **string** (`"active"`) when it's actual text — something you'll display, concatenate, or manipulate character by character. This distinction shows up constantly once hashes are in play, just below.

Arrays

```ruby
fruits = ["apple", "banana", "cherry"]

fruits[0]      # => "apple"  -- zero-indexed, same as PHP/Python
fruits[-1]     # => "cherry"  -- negative indexing, same as Python (PHP has no native equivalent)
fruits.length    # => 3
fruits << "date" # append -- equivalent to fruits.push("date")
fruits.first     # => "apple"
fruits.last      # => "date"
```

Ruby arrays are ordered, can hold mixed types in the same array (like PHP and Python lists), and the `<<` "shovel" operator for appending is Ruby-specific — PHP uses `$arr[] = $value`, Python uses `.append(value)`.

🔑 Hashes

A hash is Ruby's key-value structure — directly comparable to a PHP associative array or a Python dict. Modern Ruby code almost always uses symbol keys with a shorthand syntax:

```
# Modern shorthand (symbol keys) -- the style you'll see most often
person = { name: "Philip", age: 33 }
person[:name]    # => "Philip"
# Old "hashrocket" syntax -- still required for non-symbol keys, like strings
scores = { "Philip" => 95, "Alex" => 88 }
scores["Philip"]  # => 95
```

Key-Value Structures: PHP vs Python vs Ruby

	PHP	Python	Ruby
Literal syntax	["name" => "Philip"]	{"name": "Philip"}	{ name: "Philip" }
Read a value	$arr["name"]	d["name"]	hash[:name]
Arrays and maps	Same type (associative array)	Separate: list vs dict	Separate: Array vs Hash

nil — Ruby's "Nothing"

`nil` is Ruby's equivalent of PHP's `null` or Python's `None` — it represents the absence of a value. The important, easy-to-miss detail: in a boolean context, Ruby treats only two values as falsy — `nil` and `false`. Everything else, including `0` and `""` (empty string), is truthy.

```
if 0
puts "0 is truthy in Ruby!" # this DOES print
end
value = nil
value.nil?        # => true -- the idiomatic way to check for nil
```

⚠ 0 is truthy — a real difference from PHP and Python

PHP treats `0`, `"0"`, and `""` as falsy in a boolean context. Python treats `0`, `0.0`, and empty collections/strings as falsy too. Ruby treats **only** `nil` and `false` as falsy — `0` and `""` are both truthy. An `if 0` check that would skip a block in PHP or Python will run it in Ruby.

💻 Coding Challenges

Challenge 1: Interpolation vs Concatenation

Create variables for a product name (string) and a price (float). Build the same sentence two ways: once using string interpolation with double quotes, and once using `+` concatenation (you'll need `.to_s` on the price). Print both and compare how much more readable the interpolated version is.

Goal: Get comfortable with `#{ }` interpolation as the default, idiomatic choice.

→ Solution

Challenge 2: Symbols Are Identical Objects

In irb, create the symbol `:status` twice and compare their `.object_id` — confirm they match. Then create the string `"status"` twice and compare those `.object_id`s — confirm they *don't* match. Write one sentence explaining why this matters for hash keys.

Goal: Directly observe symbols' defining property rather than just reading about it.

→ Solution

Challenge 3: An Array of Hashes

Build an array containing three hashes, each with symbol keys `:name` and `:age`, representing three people. Loop over the array with `.each` and print a sentence for each person using string interpolation to pull values out of the hash.

Goal: Combine arrays, hashes, and symbol keys the way they'll actually be used together in real Ruby code.

→ Solution

💎 A Rule of Thumb for Symbols vs Strings

If you're ever unsure whether something should be a symbol or a string, ask: "will this value ever be displayed to a user, or manipulated as text?" If yes, it's a string. If it's really just a label your own code uses internally — a hash key, a status, a type name — reach for a symbol. This one habit will make your hashes and method calls look like idiomatic Ruby rather than PHP or Python translated line-by-line.

🎯 What's Next

Next chapter: **Control Flow** — if/unless, case/when, loops (while, until, for, each), and what it means for Ruby to be expression-oriented — where even an `if` statement returns a value you can assign.

🔀 Control Flow

Chapter 2 established that only `nil` and `false` are falsy in Ruby — everything else, including `0` and `""`, is truthy. This chapter puts that rule to work in `if`, `case`, and loops, and introduces the idea that will keep resurfacing for the rest of this course: in Ruby, control flow structures aren't just statements — they're expressions that return a value.

❓ if / elsif / else

```
age = 20
if age < 13
puts "child"
elsif age < 20
puts "teenager"
else
puts "adult"
end
```

No parentheses around the condition, no curly braces around the body — Ruby uses indentation for readability (like Python) but the block is actually delimited by the `end` keyword (like PHP's alternate `if: ... endif;` syntax, though that form is rarely used in real PHP code). Note also `elsif` — not `else if` (PHP) or `elif` (Python), a small but very common early typo.

