CAPABILITY-TYPED · STATICALLY CHECKED

Capa

The Capability-Typed Programming Language

Every function declares what it is allowed to do.

Nelson Duarte

Capa

The Capability-Typed Programming Language

Every function declares what it is allowed to do.

Nelson Duarte

Capa - The Capability-Typed Programming Language

First edition, 2026.

Written by Nelson Duarte.

Project: capa-language.com

Contact: nelson.duarte31@gmail.com

ABOUT THE AUTHOR

Nelson Duarte

Nelson Duarte is a computer-science teacher. His classes range across programming in Python and C#, and, as the need arises, cybersecurity, computer networks, and the administration of Linux and Windows systems. Years of explaining ideas to people seeing them for the first time shaped both how he teaches and how he writes: patiently, concretely, one small example at a time.

He is also the creator of Capa. The language grew out of a question that kept returning in his security and systems classes, why does software get to do things it never announces, and the conviction that a programming language could answer it. In Capa, every function must declare the authority it holds, and the compiler checks the claim, turning “this code is safe” from a hope defended in review into a property a machine can verify.

He wrote this book to bring that idea within reach of complete beginners, in the same style he uses in the classroom: detailed theory, runnable examples, and exercises to try at every step. He lives and teaches in Portugal.

CONTENTS

PART I

The Basics

The Capa language, from your first program to testing your code.

CHAPTER 1

Getting Started

Writing your first program is a small ceremony. You install a tool, you type a few lines into a file, you ask the computer to run them, and a message appears on the screen. Nothing about that moment looks dramatic, yet it is the doorway to everything else in this book. In this chapter you will walk through that doorway with **Capa**, the programming language this book is about.

We assume nothing. If you have never written a line of code, you are in exactly the right place. We will explain what a program is, set up the few tools you need, and run a working Capa program together before the chapter is over.

What Is a Program, and What Is Capa?

A **computer program** is a set of instructions that tells a computer what to do, written in a language the computer can be made to understand. Humans do not speak the computer's native language directly, so we write our instructions in a **programming language**: a precise, readable notation with a fixed set of rules. A piece of software called a **compiler** (or an interpreter) then turns what we wrote into something the machine can actually execute.

Capa is one such language. Its syntax, the way the code looks, is deliberately close to Python, so it reads cleanly and is friendly to newcomers. What makes Capa special is a single idea: **every function must declare what it is allowed to do**. A piece of Capa code that wants to read a file has to say so, in plain sight, in its signature. A piece of code that does not say so simply cannot touch the filesystem, no matter what it tries. These powers, the right

to use the screen, the files, the network, and so on, are called **capabilities**, and they are the heart of the language.

> **NOTE**
>
> Do not worry if capabilities sound abstract right now. For the first several chapters Capa will look like an ordinary, pleasant language, and that is on purpose. The capability idea arrives gently in Chapter 9, once the basics are comfortable. For now, just know it is there, and that it is the reason the language exists.

Setting Up Your Computer

To write and run Capa programs you need three things: the **Capa tool** itself, a **text editor** to write code in, and a **terminal** to run commands from. Let us take them in order.

What Is a Terminal?

A **terminal** (also called a *command line* or *shell*) is a window where you type text commands and the computer responds with text. It feels old-fashioned next to clicking icons, but it is the most direct way to talk to your tools, and every Capa program in this book is launched from one. You do not need to be an expert; you only need to know how to open it and type a command.

- On **Windows**, open the Start menu, type *Terminal* or *PowerShell*, and press Enter.
- On **macOS**, press Cmd-Space, type *Terminal*, and press Enter.
- On **Linux**, look for an app called *Terminal*, or press Ctrl-Alt-T on many systems.

Throughout the book, a line that begins with `$` (on macOS and Linux) or `PS>` (on Windows PowerShell) means *type this into the terminal*. You do not type

the `$` or `PS>` themselves; they only mark where the command starts. Lines without a prompt show what the computer prints back.

Installing Capa

Capa is distributed as a small self-contained program. The simplest way to get it is the one-line installer, which downloads the latest version and places it where your terminal can find it. Open your terminal and run the command for your system.

On **Linux** or **macOS**:

```
$ curl -fsSL https://github.com/nelsonduarte/capa-
language/releases/latest/download/install.sh | bash
```

On **Windows**, using PowerShell:

```
PS> irm https://github.com/nelsonduarte/capa-
language/releases/latest/download/install.ps1 | iex
```

If PowerShell refuses to run the script and shows an execution-policy error, it is blocking downloaded scripts for safety. Allow signed scripts once with the command below, answer Y to confirm, then run the install command again. Adding -Scope CurrentUser limits the change to your own account, so it needs no administrator rights.

```
PS> Set-ExecutionPolicy RemoteSigned
```

The installer does not require administrator rights and makes no system-wide changes; it simply drops the `capa` program into a folder for your user account. When it finishes, **close the terminal and open a new one** so it picks up the change, then check that everything worked:

```
$ capa --version
capa 1.12.0
```

If you see a version number, Capa is installed and ready. If instead you see an error like *command not found*, see the *Troubleshooting* section at the end of this chapter before continuing. Your installed version may be newer than 1.12.0; anything at or above 1.12.0 works for this book.

> **NOTE On the PATH**
>
> The terminal finds programs by looking through a list of folders called the **PATH**. The installer puts `capa` in a standard user folder, but on some systems that folder is not yet on the PATH, which is what causes *command not found*. On Windows the installer adds capa to the PATH for you; on Linux and macOS, when that folder is not yet on the PATH the installer prints the exact one-line fix to apply, copy it, paste it, open a new terminal, and try `capa --version` again.

Installing a Text Editor

Code is just text, so in principle any text editor works. In practice a good **code editor** makes life far easier by colouring your code, pointing out mistakes, and keeping things tidy. This book uses **Visual Studio Code** (VS Code) in its examples because it is free, runs on every major system, and works well with Capa, but any editor you like is fine.

1. Go to *https://code.visualstudio.com* in your web browser.
2. Download the version for your operating system and install it like any other app.
3. Open VS Code once to make sure it launches.

Capa ships an extension for VS Code that adds colour highlighting and live error checking for `.capa` files. It is optional for now and we will return to editor setup later; a plain editor is more than enough to follow this chapter.

Running Your First Program

The tradition, in every language, is that your first program prints the words *Hello, world!* to the screen. We will honour it. There are two easy ways to get there: let Capa create a small project for you, or write a single file by hand. We will do both, starting with the file by hand, because it shows you every moving part.

Writing hello.capa

Create a new folder somewhere convenient, for example a folder called `capa-book` in your home directory. Open that folder in VS Code (*File -> Open Folder*), create a new file, and name it `hello.capa`. The `.capa` ending tells everyone, you, your editor, and the Capa tool, that this file contains Capa code. Type the following two lines into it exactly as shown:

```
// hello.capa
fun main(stdio: Stdio)
    stdio.println("Hello, Capa!")
```

Save the file. Those three lines already contain several important ideas, so let us read them slowly.

The first line, beginning with `//`, is a **comment**. Anything after `//` on a line is ignored by the computer; comments exist purely for humans, to leave notes in the code. Here it simply records the file's name.

The second line, `fun main(stdio: Stdio)`, defines a **function**. A function is a named block of instructions. The keyword `fun` starts the definition, and `main` is the function's name. The name `main` is special: when you run a Capa program, the computer looks for a function called `main` and starts there. It is the front door of every program.

Inside the parentheses sits `stdio: Stdio`. This is where Capa shows its true colours. It says that `main` is asking for one capability, the **standard output**

capability, named `Stdio`, and that within this function we will refer to it as `stdio`. *Standard output* is the technical name for the ordinary stream of text a program writes to the screen. In most languages, printing to the screen is a power every piece of code silently holds. In Capa, you must ask for it, out loud, in the signature. No `Stdio`, no printing.

The third line, indented underneath, is the function's **body**, the actual work. `stdio.println("Hello, Capa!")` uses the `stdio` capability we were handed and calls its `println` method, which prints a line of text. The text to print, `"Hello, Capa!"`, is written between double quotes; a run of text like this is called a **string**.

> **NOTE Indentation matters**
>
> Notice that the third line is indented (shifted to the right) underneath `main`. In Capa, like in Python, **indentation is part of the grammar**: it is how the language knows that the `println` line belongs *inside* `main`. Use spaces consistently (four spaces per level is the convention) and do not mix in random tabs. Your editor will help you keep this tidy.

Running It

Now bring the program to life. In your terminal, move into the folder that contains `hello.capa` and ask Capa to run it:

```
$ capa --run hello.capa
Hello, Capa!
```

There it is: your instructions, executed, and the message printed back to you. The `--run` part is a **flag**, an option that tells the `capa` tool what you want it to do, in this case *transpile this file and run it immediately*. We will meet the other flags shortly.

> **NOTE Moving into a folder**
>
> To move into a folder in the terminal, use the `cd` command (it stands for *change directory*), followed by the folder's path, for example `cd capa-book`. Type `cd` on its own to return to your home folder. If `capa --run hello.capa` reports that it cannot find the file, you are almost certainly in the wrong folder; `cd` into the one where you saved it and try again.

Letting Capa Scaffold a Project

Once you are writing more than a single file, it helps to start from a ready-made project skeleton. The `capa init` command builds one for you:

```
$ capa init my-project
$ cd my-project
$ capa --run main.capa
Hello from Capa!
```

Read that first block as a three-step routine you will reach for often: capa init my-project makes a new folder called my-project and fills it with a small, ready-to-run skeleton; cd my-project moves into that folder; and capa --run main.capa runs the program it generated, which already prints a greeting. You can use any name in place of my-project, and that name becomes the folder's name.

Inside the new folder are three starter files:

```
my-project/
  main.capa      # the program; starts in fun main(stdio: Stdio)
  README.md      # a place for notes about the project
  .gitignore     # files the Git tool should ignore
```

The one that matters most is main.capa, the program's entry point: it already declares fun main(stdio: Stdio), so the capability discipline, asking for the authority to print rather than assuming it, is visible from the very first line.

README.md is an empty place for your own notes, and .gitignore lists files that the Git version-control tool should leave alone, useful once you start tracking your project's history. For the rest of Part I we will mostly write single files by hand, but it is good to know the shortcut exists.

How Capa Runs Your Code

It is worth knowing, even this early, what happens between you typing `capa --run` and the message appearing. Capa processes your file in stages, each one building on the last, and you can stop at any stage to see what it produced. This is a wonderful learning aid: when something goes wrong, you can ask Capa to show its work.

- **Tokenize.** Capa first chops the raw text into small meaningful pieces called *tokens*, the words, symbols, and numbers of your program. Running `capa` with no flag prints this token stream.
- **Parse** (`--parse`). It then arranges those tokens into a tree that mirrors the structure of your code, functions containing statements containing expressions.
- **Check** (`--check`). Next it inspects that tree for mistakes: undefined names, type mismatches, and crucially the capability rules. If your code is sound, it prints `OK`; otherwise it prints precise, friendly errors.
- **Transpile** (`--transpile`). Capa then translates your program into equivalent Python code. You will rarely need this, but it can be illuminating to see how a Capa construct is expressed underneath.
- **Run** (`--run`). Finally it executes the translated program. This is the everyday flag, and each flag in this list quietly includes the ones above it.