The modifier form

For a single statement, Ruby lets you write the condition *after* the code — a postfix form with no real equivalent in PHP or Python:

```
puts "You can vote" if age >= 18
puts "Still a minor" unless age >= 18
```

🙅 unless — Ruby's if not

`unless condition` runs its body when the condition is falsy — the direct opposite of `if`. Neither PHP nor Python has a dedicated keyword for this; both just write `if (!condition)` or `if not condition`. Ruby style strongly prefers `unless` over `if !` when there's no accompanying `else`, because it reads closer to plain English:

```
unless user.nil?
  puts "Welcome back!"
end
# Avoid: unless ... else ... end -- it reads backwards and is discouraged style
```

💡 Ruby Is Expression-Oriented

This is the chapter's central idea: in Ruby, `if` and `case` evaluate to the value of whichever branch ran, so you can assign the result directly instead of declaring a variable first and mutating it inside each branch:

```
temperature = 75
description = if temperature > 80
"hot"
elsif temperature > 60
"warm"
else
"cold"
end
puts description   # => warm
```

Assigning a Conditional Result: PHP vs Python vs Ruby

	PHP	Python	Ruby
Approach	Ternary only: $x = $a ? $b : $c;	Conditional expression: x = b if a else c	Full if/elsif/else block IS an expression
Multi-branch?	Needs nested ternaries (unreadable) or a temp variable	Needs nested conditional expressions or a temp variable	Any number of elsif branches, still one clean assignment

Ruby still has a ternary operator too (`age >= 18 ? "adult" : "minor"`), useful for short one-liners — but for anything with more than two branches, an expression-oriented `if` avoids the unreadable nested-ternary problem PHP and Python code sometimes falls into.

🔁 case / when

Ruby's answer to PHP's `switch` (Python only gained a comparable `match` statement in 3.10) — and, being Ruby, it's an expression too:

```
grade = "B"
message = case grade
when "A" then "Excellent"
when "B", "C" then "Good"
when "D" then "Needs improvement"
else "Not a valid grade"
end
puts message   # => Good
```

`when "B", "C"` matches either value with no fall-through logic needed — Ruby's `case` never falls through to the next branch the way PHP's `switch` does without an explicit `break`. `case` can also match against a `Range` or a class, which is worth knowing exists even before classes are formally covered:

```
score = 85
case score
when 90..100 then puts "A"
when 80...90 then puts "B" # matches -- 85 is in this range
else puts "Below B"
end
```

🔁 Loops — while, until, for, and .each

```ruby
i = 0
while i < 3
puts i
 i += 1
end

i = 0
until i == 3 # the exact opposite of while -- no PHP/Python keyword equivalent
puts i
 i += 1
end
for n in 1..3 # for exists, but idiomatic Ruby rarely uses it -- see below
puts n
end
```

⚠ for exists, but real Ruby code almost never uses it

Unlike PHP and Python, where `for` loops are completely normal and common, idiomatic Ruby strongly prefers iterator methods like `.each` over the `for` keyword. It's not a hard rule, but seeing a bare `for` loop in Ruby code is a signal the author is likely coming from another language and hasn't adopted Ruby style yet. The next section shows why.

```ruby
[1, 2, 3].each do |n|
 puts n
end
# Same thing, curly-brace block syntax -- idiomatic for short one-liners
[1, 2, 3].each { |n| puts n }
```

Why .each over for

A Ruby `for` loop doesn't create its own scope — the loop variable leaks out and stays defined after the loop ends, which most Ruby developers consider a wart carried over from C-style languages. `.each` is a method call that takes a block, and the block variable stays cleanly scoped to the block itself.

do...end vs { }

Both delimit a block; the convention is `do...end` for multi-line blocks and `{ }` for short one-liners. This is purely a style convention, not a functional difference — but following it is part of what makes Ruby code look idiomatic rather than translated from another language.

📜 Control Flow Keywords, Side by Side

PHP vs Python vs Ruby

Concept	PHP	Python	Ruby
Negative-else keyword	none (if !)	none (if not)	unless
Switch/match	switch (needs break)	match (3.10+)	case/when (no fall-through)
Reverse-while loop	none	none	until
Idiomatic iteration	foreach	for x in list	.each do |x| ... end
if/case as expression	No (ternary only)	Partial (conditional expr only)	Yes, fully

💻 Coding Challenges

Challenge 1: Modifier if/unless

Given an integer variable `stock`, write two lines using the modifier form: one that prints `"In stock"` using `if`, and one that prints `"Reorder soon"` using `unless`. Test both with a value above and below 10.

Goal: Get comfortable reading and writing Ruby's postfix conditional style.

→ Solution

Challenge 2: case/when as an Expression

Given an integer `day_number` (1–7), use `case/when` to assign the corresponding day name directly to a variable called `day_name` (no separate `puts` inside each branch) — then print `day_name` once, after the `case` block.

Goal: Practice treating `case` as an expression rather than a sequence of side-effecting branches.

→ Solution

Challenge 3: for vs .each

Write the same task two ways: print the numbers 1 through 5 using a `for` loop, then print them again using `.each` on a range (`(1..5).each`). Add a comment noting which version is the idiomatic Ruby choice and why.

Goal: Feel the stylistic difference firsthand, not just read about it.

→ Solution

💎 Expression-Oriented Code Reduces Temp Variables

Once `if` and `case` return values, a whole category of PHP/Python pattern — declare an empty variable, then reassign it inside every branch of a conditional — mostly disappears from idiomatic Ruby. Get in the habit of asking "can I just assign this directly?" before reaching for the declare-then-mutate pattern out of PHP/Python habit.

🎯 What's Next

Next chapter: **Methods & Blocks** — defining methods, default and keyword arguments, Ruby's implicit return, and a first proper look at blocks and `yield`.

🔧 Methods & Blocks

Chapter 3 used `.each do |n| ... end` without fully explaining what was happening. This chapter defines methods properly — arguments, implicit return, naming conventions — and then opens up what a block actually is: not a regular argument, but a distinct, implicit part of a Ruby method call unlike anything PHP or Python's callables offer directly.

📐 Defining a Method

```
def greet(name)
  "Hello, #{name}!"
end
greet("Philip")   # => "Hello, Philip!"
```

`def`...`end` defines a method, with no braces and no `function`/`def`-with-colon syntax to worry about beyond that. Notice there's no `return` keyword — the method's **last evaluated expression is automatically its return value**. This is Ruby's implicit return, and it's used constantly.

⚠ Implicit return means the LAST line matters — not the "obvious" one

Because Ruby returns whatever the final expression evaluates to, a method that ends with a `puts` call returns `nil` — `puts` always returns `nil`, regardless of what it printed. This trips up almost everyone coming from PHP or Python, where forgetting an explicit `return` in a similar spot would give a much more obvious "why is this None/null?" bug. In Ruby, it fails just as silently, but looks like working code at a glance.

The explicit `return` keyword still exists, and is the right choice for an early exit from inside a conditional:

```
def discount_price(price, is_member)
  return price unless is_member   # early exit -- return IS needed here

  price * 0.9 # implicit return -- this is the last line
end
```

🏷️ Naming Conventions: ? and !

Two method-naming conventions carry real meaning in Ruby and aren't just style preferences:

? — returns true/false

A method ending in `?` signals it returns a boolean, by convention (not enforced by the language). You've already seen this: `5.even?`, `nil.nil?`, `array.empty?`. PHP and Python have no equivalent naming convention — you'd typically prefix with `is_` instead.

! — "dangerous" / mutates in place

A method ending in `!` signals it mutates the receiver directly or is otherwise more "dangerous" than its non-`!` counterpart — `"ruby".upcase` returns a new string, while `"ruby".upcase!` modifies the original string in place. Always check what the non-`!` version does first; it's usually the safer default.

⚙️ Default Arguments

```
def greet(name = "World")
  "Hello, #{name}!"
end
greet # => "Hello, World!"  -- parentheses are optional on call too
greet("Philip")   # => "Hello, Philip!"
```

Default arguments work almost identically to PHP and Python — nothing unusual here. The more interesting difference is **keyword arguments**, next.

🔑 Keyword Arguments

```ruby
def build_profile(name:, age: 18)
  "#{name}, age #{age}"
end
build_profile(name: "Philip", age: 33)   # => "Philip, age 33"
build_profile(name: "Alex")              # => "Alex, age 18"  -- age uses its default
```

`name:` with no default is a **required** keyword argument — calling `build_profile` without it raises an `ArgumentError`. This mirrors the symbol-key hash shorthand from Chapter 2 for a reason: keyword arguments and symbol-keyed hashes share the same colon syntax deliberately.

Named Arguments: PHP vs Python vs Ruby

	PHP (8+)	Python	Ruby
Definition	function f($name, $age = 18)	def f(name, age=18):	def f(name:, age: 18)
Call site	f(name: "Philip")	f(name="Philip")	f(name: "Philip")
Can require by name?	No — still positional unless named	Yes, with keyword-only * marker	Yes — name: with no default is required

📥 Splat Args — Variable-Length Arguments

```
def sum_all(*numbers)  # like PHP's ...$args or Python's *args
  numbers.sum
end
sum_all(1, 2, 3, 4)  # => 10 -- numbers becomes [1, 2, 3, 4]
```

Ruby's single-splat `*args` and double-splat `**kwargs` (for collecting arbitrary keyword arguments into a hash) work almost exactly like Python's identical syntax — one of the closer matches between the two languages in this chapter.

🧊 Blocks — Not a Regular Argument

Here's the real departure from PHP and Python. In both of those languages, if you want to pass "a chunk of code" into a function, you pass it as a regular argument — a closure, an anonymous function, a lambda. In Ruby, every method call can carry an **implicit block** that isn't a normal argument at all:

```
def repeat(times)
 times.times do
yield # hands control to whatever block was passed in
end
end
repeat(3) { puts "Hi!" }   # prints "Hi!" three times
```

`yield` calls whatever block was attached to the current method call — the block itself doesn't appear anywhere in `repeat`'s parameter list. You can check whether a block was even given with `block_given?`, and pass values into the block by giving `yield` arguments:

```
def describe_each(items)
 return items unless block_given?   # graceful fallback with no block

 items.each do |item|
  yield item
 end
end
describe_each(["apple", "banana"]) do |fruit|
 puts "Fruit: #{fruit}"
end
```

Passing "a Chunk of Code": PHP vs Python vs Ruby

	PHP	Python	Ruby
Mechanism	Closure passed as a normal argument	Lambda/function passed as a normal argument	Implicit block, not a regular argument at all
Multi-statement?	Yes, closures can have a full body	No — lambdas are single-expression only	Yes, blocks can contain any number of statements
How the method uses it	Calls the closure variable directly, e.g. $callback()	Calls the function variable directly, e.g. callback()	yield — the block isn't a named variable at all

💎 This Is Only the Preview

Blocks can be converted into real objects (Procs and Lambdas) that *can* be stored in a variable and passed around explicitly — using the `&` operator you'll have seen hinted at once or twice already. That's genuinely the single most distinctive feature of Ruby as a language, and it gets a full chapter of its own (Chapter 8) once classes and modules are in place. For now, the goal is just recognizing `do...end`/`{ }` as a block, and `yield` as how a method hands control to one.