The most useful of these in daily work is `--check`. It runs the full analysis without executing anything, so it is the fastest way to ask *is my code valid?* Try it on your hello program:

```
$ capa --check hello.capa
```

```
OK
```

A clean `OK` means your program passed every rule the language enforces, including the promise that it only uses the capabilities it declared.

Troubleshooting

If something did not work, do not be discouraged; setup hiccups are routine and almost always quick to fix. Here are the most common ones.

"capa: command not found" or "not recognized"

The terminal cannot find the `capa` program. This usually means its folder is not on your PATH yet. Re-read the *On the PATH* note above. On Windows the installer adds capa to the PATH for you, so usually it is enough to open a fresh terminal; on Linux and macOS, apply the one-line fix the installer suggested, then **open a brand-new terminal window** (changes to the PATH only affect terminals opened afterwards) and run `capa --version` again. Appendix A lists the exact commands for Linux, macOS, and Windows, and shows how to check where capa was installed.

"No such file or directory" when running a file

Capa cannot find `hello.capa`. You are probably in a different folder from the one where you saved it. Use `cd` to move into the right folder, and confirm the file is there by listing the folder's contents (`ls` on macOS and Linux, `dir` on Windows).

The program runs but prints nothing

Check that the `println` line is indented underneath `main`, that the text is wrapped in double quotes, and that you saved the file before running it. A surprising number of "it does nothing" moments are simply an unsaved file.

NOTE When in doubt, ask the checker

Run `capa --check yourfile.capa` whenever you are unsure. Capa's error messages point at the exact line and often suggest the fix, including *did you mean...?* hints for misspelled names. Reading them carefully will teach you the language faster than anything else.

Try It Yourself

These short exercises ask you to apply what you just learned. Type real code, run it, and read whatever Capa tells you. The goal is fluency with the tools, not cleverness.

Exercise 1-1 Hello, You

Modify `hello.capa` so it prints a greeting with your own name, for example `Hello, Ana!`. Run it with `capa --run hello.capa` and confirm your message appears.

Exercise 1-2 Two Lines

Add a second `stdio.println(...)` line to `main`, indented the same way as the first, so the program prints two separate lines of text. Run it and observe that both lines appear, in order.

Exercise 1-3 Break It on Purpose

Delete the `stdio: Stdio` part from the signature, leaving `fun main()`, and run `capa --check hello.capa`. Read the error message Capa gives you, then put the capability back. This is your first glimpse of the capability discipline refusing to let a function print without permission.

Exercise 1-4 Scaffold a Project

Use `capa init practice` to create a new project, `cd` into it, and run `main.capa`. Open the generated files in your editor and read them; you will recognise the same `fun main(stdio: Stdio)` shape you wrote by hand.

Summary

You set up everything you need to write Capa: the `capa` tool, a code editor, and a terminal you can run commands from. You wrote and ran your first program, and you met the shape of every Capa program, a `main` function that declares the capabilities it needs and does its work in an indented body. You also saw how Capa processes a file in stages, and how `capa --check` reports problems before you ever run the code. Most importantly, you got your first taste of the one idea that defines the language: to print, `main` had to ask for `Stdio` out loud. In the next chapter we slow down and look closely at the raw materials every program is built from, values and the simple types that describe them.

CHAPTER 2

Values and Simple Types

Every program is, at bottom, a machine for moving and reshaping **data**. A calculator works with numbers, a chat app with text, a game with positions and scores. Before you can do anything interesting, you need to know how to write down pieces of data and how to give them names so you can refer to them later. That is what this chapter is about: the raw materials of Capa programs, and the handful of simple types that describe them.

We will keep running real code. Create a file called `values.capa` and, as each example appears, type it into `main` and run it with `capa --run values.capa`. Reading about values teaches you a little; watching them print teaches you a lot more.

Values, Types, and Names

A **value** is a single piece of data: the number `42`, the text `"hello"`, the truth value `true`. Every value in Capa has a **type**, a label that says what kind of thing it is and what you can do with it. `42` is an `Int` (an integer), `"hello"` is a `String` (text), `true` is a `Bool` (a yes/no value). Types are not decoration: they are how Capa catches mistakes before your program ever runs, such as trying to do arithmetic on a word.

To work with a value beyond the instant you write it, you give it a name. A named value is called a **binding** (you may also hear *variable*). You create one with the keyword `let`:

```
fun main(stdio: Stdio)
    let name = "Capa"
    let year = 2026
    stdio.println("${name} was released in ${year}.")
```

```
$ capa --run values.capa
Capa was released in 2026.
```

Read `let name = "Capa"` as *let the name `name` stand for the value `"Capa"`*. After that line, anywhere you write `name`, Capa substitutes the text `"Capa"`. Notice that we never told Capa that `name` is a `String` or that `year` is an `Int`; it figured that out from the values themselves. That is called **type inference**, and we will come back to it.

Naming Your Bindings

Names follow a few simple rules. A name may contain letters, digits, and the underscore `_`, but it must **start** with a letter or an underscore, never a digit. Names are case-sensitive, so `score` and `Score` are two different names. By convention, Capa names use **snake_case**: all lowercase, with underscores between words, like `first_name` or `total_score`.

- Good: `age`, `user_name`, `total_2024`, `_internal`.
- Not allowed: `2fast` (starts with a digit), `user-name` (the dash is the minus operator).

Choose names that say what the value *means*. `let d = 86400` tells the reader nothing; `let seconds_per_day = 86400` tells them everything. Good names are the cheapest documentation you will ever write.

let vs. var: Things That Change, Things That Don't

Capa gives you two keywords for naming values, and the difference between them is one of the first habits worth forming. A `let` binding is **immutable**: once it is set, it never changes. If you try to reassign it, the program will not even compile.

```
fun main(stdio: Stdio)
    let n = 42
    n = 7   // error: 'n' is bound with `let` and cannot be
reassigned
```

When you genuinely need a value that changes over time, use `var` instead. A `var` binding is **mutable**: you can assign new values to it as the program runs.

```
fun main(stdio: Stdio)
    var counter = 0
    counter = counter + 1
    counter = counter + 1
    stdio.println("counter is ${counter}")
```

```
$ capa --run values.capa
counter is 2
```

One thing stays fixed even for a `var`: its **type**. A binding that started life as an `Int` is an `Int` forever; only the value inside it can change, not the kind of value. Assigning text to a numeric `var` is an error:

```
var counter = 0
counter = "one"   // error: expected Int, got String
```

> **NOTE Prefer let**
>
> Reach for `let` by default, and only switch to `var` when a value truly needs to change, typically loop counters and running totals. Immutable bindings are easier to reason about because you can be certain they never shift under your feet, and Capa treats a `let` as a clear signal to anyone reading your code.

The Six Primitive Types

Capa is built on six **primitive types**, the simplest building blocks from which everything else is assembled:

- `Int` - whole numbers, positive or negative: `42`, `-7`, `0`.
- `Float` - numbers with a decimal point: `3.14`, `-0.5`, `1e10`.
- `Bool` - a truth value, either `true` or `false`.
- `String` - text, written between double quotes: `"Capa"`.
- `Char` - a single character, written between single quotes: `'A'`.
- `Unit` - the "empty" value, written `()`, used when there is nothing to return.

We will spend the rest of the chapter getting comfortable with numbers and strings, the two you will reach for most, and then meet the others briefly.

Numbers

Capa has two number types. An `Int` is a whole number with no fractional part. A `Float` carries a decimal point. The distinction matters in Capa more than in many languages, as we will see in a moment.

Arithmetic

The usual operators are available: `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `%` for the **remainder** (also called *modulo*), which gives what is left over after a division.

```
fun main(stdio: Stdio)
    stdio.println("${5 + 3}")     // 8
    stdio.println("${5 - 3}")     // 2
    stdio.println("${5 * 3}")     // 15
    stdio.println("${17 % 5}")    // 2  (17 = 3*5 + 2)
    stdio.println("${9.0 / 2.0}") // 4.5
```

Operator precedence works the way it does in mathematics: `*`, `/`, and `%` bind more tightly than `+` and `-`, and you can use parentheses to group sub-expressions and make your intent unmistakable, as in `(2 + 3) * 4`.

Ints and Floats Do Not Mix by Accident

Here is a rule that surprises newcomers but quickly becomes something you appreciate: Capa performs **no automatic conversion** between `Int` and `Float`. Adding one of each is a compile-time error, not a silent guess.

```
let result = 1 + 1.0   // error: cannot mix Int and Float
```

To combine them, you convert explicitly with the built-in functions `to_float` (turn an `Int` into a `Float`) and `to_int` (turn a `Float` into an `Int`, discarding the fractional part). The conversion is right there in the code, so nobody reading it later has to wonder whether a rounding happened behind their back.

```
fun main(stdio: Stdio)
    let count = 3
    let total = 10.0
    let average = total / to_float(count)
    stdio.println("average = ${average}")
```

```
$ capa --run values.capa
average = 3.3333333333333335
```

Writing Numbers Clearly

Long numbers are hard to read, so Capa lets you place underscores between digits as visual separators; they have no effect on the value. You can also write integers in other bases when it helps: hexadecimal with `0x`, octal with `0o`, and binary with `0b`.

```
let population = 1_000_000   // same as 1000000
```

```
let red   = 0xff            // hexadecimal 255
let perms = 0o755           // octal 493
let mask  = 0b1010          // binary 10
```

Turning Text into Numbers

Numbers that arrive as text, for example something a user typed, are still text until you convert them. The functions `parse_int` and `parse_float` do that. Because the text might not be a valid number, they hand back a result you have to deal with; the gentlest way for now is `unwrap_or`, which supplies a fallback value when the text cannot be parsed. We will study this pattern properly in Chapter 8.

```
fun main(stdio: Stdio)
    let typed = "42"
    let n = parse_int(typed).unwrap_or(0)
    stdio.println("${n + 1}")   // 43
```

Strings

A **string** is a piece of text, written between double quotes. Strings are how programs hold names, messages, file contents, and anything else made of characters.

```
let greeting = "Hello, world!"
let empty = ""
```

Putting Values Inside Strings: Interpolation

You have already been using the single most convenient string feature in Capa: **interpolation**. Inside a string, `${...}` is replaced by the value of whatever expression you put between the braces, converted to text automatically.

```
fun main(stdio: Stdio)
    let name = "Capa"
    let n = 42
    stdio.println("Hello, ${name}! Twice n is ${n * 2}.")
```

```
$ capa --run values.capa
Hello, Capa! Twice n is 84.
```

The expression inside the braces can be anything that produces a value, including arithmetic like `${n * 2}` or a method call. If you ever need the literal characters `${` in your output, double the dollar sign: `"$${"` prints `${`.