💻 Coding Challenges

Challenge 1: Implicit Return with Keyword Arguments

Write a method `full_name` that takes required keyword arguments `first:` and `last:`, and an optional keyword argument `middle:` defaulting to `nil`. Return the combined name (handling the case where `middle` is `nil`) using implicit return — no `return` keyword anywhere in the method.

Goal: Practice keyword arguments and trusting implicit return for the method's final value.

→ Solution

Challenge 2: Splat Args

Write a method `average` that accepts any number of numeric arguments using `*numbers` and returns their average. Handle the edge case of zero arguments by returning `0` instead of dividing by zero.

Goal: Get comfortable with `*args` collecting a variable number of arguments into an array.

→ Solution

Challenge 3: A Method That Yields

Write a method `with_timing` that takes no arguments, records nothing fancy (no real timer needed) but prints `"Starting..."`, then `yield`s, then prints `"Done!"`. Use `block_given?` to print `"No block given!"` instead if no block was passed. Call it once with a block and once without.

Goal: Write your own `yield`-based method instead of just using one Ruby provides.

→ Solution

🎯 What's Next

Next chapter: **Arrays & Hashes Deep Dive** — Enumerable methods like `map`, `select`, and `reduce`, which lean heavily on the block mechanics this chapter just introduced.

🧮 Arrays & Hashes Deep Dive

Chapter 4's `yield` and blocks weren't an isolated feature — they're the mechanism behind almost every method covered in this chapter. Ruby's **Enumerable** module gives both `Array` and `Hash` a shared set of block-powered methods for transforming, filtering, and reducing collections, and idiomatic Ruby leans on them far more heavily than PHP or Python lean on their equivalents.

🧩 One Module, Two Collection Types

Both `Array` and `Hash` include a module called **Enumerable** (modules get their own full chapter later, but the short version: a module is a bundle of methods that gets mixed into a class). That's why `map`, `select`, and `reduce` all work the same way on an array *and* a hash — they're not separately implemented for each type, they're inherited from the same shared source.

🔄 map — Transform Every Element

```ruby
numbers = [1, 2, 3, 4]

doubled = numbers.map { |n| n * 2 }
puts doubled   # => [2, 4, 6, 8]  -- a brand new array, numbers is untouched
```

`map` (aliased `collect`) runs the block once per element and returns a new array built from the block's return values — the same idea as PHP's `array_map` or a Python list comprehension, just spelled as a method call with a block instead of a separate function or comprehension syntax.

🔍 select and reject — Filter Elements

```
numbers = [1, 2, 3, 4, 5, 6]

numbers.select { |n| n.even? }   # => [2, 4, 6]  -- keep where block is true
numbers.reject { |n| n.even? }   # => [1, 3, 5]  -- keep where block is false, the opposite
```

`select` (aliased `filter`) keeps elements where the block returns something truthy; `reject` is its exact opposite. Neither PHP nor Python has a clean one-word "opposite of filter" — you'd write `array_filter($arr, fn($n) => !($n % 2 == 0))` in PHP, or `[n for n in nums if n % 2 != 0]` in Python, negating the condition yourself either way.

➕ reduce (aka inject) — Fold to a Single Value

```ruby
numbers = [1, 2, 3, 4]

total = numbers.reduce(0) { |sum, n| sum + n }
puts total  # => 10
# Shorthand: pass a symbol for the operator directly
numbers.reduce(:+)     # => 10  -- same result, no block needed at all
```

`reduce` (Ruby also accepts the alias `inject`) takes a starting value and a block that combines each element into a running accumulator — directly comparable to PHP's `array_reduce` or Python's `functools.reduce`. The `reduce(:+)` shorthand — passing a symbol naming the operator instead of a block — is Ruby-specific and worth recognizing even before it makes intuitive sense.

🔢 each_with_index — Iterate With a Counter

```ruby
fruits = ["apple", "banana", "cherry"]

fruits.each_with_index do |fruit, index|
  puts "#{index}: #{fruit}"
end
# 0: apple
# 1: banana
# 2: cherry
```

Iterating With an Index: PHP vs Python vs Ruby

	PHP	Python	Ruby
Syntax	foreach ($arr as $i => $v)	for i, v in enumerate(arr)	arr.each_with_index do |v, i|
Order of pair	index, value	index, value	value, index (reversed!)

⚠ The argument order is reversed from PHP/Python

`each_with_index` yields `|value, index|` — value first, index second. Coming from `enumerate()` or `$i => $v`, it's extremely easy to write `|index, fruit|` out of habit and get confused when the output looks scrambled. Double-check the order any time this method misbehaves.

⛓️ Method Chaining

Because every Enumerable method returns a new collection (or a single value, for `reduce`), they chain together into a readable pipeline — arguably the single most idiomatic thing you can do with Ruby collections:

```ruby
numbers = [1, 2, 3, 4, 5, 6]

result = numbers
  .select { |n| n.even? }
  .map { |n| n * 10 }
  .reduce(:+)

puts result   # => 120   ([2,4,6] -> [20,40,60] -> summed)
```

PHP and Python *can* chain similarly (Python especially, with generator expressions), but neither reads quite as close to plain English as a Ruby Enumerable chain — `select`, then `map`, then `reduce`, each step legible on its own line.

🔑 Hash Iteration

```ruby
prices = { apple: 1.50, banana: 0.75, cherry: 3.00 }

prices.each do |item, price|
  puts "#{item}: $#{price}"
end
# map on a Hash returns an Array by default
expensive = prices.select { |item, price| price > 1 }
puts expensive   # => {apple: 1.5, cherry: 3.0}  -- select keeps Hash shape
```

`.each do |key, value|` destructures each pair automatically — the block receives both the symbol key and its value in one step, similar to Python's `.items()` but without needing to call a separate method to get pairs in the first place (PHP's `foreach ($assoc as $key => $value)` is the closest direct match).

📜 Enumerable Method Equivalents

PHP vs Python vs Ruby

Operation	PHP	Python	Ruby
Transform	array_map()	map() / comprehension	.map
Filter	array_filter()	filter() / comprehension	.select
Filter (inverse)	Negate the condition manually	Negate the condition manually	.reject
Fold to one value	array_reduce()	functools.reduce()	.reduce / .inject
Index while iterating	$i => $v	enumerate()	.each_with_index

💻 Coding Challenges

Challenge 1: A select + map Pipeline

Given an array of integers from 1 to 20, chain `.select` and `.map` to produce a new array containing only the multiples of 3, each one squared. Print the result.

Goal: Practice chaining two Enumerable methods into one readable pipeline.

→ Solution

Challenge 2: reduce for a Shopping Total

Given an array of hashes, each with `:item` and `:price` keys, use `.reduce` to compute the total price of everything in the array. Try it both with an explicit block and with the `reduce(:+)` shorthand after first extracting just the prices with `.map`.

Goal: See `reduce` used on real-shaped data, not just a flat array of numbers.

→ Solution

Challenge 3: Hash Iteration and Filtering

Given a hash of student names (symbol keys) to test scores (integer values), use `.each` to print every student's pass/fail status (pass is 60+), then use `.select` to build a new hash containing only the students who passed.

Goal: Practice destructuring hash pairs in a block and filtering a hash while keeping its shape.

→ Solution

💎 Reach for Enumerable Before a Manual Loop

If you catch yourself writing `result = []` followed by a `.each` loop that pushes into it, pause — that's very often a `.map` or `.select` in disguise. Idiomatic Ruby treats the manual accumulator pattern as a code smell precisely because Enumerable already has a named, chainable method for almost every common transformation.

🎯 What's Next

Next chapter: **Classes & Objects** — defining your own classes, `initialize`, instance variables, and the `attr_accessor` family of shortcuts.

🏛️ Classes & Objects

Every chapter so far has used objects Ruby already provides — strings, arrays, hashes. This chapter shows how to build your own. Ruby's class syntax will feel broadly familiar coming from PHP or Python, but two details are genuinely different: how instance variables are written, and how little ceremony Ruby needs to turn them into public getters and setters.

 Defining a Class and initialize

```
class Person
def initialize(name, age)
  @name = name
  @age = age
 end
end
philip = Person.new("Philip", 33)
```

`initialize` is Ruby's constructor — it runs automatically whenever `.new` is called, the same role as PHP's `__construct` or Python's `__init__`. Class names are capitalized by convention (**CamelCase**, not snake_case) — this isn't just style, Ruby actually treats capitalized identifiers as constants, and class names are constants.

📌 Instance Variables — the @ Sigil

This is the first genuinely new syntax in this chapter. Ruby instance variables are written with a leading `@`, directly on the variable name — no `$this->` and no `self.` required to declare or read one from inside an instance method:

```
class Person
def initialize(name)
  @name = name
 end
def greet
"Hi, I'm #{@name}" # @name is read directly -- no self./this-> prefix needed
end
end
```

Instance Variables: PHP vs Python vs Ruby

	PHP	Python	Ruby
Declaring/setting	$this->name = $name;	self.name = name	@name = name
Reading inside a method	$this->name	self.name	@name
Requires an explicit receiver?	Yes — $this->	Yes — self.	No — @ is the whole syntax

⚠ An uninitialized instance variable is nil, not an error

Referencing `@nickname` before it's ever been assigned doesn't raise an error the way accessing an undeclared property might elsewhere — it silently evaluates to `nil`. This is convenient for quick scripts, but it means a typo in an instance variable name (`@nmae` instead of `@name`) fails silently instead of raising a clear error, which can be a genuinely confusing bug to track down.