Joining Strings

The `+` operator, which adds numbers, also **joins** (concatenates) two strings into one. As with numbers, Capa never mixes types: `+` works on two strings or two numbers, but joining a string and a number directly is an error. Convert, or better, use interpolation.

```
let first = "Nelson"
let last = "Duarte"
let full = first + " " + last   // "Nelson Duarte"
```

Escape Sequences and Raw Strings

Some characters cannot be typed directly inside a string. For those, Capa uses **escape sequences**: a backslash followed by a letter. The two you will use most are `\n` for a new line and `\t` for a tab. (Capa strings are single-line; `\n` is how you produce a line break inside one.)

```
fun main(stdio: Stdio)
    stdio.println("Line one\nLine two")
    stdio.println("Name:\tCapa")
```

```
$ capa --run values.capa
```

```
Line one
Line two
Name:   Capa
```

Sometimes you want the backslashes to stay literal, as in a Windows path or a regular expression. Prefix the string with `r` to make it **raw**, which switches off both escapes and interpolation:

```
let path = r"C:\Users\Capa\file.txt"   // backslashes kept as-is
```

What Strings Can Do: Methods

A string carries a set of built-in operations called **methods**. You call a method by writing a dot and the method name after the string, like `name.to_upper()`. A method always produces a new value; because strings are immutable, none of these change the original, they hand you a fresh string back. Here are the ones you will use constantly:

```
fun main(stdio: Stdio)
    let s = "  Capa Language  "
    stdio.println("${s.trim()}")             // "Capa Language"
    stdio.println("${s.trim().to_upper()}") // "CAPA LANGUAGE"
    stdio.println("${s.trim().length()}")   // 13
    stdio.println("${s.contains("Capa")}")  // true
```

Notice `s.trim().to_upper()`: methods can be **chained**, each one acting on the result of the one before. Other handy string methods include `to_lower()`, `starts_with(...)`, `ends_with(...)`, `replace(old, new)`, and `split(sep)`, which breaks a string into a list of pieces. The full list lives in the standard-library reference; you do not need to memorise it, only to know that such methods exist and where to look.

Bools, Chars, and Unit

The three remaining primitives are quickly told. A **Bool** is a truth value, either `true` or `false`. Bools are the answer to yes/no questions and the engine of decision-making; we will lean on them heavily in Chapter 5 when we reach `if`.

```
let is_ready = true
let is_empty = "".is_empty()   // true
```

A **Char** is exactly one character, written in single quotes: `'A'`, `'7'`, `'\n'`. Mind the quotes: `'A'` is a `Char`, while `"A"` is a one-character `String`. They are different types. You will mostly work with `String`; `Char` appears when you examine text one character at a time.

Finally, **Unit**, written `()`, is the "nothing" value. It is what a function returns when it does its work for the side effect alone and has no meaningful result, like `stdio.println`, which prints and returns `()`. You will rarely write `()` yourself, but it helps to know the empty value has a name.

Comments

A **comment** is text in your source file that Capa ignores; it exists only to explain the code to human readers. Anything after `//` on a line is a comment.

```
// Convert the user's input before using it.
let n = parse_int(typed).unwrap_or(0)   // fall back to 0 on bad
input
```

Capa also has **documentation comments**, written with `///`, that attach to the declaration just below them and can be turned into HTML documentation by the compiler. We will use those later. For now, write ordinary `//` comments to explain *why* a piece of code does what it does, not to restate *what* it plainly says.

Inferred or Explicit Types?

By default Capa infers the type of a binding from its value, so `let n = 42` gives you an `Int` without you saying so. You may also write the type explicitly, after a colon, when it adds clarity or when the value alone is not specific enough:

```
let n: Int = 42
let pi: Float = 3.14
let name: String = "Capa"
```

The convention used throughout this book mirrors the language's own: **always** write explicit types on function signatures (so a function's contract is clear at a glance), and lean on inference for ordinary local `let` and `var` bindings unless an explicit type genuinely helps the reader. There are a few cases later where an annotation is required, such as an empty list whose element type cannot yet be guessed, but those are rare and the compiler will tell you when one is needed.

Reading Capa's Error Messages

You will meet error messages constantly, and in Capa they are friends, not scolding. They name the problem, point at the exact spot, and often suggest the fix. Suppose you accidentally add a number to text:

```
fun main(stdio: Stdio)
    let result = 42 + "oops"
    stdio.println("${result}")
```

```
$ capa --run values.capa
values.capa:2:18: error: '+': expected Int + Int or Float + Float
                       or String + String, got Int + String
   2 |     let result = 42 + "oops"
                         ^
```

Read it from left to right: the file and the line and column (`2:18`), then what went wrong (`+` was given an `Int` and a `String`), then a caret pointing at the offending expression. The message even lists what `+` *would* accept. From here you know your choice: convert the number to text, or the text to a number. Running `capa --check values.capa` reports the same diagnostics without executing anything, which makes it the fastest way to ask *is my code valid?*

Try It Yourself

Type each of these into a `.capa` file and run it. Where an exercise asks you to trigger an error, read the message carefully before fixing it; learning to read errors is half of learning to program.

Exercise 2-1 About You

Create three `let` bindings holding your name (a `String`), your birth year (an `Int`), and your height in metres (a `Float`). Print a single sentence that uses all three with interpolation, for example: `Ana, born in 1998, is 1.7 m tall.`

Exercise 2-2 Simple Arithmetic

Bind two integers and print their sum, difference, product, and remainder, each on its own line, each labelled (for example `sum = 8`). Then add a `Float` average of the two numbers, using `to_float` where needed.

Exercise 2-3 Mutating a Counter

Use a `var` counter starting at `0`, add `1` to it three times, and print it after each step so you see `1`, `2`, `3`. Then try reassigning it to the string `"three"`, run the program, and read the error you get.

Exercise 2-4 String Surgery

Bind the string `"  Capa Language  "` (with the extra spaces). Using method chaining, print it trimmed, in upper case, and report its length after trimming. Confirm the original binding is unchanged by printing it again at the end.

Exercise 2-5 Mixing on Purpose

Write `let bad = 5 + 2.0` and run the program. Read the error, then fix it two different ways: once by making both values `Float`, and once by using `to_float`.

Summary

You learned how Capa represents data and how to name it. Values have types; bindings give values names, with `let` for things that never change and `var` for things that do, while the type stays fixed either way. You met the six primitives, `Int`, `Float`, `Bool`, `String`, `Char`, and `Unit`, did arithmetic, and saw Capa's insistence that `Int` and `Float` never mix without an explicit `to_float` or `to_int`. You built strings with interpolation, joined them with `+`, reshaped them with methods like `trim` and `to_upper`, and learned to read the compiler's precise, helpful error messages. With values and names in hand, you are ready to organise a sequence of instructions behind a name of its own. In the next chapter we look at the first of Capa's collection types, the list.

CHAPTER 3

Lists

So far every binding has held a single value: one number, one string, one truth value. Real programs, though, deal in *many* things at once, a roster of students, the lines of a file, the scores in a game. For that you need a way to hold a whole collection of values under one name. The simplest and most common collection in Capa is the **list**, and it is the subject of this chapter.

Open a file called `lists.capa` and run each example as we go with `capa --run lists.capa`. Lists are best learned with your hands.

What Is a List?

A **list** is an ordered sequence of values. *Ordered* means the items keep the position you put them in: there is a first, a second, a third, and so on. You create a list by writing its items between square brackets, separated by commas.

```
fun main(stdio: Stdio)
    let scores = [17, 13, 9, 20]
    stdio.println("there are ${scores.length()} scores")
```

```
$ capa --run lists.capa
there are 4 scores
```

The type of `scores` is `List<Int>`, read aloud as *list of Int*. The part in angle brackets is the **element type**, the kind of value the list holds. This points at an important rule: a Capa list is **homogeneous**, meaning every element must be the same type. A `List<Int>` holds only integers; a `List<String>` holds only strings. Mixing a number and a string in one list is a type error.

```
let names = ["Ana", "Bruno", "Clara"]   // List<String>
let mixed = [1, "two", 3]               // error: elements must share
one type
```

The Empty List

A list can start empty, written `[]`, and grow later. But an empty list poses a small puzzle for the compiler: empty of *what*? With nothing inside, it cannot yet tell whether you mean a list of numbers, strings, or anything else. There are two ways to resolve this. You can annotate the type explicitly:

```
let xs: List<Int> = []
```

Or you can let the first use pin it down. The moment you add an integer, Capa infers `List<Int>` and holds you to it from then on:

```
fun main(stdio: Stdio)
    let xs = []
    xs.push(42)        // now xs is a List<Int>
    xs.push("oops")    // error: expected Int, got String
```

Reading Elements

Each element has a position called its **index**. Indexes start at **zero**, not one: the first element is at index `0`, the second at index `1`, and so on. This off-by-one feeling is universal in programming and becomes second nature with a little practice. You read an element with square brackets after the list:

```
fun main(stdio: Stdio)
    let names = ["Ana", "Bruno", "Clara"]
    stdio.println(names[0])   // Ana
    stdio.println(names[2])   // Clara
```

> **NOTE Index out of range**
>
> Plain bracket access `names[5]` on a three-element list does no bounds checking and will crash the program at run time. When you are not certain an index exists, prefer the safe method `get(i)` described next; it never crashes.

Safe Access with get, first, and last

The method `get(i)` reads the element at index `i` *safely*. Instead of risking a crash, it returns an `Option`, a value that is either `Some(x)` when the element exists or `None` when the index is out of range. We will study `Option` in full in Chapter 8; for now the easiest way to use one is `unwrap_or`, which gives you the value if it is there and a fallback of your choosing if it is not.

```
fun main(stdio: Stdio)
    let names = ["Ana", "Bruno", "Clara"]
    stdio.println(names.get(1).unwrap_or("?"))   // Bruno
    stdio.println(names.get(9).unwrap_or("?"))   // ? (no element 9)
```

Two convenience methods read the ends of a list, also safely: `first()` returns the first element and `last()` the last, each as an `Option`.

```
let xs = [10, 20, 30]
stdio.println("${xs.first().unwrap_or(0)}")   // 10
stdio.println("${xs.last().unwrap_or(0)}")    // 30
```

Asking Questions About a List

Three methods answer the most common questions. `length()` tells you how many elements there are, `is_empty()` reports whether there are none, and `contains(x)` tells you whether a particular value is present.

```
fun main(stdio: Stdio)
    let xs = [10, 20, 30]
    stdio.println("${xs.length()}")        // 3
    stdio.println("${xs.is_empty()}")      // false
    stdio.println("${xs.contains(20)}")    // true
    stdio.println("${xs.contains(99)}")    // false
```

Growing a List with push

Lists are **mutable**: you can add to them after they are created. The `push(x)` method appends a value to the end. Interestingly, you can `push` onto a list even when it is bound with `let`. That is not a contradiction: the `let` fixes the *name*, the connection between `xs` and that one list, while the list itself is a container whose contents are free to change.

```
fun main(stdio: Stdio)
    let xs = [1, 2, 3]
    xs.push(4)
    xs.push(5)
    stdio.println("now ${xs.length()} elements")   // now 5 elements
```

> **NOTE Adding, but not arbitrary removing**
>
> Capa's built-in list is deliberately small: you grow it with `push` and query it with the methods above, but there is no `remove`-by-value or `sort` built in. When you need a list *without* certain elements, you build a new one by filtering, which you will see at the end of this chapter. Working with immutable transformations instead of in-place edits tends to make programs easier to reason about.