attr_accessor, attr_reader, attr_writer

By default, instance variables are private to the object — there's no automatic public getter or setter the way some languages provide. Writing them by hand gets repetitive fast:

```
# The manual way
class Person
def initialize(name)
  @name = name
 end
def name # manual getter
@name
end
def name=(value)  # manual setter -- note the = is part of the method name
@name = value
 end
end
```

Ruby collapses that entire pattern into one line:

```
class Person
attr_accessor :name, :age # generates BOTH a getter and setter for each
def initialize(name, age)
  @name = name
  @age = age
 end
end
philip = Person.new("Philip", 33)
philip.name       # => "Philip"  -- the generated getter
philip.name = "P." # the generated setter
```

attr_reader — getter only

`attr_reader :name` generates only the getter, leaving the instance variable read-only from outside the class. Use this for values that shouldn't be changed after construction — an ID, a creation timestamp, anything meant to stay fixed.

attr_writer — setter only

`attr_writer :password` generates only the setter, with no way to read the value back out through a public method. Rare in practice, but useful for write-only style properties.

Getters/Setters: PHP vs Python vs Ruby

	PHP	Python	Ruby
Approach	Write getName()/setName() by hand, or PHP 8.1+ readonly properties	@property / @name.setter decorators	attr_accessor / attr_reader / attr_writer
Lines needed for a full getter+setter	~6-10	~6-8	1 (attr_accessor :name)

Customizing to_s

Every Ruby object has a default `to_s` that produces an unhelpful string like `#<Person:0x00007f8b1a0b8c a8>`. Override it to control how an object prints — the equivalent of PHP's `__toString` or Python's `__str__`:

```
class Person
attr_accessor :name, :age
def initialize(name, age)
  @name = name
  @age = age
 end
def to_s
"#{@name} (#{@age})"
end
end
puts Person.new("Philip", 33)  # => Philip (33)  -- puts calls to_s automatically
```

⚡ Class Methods — def self.method_name

A method defined with `self.` belongs to the class itself, not to individual instances — the equivalent of PHP's `static` methods or Python's `@staticmethod`/`@classmethod`:

```
class Person
def self.species
"Homo sapiens"
end
end
Person.species   # => "Homo sapiens"  -- called on the class, no instance needed
```

💻 Coding Challenges

Challenge 1: A Book Class

Write a `Book` class with `attr_accessor` for `:title`, `:author`, and `:pages`, plus an `initialize` that sets all three. Create two instances and print each one's title using the generated getter.

Goal: Get comfortable with the initialize + attr_accessor pattern that appears in nearly every Ruby class.

→ Solution

Challenge 2: Custom to_s

Add a `to_s` method to your `Book` class from Challenge 1 that returns a formatted string like `"'Title' by Author (pages pp.)"`. Use `puts` on an instance directly to confirm it's picked up automatically.

Goal: Practice overriding a default Ruby method to control an object's string representation.

→ Solution

Challenge 3: A Class Method Counter

Modify the `Book` class to track how many books have been created: add a class variable (research `@@count` briefly, or use a class-level instance variable with `self.count`) that increments inside `initialize`, and a class method `self.total_books` that returns the current count.

Goal: See class-level state and a class method working together, distinct from any single instance.

→ Solution

💎 Default to attr_accessor, Narrow When You Have a Reason

It's tempting to always reach for `attr_accessor` since it's the shortest to type, but the moment a value shouldn't be freely rewritten from outside the class — an ID, a total that should only change through a specific method — switch to `attr_reader` and add an explicit method for any controlled mutation. Default to the more restrictive option and only open things up when you actually have a reason to.

🎯 What's Next

Next chapter: **Modules & Mixins** — `include` vs `extend`, and how Ruby gets the benefits of multiple inheritance without actually allowing a class to have more than one superclass.

🧩 Modules & Mixins

Ruby classes support only **single inheritance** — a class has exactly one superclass, full stop. Chapter 5 mentioned, without fully explaining, that `Array` and `Hash` both share the same `Enumerable` methods despite having no inheritance relationship to each other. This chapter explains how: modules, and the `include`/`extend` mechanism Ruby uses instead of true multiple inheritance.

🧬 One Superclass, No Exceptions

```ruby
class Dog < Animal # fine -- one superclass
end
# There is no syntax for this in Ruby at all:
# class Dog < Animal, Pet
```

Inheriting From Multiple Sources: PHP vs Python vs Ruby

	PHP	Python	Ruby
True multiple inheritance?	No — single class inheritance	Yes — class C(A, B): is valid	No — single class inheritance
Shared-behavior workaround	Traits (use TraitName;)	Multiple inheritance directly, or mixins by convention	Modules, mixed in with include/extend

This table has a genuine surprise in it: **Python actually allows true multiple inheritance** directly (`class C(A, B):`), something neither PHP nor Ruby permits. PHP's answer is **traits**. Ruby's answer is **modules** — and while the underlying goal (share behavior across otherwise-unrelated classes) is the same as PHP traits, Ruby's mechanism looks different enough to be worth learning on its own terms.

📦 What Is a Module?