Looping Over a List

Reading elements one index at a time is tedious and error-prone. Almost always you want to visit *every* element in turn, and the tool for that is the **for**

loop. We will study loops thoroughly in Chapter 5; here is just enough to put lists to work. The form is `for name in list`, and the indented body runs once for each element, with `name` bound to that element.

```
fun main(stdio: Stdio)
    let names = ["Ana", "Bruno", "Clara"]
    for name in names
        stdio.println("Hello, ${name}!")
```

```
$ capa --run lists.capa
Hello, Ana!
Hello, Bruno!
Hello, Clara!
```

Read the loop in plain English: *for each `name` in `names`, print a greeting*. The body is indented underneath, exactly like a function body, and runs three times here, once per element, in order.

Building a List as You Go

A very common pattern combines an empty list, a loop, and `push`: start with nothing, then build up a result one element at a time. Here we take a list of numbers and build a second list holding each number doubled.

```
fun main(stdio: Stdio)
    let numbers = [1, 2, 3, 4]
    let doubled = []
    for n in numbers
        doubled.push(n * 2)
    stdio.println("first doubled value is
${doubled.first().unwrap_or(0)}")
```

```
$ capa --run lists.capa
first doubled value is 2
```

Transforming Lists Without a Loop

Building a new list from an old one, transforming every element, or keeping only some, is so common that Capa offers methods that do it directly, often more clearly than a hand-written loop. These methods take a small function as their argument, so we first need to meet the idea of an inline function.

An **inline function**, or *lambda*, is a function written right where it is needed, without a name. The syntax is `fun (param: Type) -> ReturnType => expression`. For instance `fun (x: Int) -> Int => x * 2` is a nameless function that takes an `Int` and gives back its double. Read `=>` as *produces*.

map: Transform Every Element

The `map` method builds a new list by applying a function to every element of the original. The original list is left untouched.

```
fun main(stdio: Stdio)
    let numbers = [1, 2, 3, 4]
    let squares = numbers.map(fun (x: Int) -> Int => x * x)
    for s in squares
        stdio.println("${s}")
```

```
$ capa --run lists.capa
1
4
9
16
```

filter: Keep Only What Matches

The `filter` method builds a new list containing only the elements for which a test function returns `true`. The test takes an element and returns a `Bool`. Here we keep just the even numbers; `x % 2 == 0` is true exactly when `x` divides evenly by two.

```
fun main(stdio: Stdio)
    let numbers = [1, 2, 3, 4, 5, 6]
    let evens = numbers.filter(fun (x: Int) -> Bool => x % 2 == 0)
    stdio.println("kept ${evens.length()} of ${numbers.length()}")
```

```
$ capa --run lists.capa
kept 3 of 6
```

fold: Reduce to a Single Value

Sometimes you want to collapse a whole list into one value, a sum, a maximum, a joined string. That is the job of `fold`. It takes a starting value and a function of two arguments: the running result so far and the next element. Here we add up a list of numbers, starting from `0`.

```
fun main(stdio: Stdio)
    let numbers = [10, 20, 30, 40]
    let total = numbers.fold(0, fun (acc: Int, x: Int) -> Int => acc
+ x)
    stdio.println("total = ${total}")
```

```
$ capa --run lists.capa
total = 100
```

Trace it once and the mechanism is clear. `acc` (short for *accumulator*) starts at `0`. For each element it becomes `acc + x`: `0+10`, then `10+20`, then `30+30`, then `60+40`, ending at `100`. `map`, `filter`, and `fold` together cover a huge share of everyday list work, and they read top-to-bottom like a description of what you want rather than how to loop for it.

Ranges: Lists of Numbers Made Quickly

You will often want a list of consecutive integers, say `0, 1, 2, ..., 9`. Writing them out is silly, so Capa gives you **ranges**. `a..b` produces the

integers from `a` up to but **not including** `b`, while `a..=b` includes `b`. A range is just a `List<Int>`, so everything you have learned about lists applies to it, including looping and the transforming methods.

```
fun main(stdio: Stdio)
    for i in 0..5
        stdio.print("${i} ")      // 0 1 2 3 4
    stdio.println("")
    for i in 1..=5
        stdio.print("${i} ")      // 1 2 3 4 5
```

```
$ capa --run lists.capa
0 1 2 3 4
1 2 3 4 5
```

Both endpoints must be integers, and they can themselves be expressions: `(n - 1)..(n * 2)` is a perfectly good range. Because a range is a list, you can chain the same methods onto it, for example `(0..10).filter(fun (x: Int) -> Bool => x % 2 == 0)` to get the even numbers below ten.

Try It Yourself

Build each of these as a small program. Print enough output to see that it works, and reach for `get`, `first`, or `last` whenever an index might be missing.

> **Exercise 3-1 Your Week**
>
> Make a `List<String>` of the seven days of the week. Print the first day and the last day using `first()` and `last()` (with `unwrap_or`), then print how many days the list holds.

> **Exercise 3-2 Greet Each**

Make a list of three friends' names and use a `for` loop to print a personalised greeting for each one, such as `Hi, Ana, nice to see you!`.

Exercise 3-3 Grow It

Start from an empty list, `push` the numbers `1` through `5` onto it inside a `for i in 1..=5` loop, then print the list's length to confirm it is `5`.

Exercise 3-4 Squares with map

From the list `[1, 2, 3, 4, 5]`, use `map` to build a list of each number's square, then loop over the result and print each square on its own line.

Exercise 3-5 Only the Big Ones

From a list of test scores, use `filter` to keep only the scores of `10` or more, and print how many passed. Then use `fold` to compute the total of all scores.

Exercise 3-6 Count to a Hundred

Using a range, build the numbers `1` to `100`, keep only the multiples of three with `filter`, and print how many there are. (Hint: a multiple of three satisfies `x % 3 == 0`.)

Summary

You can now hold many values at once. A list is an ordered, homogeneous collection written with square brackets; its elements live at zero-based indexes you read with `xs[i]` or, more safely, with `get(i)`, `first()`, and

`last()`. You asked lists questions with `length`, `is_empty`, and `contains`, grew them with `push`, and walked them with a `for` loop. You also met three powerful transforming methods, `map`, `filter`, and `fold`, along with the inline functions they take, and saw how ranges conjure lists of consecutive integers. Lists give every element the same type and find things by position. In the next chapter we meet two more collections, the map, which finds things by a key of your choosing, and the set, which holds unique values.

CHAPTER 4

Maps and Sets

A list finds things by *position*: the third element, the last element. That is perfect when order is what matters, but often it is not. When you want to look something up by a name, a phone number by a person, a price by a product code, you want a **map**. And when all you care about is whether a value is *present*, with no duplicates and no order, you want a **set**. This chapter introduces both, plus a small but useful companion type, the tuple.

As always, type the examples into a file (`collections.capa`) and run them.

Maps: Looking Things Up by Key

A **map** stores **key-value pairs**. Each *key* is associated with one *value*, and you use the key to find the value, the way you use a word to find its definition in a dictionary. (Indeed, some languages call this type a *dictionary*.) A map has two type parameters: the type of its keys and the type of its values. `Map<String, Int>` maps strings to integers, ideal for, say, a tally of how many times each word appears.

Creating a Map

Unlike a list, a map has no special bracket literal. You create an empty one with the built-in function `new_map()`. Because there is nothing inside to infer the types from, you **must** annotate the binding with the key and value types:

```
let ages: Map<String, Int> = new_map()
```

Read `Map<String, Int>` as *a map from String to Int*. The annotation is not optional here; without it the compiler cannot know what kinds of keys and values you intend, and it will say so.

Adding and Updating Entries

The `set(key, value)` method adds a new entry, or, if the key already exists, replaces its value. There is only one method for both because, in a map, each key appears at most once.

```
fun main(stdio: Stdio)
    let ages: Map<String, Int> = new_map()
    ages.set("Ana", 30)
    ages.set("Bruno", 25)
    ages.set("Ana", 31)            // updates Ana, does not add a
second entry
    stdio.println("entries: ${ages.length()}")
```

```
$ capa --run collections.capa
entries: 2
```

Looking Up a Value

You retrieve a value with `get(key)`. Since the key might not be in the map, `get` returns an `Option`, just like a list's `get` did in the previous chapter. The gentle way to use it remains `unwrap_or`, which supplies a default when the key is absent.

```
fun main(stdio: Stdio)
    let ages: Map<String, Int> = new_map()
    ages.set("Ana", 31)
    stdio.println("Ana is ${ages.get("Ana").unwrap_or(0)}")
    stdio.println("Eve is ${ages.get("Eve").unwrap_or(0)}")   // 0,
not present
```

```
$ capa --run collections.capa
Ana is 31
```

```
Eve is 0
```

If you only need to know whether a key is present, without its value, use `contains_key(key)`, which returns a `Bool`. And `length()` and `is_empty()` work just as they do on lists.

A Real Use: Counting

Counting how often each item appears is the classic map task, and it shows the lookup-then-update rhythm you will use again and again. For each word we read its current count (defaulting to `0` the first time we see it) and store one more.

```
fun main(stdio: Stdio)
    let words = ["pear", "apple", "pear", "pear", "apple"]
    let counts: Map<String, Int> = new_map()
    for word in words
        let current = counts.get(word).unwrap_or(0)
        counts.set(word, current + 1)
    stdio.println("pears: ${counts.get("pear").unwrap_or(0)}")
```

```
$ capa --run collections.capa
pears: 3
```

A Quick Word on Tuples

Before we loop over a whole map, we need a small new idea. A **tuple** is a fixed group of values bundled together, written between parentheses. Unlike a list, a tuple has a fixed size and its parts may have *different* types. A pair of an `Int` and a `String` has type `(Int, String)`:

```
let entry = (1, "one")          // type (Int, String)
```

The most pleasant way to take a tuple apart is **destructuring**: bind several names at once, one per slot, in a single `let`.

```
fun main(stdio: Stdio)
    let entry = (1, "one")
    let (number, word) = entry
    stdio.println("${number} is spelled ${word}")
```

```
$ capa --run collections.capa
1 is spelled one
```

Tuples shine whenever a function needs to hand back two things at once, or a collection pairs values together, which is exactly what a map does.

Walking Through a Map

Three methods turn a map into lists you can loop over. `keys()` gives a list of all the keys, `values()` a list of all the values, and `pairs()` a list of `(key, value)` tuples, the most useful of the three because it hands you both at once. Combine `pairs()` with tuple destructuring in a `for` loop and a map reads beautifully:

```
fun main(stdio: Stdio)
    let prices: Map<String, Int> = new_map()
    prices.set("coffee", 2)
    prices.set("tea", 1)
    prices.set("cake", 4)
    for (item, price) in prices.pairs()
        stdio.println("${item} costs ${price}")
```

```
$ capa --run collections.capa
coffee costs 2
tea costs 1
cake costs 4
```

> **NOTE Maps have no inherent order**
>
> A map is organised for fast lookup by key, not for keeping things in a particular sequence. Do not rely on `pairs()`, `keys()`, or `values()` coming back in the order you inserted them, or in any other order. When order matters, keep a separate list of keys, or sort the pairs yourself.

Sets: Membership Without Duplicates

A **set** is a collection of **unique** values. Add the same value twice and it is still there only once. A set does not track positions or counts; its single question is *is this value in here or not?*, which it answers very quickly. Reach for a set when you want to remember which things you have already seen, or to strip duplicates out of a list.

Like a map, a set is created with a function, `new_set()`, and needs a type annotation because it starts empty:

```
let seen: Set<String> = new_set()
```

Add elements with `add(x)`; adding a value already present simply does nothing. Test membership with `contains(x)`, remove with `remove(x)` (a no-op if the value is absent), and turn the set back into a list with `to_list()` when you need to loop over it. `length()` and `is_empty()` are there too.

```
fun main(stdio: Stdio)
    let seen: Set<String> = new_set()
    seen.add("apple")
    seen.add("pear")
    seen.add("apple")              // duplicate: ignored
    stdio.println("unique items: ${seen.length()}")   // 2
    stdio.println("has pear? ${seen.contains("pear")}")  // true
```

```
$ capa --run collections.capa
```

```
unique items: 2
has pear? true
```

Removing Duplicates from a List

Putting it together: to collect the distinct values from a list, add each one to a set and read the set back out. Because the set discards repeats automatically, the result holds each value exactly once.

```
fun main(stdio: Stdio)
    let raw = ["a", "b", "a", "c", "b", "a"]
    let uniq: Set<String> = new_set()
    for item in raw
        uniq.add(item)
    for item in uniq.to_list()
        stdio.println(item)
```

```
$ capa --run collections.capa
a
b
c
```

Choosing the Right Collection

You now have three collections, and picking the right one is half of writing clear code. A quick guide:

- Use a **list** when order matters or values repeat: a sequence of events, the lines of a file, scores in the order they were earned.
- Use a **map** when you look things up by a key: a price per product, a count per word, a user record per id.
- Use a **set** when you only care about presence and want no duplicates: the tags on an article, the ids you have already processed.

These are not rigid laws, and real programs often combine them, a map whose values are lists, a list of tuples, a set built from a list. But starting from *what question will I ask of this data?* will usually point you straight at the right type.

Try It Yourself

Write each as its own small program and print enough to confirm it behaves.

Exercise 4-1 Capital Cities

Build a `Map<String, String>` from a few countries to their capitals. Look up two of them with `get(...).unwrap_or("unknown")` and print the results, including one country you did *not* add, to see the default appear.

Exercise 4-2 Inventory

Make a `Map<String, Int>` of products to quantities. Print every product and its quantity with a `for (name, qty) in stock.pairs()` loop, then print the total number of distinct products with `length()`.

Exercise 4-3 Word Count

Given a list of words (with repeats), build a `Map<String, Int>` of how many times each word appears, using the lookup-then-update pattern. Print each word and its count.

Exercise 4-4 Unique Visitors

Given a list of visitor names with duplicates, add them all to a `Set<String>` and print how many *distinct* visitors there were, then list them.

Exercise 4-5 Seen Before?

Loop over a list of numbers and, using a `Set<Int>`, print each number the first time you see it and the word `repeat` every later time. (Hint: check `contains` before you `add`.)

Summary

You added two collections to your toolkit and a small connector between them. A map stores key-value pairs: create it with `new_map()` and a type annotation, add or update with `set`, look up with `get` (handling the missing case with `unwrap_or`), and walk it with `keys`, `values`, or, best, `pairs` together with tuple destructuring. A set stores unique values: `add`, `contains`, `remove`, and `to_list`, perfect for membership tests and removing duplicates. You met tuples, fixed groups of possibly-different types that destructure cleanly, and you saw how to choose between list, map, and set by asking what question your data must answer. With values and collections in hand, it is time to make programs that make decisions and repeat work. The next chapter is about control flow.

CHAPTER 5

Control Flow

Until now our programs have run straight through, top to bottom, doing every line exactly once. Real programs are not so obedient: they make **decisions** (do this *only if* that is true) and they **repeat** work (do this once *for each* item). These two abilities, branching and looping, are together called **control flow**, and they are what turn a list of instructions into something that can actually respond to its data. This chapter covers both.

Type the examples into `flow.capa` and run them as you go.

Asking Yes/No Questions

Every decision rests on a question that is either true or false, and in Chapter 2 you met the type that captures exactly that: `Bool`. The most common way to produce a `Bool` is to **compare** two values. Capa has six comparison operators:

- `==` equal to, and `!=` not equal to.
- `<` less than, and `>` greater than.
- `<=` less than or equal to, and `>=` greater than or equal to.

Each comparison evaluates to `true` or `false`. Note carefully the difference between `=` and `==`: a single `=` *assigns* a value to a binding, while a double `==` *asks whether two values are equal*. Mixing them up is a classic beginner slip.

```
fun main(stdio: Stdio)
    stdio.println("${5 > 3}")        // true
    stdio.println("${5 == 3}")       // false
    stdio.println("${"a" == "a"}")   // true (strings compare too)
    stdio.println("${10 != 7}")      // true
```

Combining Conditions: and, or, not

Often a decision depends on more than one thing. Capa joins conditions with the keywords `and` and `or`, and flips one with `not`. `a and b` is true only when *both* are true; `a or b` is true when *at least one* is; `not a` is true when `a` is false.

```
fun main(stdio: Stdio)
    let age = 20
    let has_ticket = true
    stdio.println("${age >= 18 and has_ticket}")   // true
    stdio.println("${age < 13 or age > 65}")       // false
    stdio.println("${not has_ticket}")             // false
```

> **NOTE Short-circuit evaluation**
>
> `and` and `or` are *short-circuiting*: Capa stops as soon as the answer is certain. In `a and b`, if `a` is already false the whole thing is false and `b` is never evaluated; in `a or b`, if `a` is true `b` is skipped. This is not just an optimisation, it lets you write a cheap test first to guard a more expensive or riskier one second.

The if Statement

An **if statement** runs a block of code only when a condition is true. Write `if`, the condition, and then the body indented underneath. The simplest form has no alternative at all:

```
fun main(stdio: Stdio)
    let score = 95
    if score >= 90
        stdio.println("Top marks!")
```

If the condition is false, the body is simply skipped and the program continues. One firm rule: in Capa the condition **must be a `Bool`**. Unlike

some languages, you cannot use a number or a string as a stand-in for true or false; you write the comparison out, such as `count != 0`, which keeps the meaning explicit.

if-else: Choosing Between Two Paths

Add an `else` block to handle the case where the condition is false. Exactly one of the two blocks runs, never both.

```
fun main(stdio: Stdio)
    let score = 58
    if score >= 60
        stdio.println("You passed.")
    else
        stdio.println("Not quite, try again.")
```

```
$ capa --run flow.capa
Not quite, try again.
```

if-elif-else: More Than Two Choices

When there are several possibilities, chain them with `elif` (short for *else if*). Capa checks each condition in order, runs the body of the **first** one that is true, and skips the rest. A final `else` catches everything that matched none of the conditions.

```
fun main(stdio: Stdio)
    let score = 82
    if score >= 90
        stdio.println("Grade: A")
    elif score >= 80
        stdio.println("Grade: B")
    elif score >= 70
        stdio.println("Grade: C")
    else
        stdio.println("Grade: F")
```

```
$ capa --run flow.capa
Grade: B
```

Because only the first matching branch runs, order matters. Here `82` satisfies `score >= 70` as well, but it never gets that far: `score >= 80` matched first. The `else` is optional, leave it out when there is nothing to do for the leftover cases, and you can have as many `elif` branches as you need.

if as an Expression

Sometimes all you want is to *choose a value* based on a condition. You can do that with a full if statement and a `var`, but Capa offers something neater: `if` can be an **expression** that produces a value directly, using the `then` keyword.

```
fun main(stdio: Stdio)
    let score = 72
    let result = if score >= 60 then "pass" else "fail"
    stdio.println("Result: ${result}")
```

```
$ capa --run flow.capa
Result: pass
```

The keyword `then` is what tells Capa you mean the expression form, the one that yields a value to be bound to `result`, rather than the statement form with indented blocks. Both branches must produce values of the same type, and an `else` is required (a value must always be produced). Reach for this when you are picking between two values; reach for the statement form when each branch *does* something.

Repeating with while

A **while loop** repeats its body for as long as a condition stays true. It checks the condition, runs the body if true, then checks again, and keeps going until

the condition becomes false. It is the right tool when you do not know in advance how many times you will loop.

```
fun main(stdio: Stdio)
    var count = 1
    while count <= 5
        stdio.println("count is ${count}")
        count = count + 1
```

```
$ capa --run flow.capa
count is 1
count is 2
count is 3
count is 4
count is 5
```

Notice the three ingredients every sound `while` loop has: a `var` that is set up *before* the loop (`count = 1`), a condition that depends on it (`count <= 5`), and a step *inside* the body that moves it toward making the condition false (`count = count + 1`). Leave out that last step and the condition never changes.

> **NOTE Beware the infinite loop**
>
> If the condition can never become false, the loop runs forever and your program hangs. The usual cause is forgetting to update the variable the condition depends on. If a program seems frozen, press Ctrl-C in the terminal to stop it, then check that something inside the loop actually changes the condition.

Steering a Loop: break and continue

Two keywords give you finer control inside any loop. `break` stops the loop immediately and jumps to whatever comes after it. `continue` skips the rest of

the current pass and goes straight to the next one. Both work in `while` and `for` loops alike.

```
fun main(stdio: Stdio)
    for n in 1..=10
        if n == 7
            break               // stop entirely at 7
        if n % 2 == 1
            continue            // skip odd numbers
        stdio.println("${n}")
```

```
$ capa --run flow.capa
2
4
6
```

Walk it through: for each `n` from 1 to 10, we first stop the whole loop the moment `n` reaches `7`; before that, we skip odd numbers with `continue`; what survives both tests, the even numbers below seven, gets printed. Used sparingly, `break` and `continue` express *stop now* and *skip this one* far more directly than piling on extra conditions.

Looping with for

You met the **for loop** while working with lists; it is the everyday way to do something once for each item in a collection. It works over any list, including the ranges from Chapter 3.

```
fun main(stdio: Stdio)
    let names = ["Ana", "Bruno", "Clara"]
    for name in names
        stdio.println("Hi, ${name}")
    for i in 1..=3
        stdio.println("line ${i}")
```

Use a `for` loop when you have a collection to walk or a known count to repeat; use a `while` loop when you are waiting for some condition to change

and cannot say in advance how many turns it will take. Most counting loops are clearest as a `for` over a range.