A **module** is defined like a class, but it can never be instantiated — there's no `Module.new`. It's purely a named container for methods (and constants) meant to be shared:

```
module Greetable
def greet
"Hello, I'm #{name}!" # assumes the including class defines a name method
end
end
```

➕ include — Mix In Instance Methods

```
module Greetable
def greet
"Hello, I'm #{name}!"
end
end
class Person
include Greetable
attr_accessor :name
def initialize(name)
  @name = name
 end
end
class Robot
include Greetable # completely unrelated to Person -- no shared superclass
attr_accessor :name
def initialize(name)
  @name = name
 end
end
Person.new("Philip").greet   # => "Hello, I'm Philip!"
Robot.new("R2D2").greet    # => "Hello, I'm R2D2!"
```

`include` mixes a module's methods into a class as **instance methods** — every instance of `Person` and `Robot` now has `greet` available, even though the two classes share no common ancestor besides `Object`. This is the direct Ruby equivalent of PHP's `use TraitName;` inside a class body.

🔌 extend — Mix In Class Methods

Where `include` adds instance methods, `extend` adds the module's methods as **class methods** instead — an alternative to writing `def self.method_name` directly, from Chapter 6:

```
module Identifiable
def class_id
"ID: #{name}" # name here refers to the class's own name
end
end
class Product
extend Identifiable
end
Product.class_id  # => "ID: Product"  -- called on the class, not an instance
```

include → instance methods

Methods become available on *objects* created from the class — `Person.new(...).greet`. This is by far the more common of the two in everyday Ruby code.

extend → class methods

Methods become available on the *class itself* — `Product.class_id`, no instance involved. Reach for this specifically when the shared behavior is about the class as a whole, not about individual instances of it.

🔮 The Payoff: Enumerable, Revealed

Chapter 5's `map`, `select`, and `reduce` now make complete sense as a mechanism, not just a fact to memorize. Ruby's actual standard library source is, in simplified spirit, doing exactly this:

```
class Array
include Enumerable # (simplified -- this is conceptually what's happening)
end
class Hash
include Enumerable # same module, completely different class
end
```

`Array` and `Hash` share no inheritance relationship whatsoever — one isn't a subclass of the other. They both get `map`/`select`/`reduce` for the exact reason `Person` and `Robot` both got `greet` above: the same module, mixed into two otherwise-unrelated classes.

💻 Coding Challenges

Challenge 1: A Loggable Module

Write a module `Loggable` with a method `log(message)` that prints `"[LOG] #{message}"`. Create a class `Order` that includes `Loggable`, instantiate it, and call `.log("Order created")` on the instance.

Goal: Write your first module and mix it into a class with `include`.

→ Solution

Challenge 2: Shared Behavior, No Common Ancestor

Reusing the `Loggable` module from Challenge 1, create a second, completely unrelated class `Invoice` (no shared superclass with `Order`) that also includes `Loggable`. Call `.log` on an instance of each class to prove both work identically.

Goal: See directly that `include` shares behavior without requiring any inheritance relationship between the classes.

→ Solution

Challenge 3: extend for a Class-Level Method

Write a module `Describable` with a method `describe` that returns `"This is the #{name} class"`. `extend` it into a class of your choice and call `.describe` directly on the class (not an instance).

Goal: Feel the difference between `include` (instance methods) and `extend` (class methods) firsthand.

→ Solution

💎 Why Ruby Avoided True Multiple Inheritance

True multiple inheritance (Python's approach) runs into the classic "diamond problem" — if two parent classes both define a method with the same name, which one wins? Python resolves this with a defined, sometimes-surprising resolution order (MRO). Ruby's designers sidestepped the ambiguity entirely: a class still has exactly one superclass, and mixed-in modules are inserted into a single, well-defined **ancestor chain** (inspectable with `SomeClass.ancestors`) — so there's always one unambiguous answer for which `greet` method actually runs.

🎯 What's Next

Next chapter: **Blocks, Procs & Lambdas** — the final chapter of Ruby Fundamentals, taking the block/`yield` mechanics from Chapter 4 the rest of the way and turning blocks into objects you can store, pass around, and call explicitly.

🎭 Blocks, Procs & Lambdas

Chapter 4 introduced blocks and `yield`, but was upfront that a block isn't a real object — it can't be stored in a variable or passed around explicitly, only attached to a single method call. This final chapter of Ruby Fundamentals removes that limitation: turning blocks into genuine objects (Procs), meeting their stricter cousin (Lambdas), and closing out the single feature most responsible for Ruby's reputation as an unusually expressive language.

🧊 Recap: a Block Alone Isn't an Object

```
[1, 2, 3].each { |n| puts n }   # the { |n| puts n } part has no independent existence
# This doesn't work -- a bare block isn't a value you can assign:
# my_block = { |n| puts n }   # SyntaxError
```

📦 Proc — a Block You Can Store

A **Proc** wraps a block of code into a real object, assignable to a variable and callable whenever you want with `.call`:

```
say_hi = Proc.new { |name| puts "Hi, #{name}!" }

say_hi.call("Philip")   # => Hi, Philip!
say_hi.("Alex")         # => Hi, Alex!   -- shorthand for .call
say_hi["Jordan"]        # => Hi, Jordan! -- another shorthand for .call
```