Putting Conditions Inside Loops

The real power comes from combining the two ideas: a loop that makes a decision on each pass. Here we walk a list of scores, tallying how many passed and printing a note for the truly excellent ones.

```
fun main(stdio: Stdio)
    let scores = [55, 91, 73, 48, 88]
    var passes = 0
    for s in scores
        if s >= 60
            passes = passes + 1
        if s >= 90
            stdio.println("${s} is outstanding")
    stdio.println("${passes} of ${scores.length()} passed")
```

```
$ capa --run flow.capa
91 is outstanding
3 of 5 passed
```

Writing Readable Conditions

Control flow is where code most easily turns into a tangle, so a few habits pay off. Prefer the positive, plainly-named condition: `if order.is_empty()` reads better than a double negative. Keep nesting shallow, deeply indented `if` inside `if` inside `if` is hard to follow, and can often be flattened with an `elif` chain or an early `continue`. And when a condition gets long, give it a name with a `let` first; `let is_adult = age >= 18` makes the `if` that follows tell a story.

Try It Yourself

Build each program, run it, and check the output matches what you expect before moving on.

Exercise 5-1 Number Sign

Bind an integer and use an `if-elif-else` chain to print whether it is `positive`, `negative`, or `zero`. Test it with a value of each kind.

Exercise 5-2 Grades

Write a program that turns a numeric `score` into a letter grade with an `if-elif-else` chain (A for 90+, B for 80+, C for 70+, D for 60+, otherwise F). Then rewrite the pass/fail decision alone as a single `if` *expression* using `then`/`else`.

Exercise 5-3 Admission

Given an `age` and a `Bool has_ticket`, print `Welcome` only if the person is at least 18 **and** holds a ticket, and an explanatory message otherwise. Use `and` in the condition.

Exercise 5-4 Countdown

Use a `while` loop and a `var` to count down from 5 to 1, printing each number, then print `Lift off!` after the loop ends.

Exercise 5-5 First Multiple

Loop over the numbers 1 to 100 and print the first one that is divisible by both 3 and 7, then `break`. (Hint: `n % 3 == 0 and n % 7 == 0`.)

Exercise 5-6 Skip the Negatives

Given a list of integers that includes some negative values, loop over it, use `continue` to skip the negatives, and print the sum of only the non-negative numbers.

Summary

Your programs can now decide and repeat. You build conditions from the comparison operators and combine them with `and`, `or`, and `not`, which short-circuit. You branch with `if`, `elif`, and `else`, remembering that only the first matching branch runs and that a condition must be a `Bool`, and you choose a value inline with the `if ... then ... else` expression. You repeat an unknown number of times with `while`, guarding against infinite loops, and a known number of times with `for` over a list or range, steering either kind with `break` and `continue`. Combine a loop with a condition and you can already express a great deal of useful logic. So far, though, all that logic has lived inside `main`. In the next chapter we learn to package it into functions of our own.

CHAPTER 6

Functions

Every program you have written lives inside one function called `main`. As programs grow, cramming everything into `main` becomes unwieldy: the same few lines get copied around, and the big picture drowns in detail. A **function** is the cure. It is a named, reusable block of instructions that does one job. You write it once and call it by name wherever you need it, which keeps your code short, clear, and easy to change. In this chapter you learn to write your own.

Use a file called `functions.capa` and run the examples as we build them up.

Defining and Calling a Function

You have been reading function definitions since Chapter 1; now you will write them. A function definition has five parts: the keyword `fun`, a name, a list of **parameters** in parentheses, an optional **return type** after `->`, and the **body** indented underneath.

```
fun double(n: Int) -> Int
    return n * 2
```

Read it as: *`double` is a function that takes one `Int`, called `n`, and gives back an `Int`*. The `return` keyword hands a value back to whoever called the function. To **call** (run) a function, write its name followed by the values you are passing, in parentheses:

```
fun main(stdio: Stdio)
    let result = double(7)
    stdio.println("${result}")    // 14
```

When `double(7)` runs, the value `7` is handed to the parameter `n`, the body computes `7 * 2`, and `return` sends `14` back, which we bind to `result`. The function is defined once and can be called as many times as you like, with different values each time.

Parameters and Arguments

Two words that are easy to confuse, and worth pinning down. A **parameter** is the name in the definition (`n` above): a placeholder for a value the function will receive. An **argument** is the actual value you pass in when you call it (`7` above). The parameter is the empty seat; the argument is the passenger who sits in it.

> **NOTE Every parameter needs a type**
>
> In Capa, **every parameter must declare its type**, with no exceptions. The compiler never guesses from how you call the function. This is deliberate: a function's signature is the single most useful summary of what it does, so Capa insists the signature be complete on its own. Likewise, when a function returns a value, its **return type is required** after `->`.

Functions That Return a Value, and Functions That Don't

`double` produces a value, so it declares `-> Int`. Many functions, though, exist purely to *do* something, print a report, save a file, and have no meaningful result to hand back. Such a function simply omits the return type. Behind the scenes it returns `Unit`, the "nothing" value from Chapter 2; you could write `-> ()` explicitly, but it is conventional to leave it off.

```
fun announce(stdio: Stdio, message: String)
    stdio.println("*** ${message} ***")
```

```
fun main(stdio: Stdio)
    announce(stdio, "Welcome to Capa")
```

```
$ capa --run functions.capa
*** Welcome to Capa ***
```

That first parameter, `stdio: Stdio`, deserves a closer look, and it is the subject of the next section.

Passing the Screen to a Function

Here is a Capa idea that surprises newcomers and is worth meeting now, even though we will not explore it fully until Chapter 9. Printing to the screen requires the `Stdio` capability. Only `main` is handed one by the runtime. So if a helper function like `announce` wants to print, **it must be given `stdio` as a parameter**, and the caller must pass it along. A function that never receives `stdio` simply cannot print, no matter what its body tries.

This is the heart of the whole language showing through: the authority to touch the outside world travels openly, as an argument, from one function to the next. For now, just follow the pattern, if a function needs to print, give it a `stdio: Stdio` parameter and pass `stdio` when you call it. You will understand exactly why in Chapter 9.

Multiple Parameters

A function can take any number of parameters, separated by commas. Their order at the call site matches their order in the definition: the first argument fills the first parameter, and so on. These are called **positional arguments**.

```
fun rectangle_area(width: Int, height: Int) -> Int
    return width * height

fun main(stdio: Stdio)
    stdio.println("${rectangle_area(4, 3)}")   // 12
```

Position is everything here: `rectangle_area(4, 3)` means width `4`, height `3`. Get the order wrong and the program may still run but compute the wrong thing, which brings us to a feature that makes calls clearer.

Naming Your Arguments

When a function takes several parameters, a bare call like `make_box(4, 3, 2, true)` is hard to read, what does `true` mean? Capa lets you pass arguments **by name**, labelling each one at the call site:

```
fun greet(name: String, age: Int) -> String
    return "${name} is ${age}."

fun main(stdio: Stdio)
    let s = greet(name: "Ana", age: 30)
    stdio.println(s)
```

```
$ capa --run functions.capa
Ana is 30.
```

Named arguments make a call self-documenting and free you from remembering the exact order. Two limits to keep in mind: you cannot place a positional argument *after* a named one, and named arguments work only on functions you define, not on the built-in methods of types like `String` or `Map` (their parameter names are not tracked by the type system), which always take positional arguments.

No Defaults, No Overloading

Two rules round out Capa's view of functions, and both trade a little convenience for a lot of clarity. First, **there are no default argument values**: every parameter must be supplied at every call. Second, **there is no overloading**: a name refers to exactly one function. You cannot have two functions both called `double`; if you need a variant, give it its own name, like

`double_float`. The reasoning is the same in both cases, what a call does should be obvious from reading it, not dependent on hidden defaults or on which overload the compiler happened to pick.

Returning More Than One Value

A function returns a single value, but with the tuples from Chapter 4 that "single value" can bundle several together. A function that needs to report, say, both the smallest and largest of a list returns a tuple, and the caller destructures it.

```
fun min_max(numbers: List<Int>) -> (Int, Int)
    var lo = numbers.first().unwrap_or(0)
    var hi = numbers.first().unwrap_or(0)
    for n in numbers
        if n < lo
            lo = n
        if n > hi
            hi = n
    return (lo, hi)

fun main(stdio: Stdio)
    let (low, high) = min_max([7, 2, 9, 4])
    stdio.println("low ${low}, high ${high}")
```

```
$ capa --run functions.capa
low 2, high 9
```

Functions That Take Functions

Because a function is a value, you can pass one *to* another function. The type of a function is written `Fun(ParamTypes) -> ReturnType`. A function that applies another function twice looks like this:

```
fun apply_twice(f: Fun(Int) -> Int, x: Int) -> Int
    return f(f(x))
```

```
fun inc(n: Int) -> Int
    return n + 1

fun main(stdio: Stdio)
    stdio.println("${apply_twice(inc, 10)}")   // 12
```

We passed `inc` by name. You can also write the function right at the call site as a **lambda**, the nameless inline functions you first met with `map` and `filter` in Chapter 3. A lambda uses the same `fun` keyword with a `=>` before its body:

```
fun main(stdio: Stdio)
    let add = fun (a: Int, b: Int) -> Int => a + b
    stdio.println("${add(3, 4)}")   // 7
```

> **NOTE Lambdas and capabilities**
>
> A lambda cannot reach out and grab a capability such as `stdio` from the surrounding code; capabilities must always arrive as explicit parameters. This is part of the discipline that stops authority from being smuggled around invisibly. It will make complete sense once we reach Chapter 9; for now, just know a lambda works with the values you pass it, not with the outside world.

Two Common Errors

Two mistakes are so common when writing functions that Capa has clear, dedicated messages for them. The first is **forgetting to return** on some path. If a function promises a return type but a path through its body can reach the end without returning, Capa refuses it:

```
fun double(n: Int) -> Int
    let r = n * 2          // computed, but never returned
```

```
$ capa --run functions.capa
functions.capa:1:5: error: function 'double': declared return type
'Int', but the
                    body can fall through without returning a value
```

The fix is to add `return r`. The second mistake is **returning the wrong type**. The value you return must match the declared return type exactly, since Capa never converts implicitly:

```
fun double(n: Int) -> Int
    return "twice that"   // error: expected Int, got String
```

Both errors are caught at compile time, before the program runs, by `capa --check` or `capa --run`. Read them as helpful reminders of the contract your signature made.

Writing Good Functions

A few habits make functions a pleasure to use. Name a function for the action it performs, usually a verb: `calculate_total`, `format_name`, `is_valid`. Keep each function focused on **one** job; if you find yourself describing it with the word "and", it may want splitting in two. Keep them short enough to read at a glance. And document anything non-obvious with a **doc comment**, written with `///` directly above the function, which Capa can later collect into HTML documentation.

```
/// Returns the larger of two integers.
fun max_of(a: Int, b: Int) -> Int
    if a >= b
        return a
    return b
```

Try It Yourself

Write each function, then call it from `main` and print the result to confirm it works.

Exercise 6-1 Greeter

Write a function `greeting(name: String) -> String` that returns a friendly sentence using the name, then call it from `main` and print the result for two different names.

Exercise 6-2 Celsius to Fahrenheit

Write `to_fahrenheit(celsius: Float) -> Float` that returns `celsius * 9.0 / 5.0 + 32.0`. Print the result for a few temperatures. (Remember: all the numbers must be `Float`.)

Exercise 6-3 Named Call

Write `describe(name: String, age: Int, city: String) -> String` and call it once using only **named** arguments. Then try putting a positional argument after a named one and read the error Capa gives you.

Exercise 6-4 Sum and Count

Write `sum_and_count(numbers: List<Int>) -> (Int, Int)` that returns both the total and how many numbers there were. Destructure the result in `main` and print the average (convert with `to_float`).

Exercise 6-5 Printing Helper

Write `print_banner(stdio: Stdio, text: String)` (returning nothing) that prints the text surrounded by a line of dashes above and below. Call it

from `main`, passing `stdio` through. Notice that without the `stdio` parameter it could not print at all.

Exercise 6-6 Apply a Function

Write `apply_to_all(numbers: List<Int>, f: Fun(Int) -> Int) -> List<Int>` that returns a new list with `f` applied to each element (use the list's `map`). Call it once with a named function and once with a lambda.

Summary

Functions let you name a job once and reuse it everywhere, keeping programs short and readable. You learned the shape of a definition, `fun`, name, typed parameters, an optional return type, and a body, and the difference between a parameter and the argument that fills it. You wrote functions that return values and functions that just act and return `Unit`, called them with positional and named arguments, and returned several values at once with a tuple. You met Capa's firm stances, every parameter typed, no defaults, no overloading, and got your first real taste of the capability idea: to print, a helper must be handed `stdio`. Finally you passed functions to functions and read the two errors Capa raises about returns. Next we move from the language's built-in types to types you design yourself: structs and sum types.

CHAPTER 7

Structs and Sum Types

The built-in types, numbers, strings, lists, maps, describe data in general. But the things your programs are really about, a user, a payment, a shape on screen, a move in a game, are specific, and they deserve types of their own. Capa gives you two ways to build them. A **struct** bundles several named pieces that are all present together (a point *has* an x *and* a y). A **sum type** describes a value that is exactly one of several alternatives (a traffic light is red *or* amber *or* green). Master these two and you can model almost any idea precisely.

Type the examples into `types.capa` and run them as we go.

Structs: Grouping Related Data

A **struct** is a record made of named **fields**. You define one with the `type` keyword, a name, and the fields in braces, each annotated with its type.

```
type Point {
    x: Int,
    y: Int
}
```

To build a value of this type, an **instance**, you write the type name followed by a value for each field, much like a JSON object. You then read a field with a dot.

```
fun main(stdio: Stdio)
    let p = Point { x: 3, y: 4 }
    stdio.println("(${p.x}, ${p.y})")
```

```
$ capa --run types.capa
(3, 4)
```

The order in which you list the fields at construction does not matter, but **every field must be provided**, there are no default values. If you frequently need a particular starting value, the idiom is to write a small **constructor function** that fills the fields for you:

```
fun origin() -> Point
    return Point { x: 0, y: 0 }
```

Structs Are Nominal

Two structs with the very same fields are still **different types** if they have different names. A `Point` is not interchangeable with some `Pixel` that also has `x: Int, y: Int`. This is called **nominal typing**, and it is on purpose: the name carries meaning, and Capa uses it to catch the mistake of passing a screen pixel where a map coordinate was expected, even when their shapes coincide.

Taking a Struct Apart

Just as you destructured a tuple in Chapter 4, you can destructure a struct, pulling its fields into local names in a single `let`:

```
fun main(stdio: Stdio)
    let p = Point { x: 3, y: 4 }
    let Point { x, y } = p
    stdio.println("sum of coordinates: ${x + y}")
```

Giving a Struct Methods

Functions that naturally belong to a type can be attached to it as **methods** inside an `impl` block. Within the block, the special first parameter `self` refers

to the instance the method is called on. You then call the method with a dot, just like a built-in one.

```
type Rect {
    width: Int,
    height: Int
}

impl Rect
    fun area(self) -> Int
        return self.width * self.height

    fun is_square(self) -> Bool
        return self.width == self.height

fun main(stdio: Stdio)
    let r = Rect { width: 4, height: 3 }
    stdio.println("area ${r.area()}")          // area 12
    stdio.println("square? ${r.is_square()}")  // square? false
```

Methods keep the operations on a type close to the type itself, which makes code easier to find and to read. (An `impl` block like this one defines a type's own *inherent* methods. A second kind, `impl SomeTrait for Rect`, makes a type satisfy a shared interface; traits wait until Chapter 11.)

Sum Types: One of Several Shapes

A struct says *and*: a `Rect` has a width **and** a height. A **sum type** says *or*: a value is one **of** a fixed set of named **variants**. Each variant may optionally carry a payload. Here is the classic example, a shape that is a circle, a rectangle, or a square:

```
type Shape =
    Circle(Float)
    Rectangle((Float, Float))
    Square(Float)
```

`Circle` carries one `Float` (its radius); `Square` one `Float` (its side); `Rectangle` carries two numbers, and a variant that needs more than one value carries them as a **tuple**, which is why you see the double parentheses: the outer pair is the variant, the inner pair is the tuple. You build a value by naming the variant and supplying its payload.

```
let c = Circle(5.0)
let r = Rectangle((3.0, 4.0))
let s = Square(2.0)
```

Variants Without a Payload

A variant need not carry anything. A sum of payloadless variants is exactly what other languages call an *enum*, a small fixed set of named options:

```
type Color =
    Red
    Green
    Blue

fun main(_stdio: Stdio)
    let favourite = Red
```

> **NOTE An unused capability**
>
> Notice `_stdio` with a leading underscore. A capability parameter must be used, or Capa warns that it is idle. When a function genuinely does not need it, prefixing the name with `_` tells the compiler the omission is intentional and silences the warning.

Pattern Matching with match

How do you act on a value that might be any of several variants? With **match**, the tool that inspects a sum value, figures out which variant it is, and

pulls out its payload, all at once. Each **arm** names a variant, binds its payload to fresh local names, and gives the code to run for that case.

```
fun area(shape: Shape) -> Float
    match shape
        Circle(r) ->
            return 3.14159 * r * r
        Rectangle((w, h)) ->
            return w * h
        Square(side) ->
            return side * side
```

In the `Circle` arm, `r` is bound to the radius; in `Rectangle`, the tuple pattern `(w, h)` splits the payload into two `Float` values; in `Square`, `side` is the side length. Inside each arm those are ordinary local values you can compute with.

match as an Expression

Often each arm simply produces a value, and you want the whole `match` to *be* that value. Like `if`, `match` can be an expression: drop the `return`s and let each arm yield its result, then bind or use the whole thing.

```
fun area(shape: Shape) -> Float
    return match shape
        Circle(r) -> 3.14159 * r * r
        Rectangle((w, h)) -> w * h
        Square(side) -> side * side
```

All arms must produce the same type (`Float` here). This expression form is usually the clearer of the two when every branch just computes a value.

The Exhaustiveness Guarantee

Here is one of Capa's most valuable safety nets. A `match` on a sum type must handle **every** variant. Leave one out and the program will not compile:

```
fun area(shape: Shape) -> Float
```

```
    match shape
        Circle(r) -> return 3.14159 * r * r
        Rectangle((w, h)) -> return w * h
    // forgot Square
```

```
$ capa --run types.capa
types.capa:2:9: error: non-exhaustive match: missing variant 'Square'
   2 |     match shape
             ^
```

This sounds like a small convenience, but it is a profound one. The day you add a fourth shape, say `Triangle`, **every** `match` on `Shape` that did not already have a catch-all instantly becomes a compile error, listing exactly the spots you must update. The class of bug where you add a case and silently forget to handle it somewhere simply cannot happen. When you genuinely want to handle "everything else" in one arm, use the wildcard `_`:

```
fun describe(color: Color) -> String
    return match color
        Red -> "stop"
        _ -> "go or caution"
```

Guards and Multiple Patterns

Two refinements make `match` even more expressive. A **guard** adds an `if` condition to an arm, so it matches only when the condition also holds. And an **or-pattern**, written with `|`, lets one arm cover several alternatives at once.

```
fun sign(n: Int) -> String
    return match n
        x if x > 0 -> "positive"
        x if x < 0 -> "negative"
        _ -> "zero"
```

Modelling a Small Domain

Structs and sums shine brightest together. Suppose you are recording what happens in an application. Each event is one of a few kinds, some carrying data; a sum type captures that exactly, and a single `match` turns any event into a readable line.

```
type Event =
    Login(String)
    Logout(String)
    Failure((Int, String))

fun describe(e: Event) -> String
    return match e
        Login(user) -> "${user} logged in"
        Logout(user) -> "${user} logged out"
        Failure((code, msg)) -> "error ${code}: ${msg}"

fun main(stdio: Stdio)
    let events = [Login("ana"), Failure((404, "not found")),
Logout("ana")]
    for e in events
        stdio.println(describe(e))
```

```
$ capa --run types.capa
ana logged in
error 404: not found
ana logged out
```

Notice how a `List<Event>` can hold all three kinds at once, because they share the one type `Event`, and how `describe` is guaranteed by the compiler to handle every kind. That combination, model the alternatives as a sum, then `match` on it, is one of the most powerful habits in the language.

A Word on Generic Types

Like functions, your types can take type parameters, so a single definition works for many element types. You have already been using two such generic sum types: `Option<T>` and `Result<T, E>` are nothing more than built-in sum

types parameterised by their payload. That is the subject of the very next chapter, where the reason a function returns a sum instead of a plain value gets a name: the operation might fail.

Two Common Errors

Beyond the non-exhaustive `match` above, the other frequent slip is **leaving out a field** when constructing a struct. Because there are no defaults, Capa requires every field and names the one you forgot:

```
fun main(stdio: Stdio)
    let p = Point { x: 3 }     // missing y
    stdio.println("${p.x}")
```

```
$ capa --run types.capa
types.capa:2:13: error: struct literal 'Point': missing field 'y'
   2 |     let p = Point { x: 3 }
                   ^
```

Try It Yourself

Define each type, then write a `main` that builds a few values and prints them.

> **Exercise 7-1 A Book**
>
> Define a struct `Book` with fields `title: String`, `author: String`, and `year: Int`. Build two books, and print each as a sentence using field access. Then destructure one book into local names and print those.

> **Exercise 7-2 Circle Methods**
>
> Define a struct `Circle` with a `radius: Float`. Give it an `impl` block with a method `area(self) -> Float` (use `3.14159 * self.radius *`

`self.radius`) and a method `diameter(self) -> Float`. Build a circle and print both.