🔗 The & Operator — Converting Between Blocks and Procs

The `&` operator you've glimpsed once or twice now converts in both directions: capture a method's incoming block as a named Proc parameter, or expand a stored Proc back out into a block when calling another method:

```
def run_twice(&block)  # ▌captures the incoming block as a real Proc
  block.call
  block.call
end
run_twice { puts "Hey!" }  # Hey! / Hey!  -- prints twice
my_proc = Proc.new { puts "From a stored Proc" }
[1, 2].each(&my_proc)  # &my_proc expands the Proc back into a block for .each
```

⚡ Symbol#to_proc — the &:method_name Shorthand

This idiom shows up constantly in real Ruby code, and it's a direct application of `&` converting a value into a block:

```
names = ["philip", "alex", "jordan"]

# Long form
names.map { |name| name.upcase }

# Shorthand -- identical result
names.map(&:upcase)   # => ["PHILIP", "ALEX", "JORDAN"]
```

`:upcase` is a symbol (Chapter 2), and every symbol has a built-in `to_proc` method that converts it into a Proc calling that method name on whatever it's given. `&:upcase` then expands that Proc into a block — three separate features from earlier chapters (symbols, blocks, `&`) combining into one extremely common one-liner.

🎯 Lambda — a Stricter Proc

```
# Two equivalent ways to write a lambda
add = lambda { |a, b| a + b }
add = ->(a, b) { a + b }   # "stabby lambda" syntax -- more common in modern Ruby

add.call(2, 3)   # => 5
```

A lambda is technically a kind of Proc, but with two behavioral differences that matter enough to treat it as its own concept:

Strict argument checking

Calling a lambda with the wrong number of arguments raises an `ArgumentError`, exactly like a normal method. A regular Proc is lenient — extra arguments are silently dropped, and missing ones are filled in with `nil`, no error at all.

return behaves differently

`return` inside a lambda exits just the lambda, the same as returning from an ordinary method. `return` inside a plain Proc attempts to return from the *enclosing method* the Proc was created in — a subtle, genuinely surprising gotcha if you don't know to expect it.

⚠ return inside a Proc can crash your program

If a Proc containing `return` is called after the method that created it has already finished executing, Ruby raises a `LocalJumpError` — there's no enclosing method left to return from. This exact failure mode is a common reason experienced Ruby developers default to lambdas over plain Procs whenever `return` might appear inside the block.

📜 Block vs Proc vs Lambda

The Full Picture

	Block	Proc	Lambda
Storable in a variable?	No	Yes	Yes
Created with	do...end or { } after a method call	Proc.new { } / proc { }	lambda { } / -> ( ) { }
Argument count checking	Lenient	Lenient	Strict (raises ArgumentError)
return behavior	N/A (tied to yield)	Returns from the enclosing method	Returns from the lambda itself

📐 Closing the Loop with PHP and Python

Passing "a Chunk of Code," Revisited

	PHP	Python	Ruby
Anonymous callable	function() {} or fn() =>	lambda: expr (single expression only)	Block, Proc, or Lambda — three distinct options
Multi-statement anonymous code?	Yes, closures support full bodies	No — Python lambdas are expression-only	Yes — all three (block, Proc, lambda) support full bodies
Implicit, un-named argument to every call?	No — always an explicit parameter	No — always an explicit parameter	Yes — the block (Chapter 4)

Python's lambdas are deliberately limited to a single expression, which is actually the opposite trade-off from Ruby's — Ruby's lambdas are just as full-featured as ordinary methods, and it's the plain *block* (not the lambda) that's the odd one out, precisely because a block isn't a regular argument at all.

💻 Coding Challenges

Challenge 1: Capture and Call a Block

Write a method `repeat_message(times, &block)` that calls the given block `times` times, passing the current iteration number (starting at 1) into the block each time. Call it with a block that prints `"Attempt #{n}"`.

Goal: Practice capturing an incoming block as a named Proc with `&` and calling it explicitly with arguments.

→ Solution

Challenge 2: Symbol#to_proc in Practice

Given an array of mixed-case strings, use the `&:downcase` shorthand with `.map` to lowercase every element in one line. Then do the same thing again using a full block, and compare the two lines.

Goal: Get comfortable recognizing and writing the `&:method_name` idiom.

→ Solution

Challenge 3: Proc vs Lambda Arity

Create a Proc and a Lambda that each expect exactly two arguments but simply return their sum. Call each one with only one argument. Observe (and write down in a comment) what happens differently — one should silently produce a strange result, the other should raise an error.

Goal: Directly witness the strict-vs-lenient argument checking difference instead of just reading about it.

→ Solution

💎 Why This Chapter Closes the Course

Blocks, Procs, and Lambdas aren't a niche feature — they're the mechanism underneath nearly every piece of "magic" Ruby is known for, including Ruby on Rails' routing DSL, its database migrations, and RSpec's `describe`/`it` testing syntax (all bucket-listed as possible follow-ups to this course). Every one of those reads like a small custom language built on top of Ruby, and every one of them is built from exactly the same block-passing mechanics this chapter just covered.

🎯 What's Next

Course 1 (Ruby Fundamentals) is complete. Ruby Intermediate/Advanced (Course 2) begins with **Exception Handling** — `begin`/`rescue`/`ensure`/`raise`, and writing custom exception classes.