Exercise 7-3 Constructor

Add a constructor function `unit_circle() -> Circle` that returns a circle of radius `1.0`, and use it in `main`. (Recall there are no default field values, so constructor functions are how you make ready-made values.)

Exercise 7-4 Traffic Light

Define a payloadless sum type `Light` with variants `Red`, `Amber`, and `Green`. Write `action(light: Light) -> String` using `match` that returns `stop`, `wait`, or `go`. Then delete the `Green` arm and read the exhaustiveness error.

Exercise 7-5 Shapes

Using the `Shape` sum from this chapter, write `perimeter(shape: Shape) -> Float` with a `match` (circle: `2.0 * 3.14159 * r`; rectangle: `2.0 * (w + h)`; square: `4.0 * side`). Build one of each and print their perimeters.

Exercise 7-6 Events

Extend the `Event` type with a new variant `Signup(String)`. Notice that, until you add a `Signup` arm, every `match` on `Event` fails to compile, exactly the safety net described in this chapter. Then handle the new case.

Summary

You can now design your own types. A struct groups several named fields that are all present at once; you construct it by naming every field (there are no defaults), read fields with a dot, destructure it like a tuple, and attach methods in an `impl` block where `self` is the instance. A sum type describes a value that is exactly one of a fixed set of variants, each optionally carrying a payload (a tuple when it needs several values). You operate on sums with `match`, which binds payloads and, crucially, is checked for exhaustiveness, so adding a variant turns every unhandled `match` into a compile error rather than a silent bug. You combined structs and sums to model a small domain, and saw that `Option` and `Result` are themselves generic sum types. Those two are how Capa handles operations that might fail, the subject of the next chapter.

CHAPTER 8

Errors as Values

Things go wrong. A user types `"banana"` where a number was expected, a file is missing, a key is not in the map. A program that ignores these possibilities is a program waiting to crash. Many languages handle failure with **exceptions**, invisible jumps that can erupt from almost any line. Capa takes a different, quieter path: it has **no exceptions at all**. Instead, failure is just an ordinary **value** of a particular type, and the compiler refuses to let you forget about it. You have already been brushing against the two types that make this work, `Option` and `Result`; this chapter makes them explicit.

Type the examples into `errors.capa` and run them as we go.

Option: a Value That Might Not Be There

Some operations have no sensible answer to give. What is the first element of an empty list? What integer is `"hello"`? Capa's reply is `Option<T>`, a sum type (from Chapter 7) with exactly two variants: `Some(value)` when there is a value, and `None` when there is not.

```
// built-in; the names Some and None are reserved
type Option<T> =
    Some(T)
    None
```

You have met `Option` already, perhaps without noticing: `xs.get(i)`, `xs.first()`, `map.get(key)`, and `parse_int(text)` all hand one back, precisely because each might come up empty. The crucial point is that Capa **will not let you use the inner value without first dealing with the `None` case**. The honest, explicit way is a `match`:

```
fun main(stdio: Stdio)
    match parse_int("42")
        Some(n) ->
            stdio.println("parsed: ${n}")
        None ->
            stdio.println("not a number")
```

```
$ capa --run errors.capa
parsed: 42
```

Because `match` is exhaustive, you cannot accidentally skip the `None` branch; the compiler insists you say what happens when the value is absent. This single rule eliminates an entire family of crashes.

Convenient Option Methods

A full `match` is sometimes more ceremony than a situation deserves. `Option` carries methods for the common cases, and you have already used the first one:

- `opt.unwrap_or(default)` gives the inner value, or a fallback when it is `None`.
- `opt.is_some()` and `opt.is_none()` are simple boolean checks.
- `opt.map(f)` applies `f` to the value when present, leaving `None` untouched.
- `opt.and_then(f)` chains another `Option`-returning step (more on this with `?`).

```
fun main(stdio: Stdio)
    let maybe = parse_int("7")
    stdio.println("${maybe.unwrap_or(0)}")              // 7
    stdio.println("${parse_int("x").unwrap_or(0)}")     // 0
    let doubled = maybe.map(fun (n: Int) -> Int => n * 2)
    stdio.println("${doubled.unwrap_or(0)}")            // 14
```

Result: a Failure That Explains Itself

`Option` tells you *whether* there is a value, but not *why* there is not. When the reason matters, you reach for `Result<T, E>`: it holds either `Ok(value)` of type `T` on success, or `Err(error)` of type `E` on failure, and the error carries information about what went wrong.

```
// built-in; the names Ok and Err are reserved
type Result<T, E> =
    Ok(T)
    Err(E)
```

Anything that touches the outside world, reading a file, fetching a URL, parsing a document, returns a `Result`, usually `Result<T, IoError>` where `IoError` is Capa's built-in description of an I/O failure. You handle it with `match`, exactly as you would any sum type:

```
fun main(stdio: Stdio, fs: Fs)
    match fs.read("config.txt")
        Ok(text) ->
            stdio.println("file says: ${text}")
        Err(e) ->
            stdio.eprintln("could not read: ${e}")
```

> **NOTE Fs is a capability**
>
> That `fs: Fs` parameter is, like `stdio`, a **capability**: the authority to touch the filesystem, requested openly in the signature. A function with no `Fs` cannot read or write files at all. Capabilities are the whole point of the next chapter; here we only borrow `Fs` to have something that can fail for a real reason.

`Result` offers the same conveniences as `Option`, `is_ok()`, `is_err()`, `unwrap_or(default)`, `map(f)`, plus `map_err(g)` to transform the error and

the projections `ok()` and `err()`, which convert a `Result` into an `Option` by keeping the value or the error respectively.

The ? Operator: Propagating Failure

When a function does several fallible things in a row, writing a `match` after each one buries the logic in error handling. Capa offers a shortcut: the **`?` operator**. Append `?` to an expression that produces a `Result` or an `Option`, and it does the obvious thing, if the value is `Ok`/`Some`, it unwraps it and carries on; if it is `Err`/`None`, it stops the whole function and returns that failure immediately.

```
fun load_config(fs: Fs) -> Result<String, IoError>
    let text = fs.read("config.txt")?    // on Err, return it now
    let trimmed = text.trim()
    return Ok(trimmed)
```

Read the `?` line aloud: *read the file; if that failed, return the error from `load_config`; otherwise unwrap the text and continue*. One character replaces a five-line `match`. The same operator propagates `None` through a function that returns an `Option`:

```
fun first_two_sum(xs: List<Int>) -> Option<Int>
    let a = xs.first()?      // None if the list is empty
    let b = xs.get(1)?       // None if there is no second element
    return Some(a + b)
```

> **NOTE The ? rule**
>
> `?` only works inside a function whose return type matches what it is propagating. If the expression yields a `Result`, the enclosing function must return a `Result` (with the same error type); if it yields an `Option`, the function must return an `Option`. Otherwise the compiler rejects it, with a message saying exactly that.

Writing Your Own Failing Functions

You return these types yourself whenever an operation might not succeed. Use `Option` when the only thing to report is *absence*; here, dividing by zero has no answer, so we return `None`:

```
fun safe_divide(a: Float, b: Float) -> Option<Float>
    if b == 0.0
        return None
    return Some(a / b)

fun main(stdio: Stdio)
    stdio.println("${safe_divide(10.0, 2.0).unwrap_or(0.0)}")   // 5
    stdio.println("${safe_divide(1.0, 0.0).unwrap_or(0.0)}")    // 0
```

Use `Result` when the caller deserves to know *why* it failed. The error type can be anything; a `String` message is the simplest, though a sum type of error cases is often better in larger programs.

```
fun check_age(age: Int) -> Result<Int, String>
    if age < 0
        return Err("age cannot be negative")
    if age < 18
        return Err("must be at least 18")
    return Ok(age)

fun main(stdio: Stdio)
    match check_age(15)
        Ok(a) -> stdio.println("welcome, ${a}")
        Err(why) -> stdio.println("denied: ${why}")
```

```
$ capa --run errors.capa
denied: must be at least 18
```

Why No Exceptions?

It is worth pausing on the design choice. With exceptions, any function can fail in ways its signature never mentions, and the type system cannot enumerate those failure modes for you; the only safety net is to wrap everything in `try`, which real code rarely does consistently. Capa's bet is the opposite: make failure a value that carries a type, and have the compiler refuse to let you ignore it. The `?` operator then gives you the easy *fail-through* ergonomics of exceptions for the cases where you just want to pass a failure upward, while full `match` remains available for the cases where you want to recover. You get convenience without losing the guarantee that every failure is accounted for.

Two Common Errors

The first trips up almost everyone once: using `?` in a function whose return type does not fit. If you propagate a `Result` but declare a plain return type, Capa stops you.

```
fun read_first_line(fs: Fs) -> String
    let text = fs.read("config.txt")?   // returns Result, but fn
returns String
    return text
```

```
$ capa --run errors.capa
errors.capa:2:16: error: '?' can only propagate to a function
returning
                  Result<_, E> or Option<_>; this function returns
String
```

The fix is either to change the return type to `Result<String, IoError>`, or to stop using `?` and `match` locally instead. The second common mistake is treating an `Option` as if it were the value inside it, for example doing arithmetic on it directly:

```
let xs = [10, 20, 30]
let bad = xs.first() * 2    // error: Option<Int> is not an Int
```

The remedy is always one of the three tools from this chapter: `match` on the `Option`, call `.unwrap_or(default)`, or apply `?` when the enclosing function returns a compatible type. Once the distinction between *an `Option<Int>`* and *an `Int`* clicks, this error stops appearing.

Option or Result?

A quick rule for choosing between them. If the only thing a caller needs to know is *whether* a value exists, a lookup, a search, a parse, return an `Option`. If a failure has a *reason* the caller might log, display, or branch on, especially anything involving the outside world, return a `Result` so that reason travels with it. When in doubt, start with `Option`; you can always upgrade to `Result` when you find you need to explain the failure.

Try It Yourself

Write each function, then exercise it from `main` for both the success and the failure case.

> **Exercise 8-1 Safe Lookup**
>
> Build a `Map<String, Int>` of a few products to prices. Write a function `price_of(stock: Map<String, Int>, name: String) -> Int` that returns the price, or `0` when the product is missing, using `get(...).unwrap_or(0)`. Test it with a present and an absent product.

> **Exercise 8-2 Parse or Default**

Write `read_number(text: String) -> Int` that returns the parsed integer, or `-1` if the text is not a number. Then rewrite it to return `Option<Int>` instead and have `main` `match` on the result.

Exercise 8-3 Safe Divide Chain

Using the `safe_divide` from this chapter, write `fun divide_twice(a: Float, b: Float, c: Float) -> Option<Float>` that divides `a` by `b`, then that result by `c`, using `?` so the whole thing is `None` if either divisor is zero.

Exercise 8-4 Validated Result

Write `parse_grade(text: String) -> Result<Int, String>` that parses the text and returns `Err("not a number")` if parsing fails, `Err("out of range")` if the number is not between 0 and 100, and `Ok(n)` otherwise. Print a friendly message for each outcome with a `match`.

Exercise 8-5 Propagate It

Write `fun second_word(line: String) -> Option<String>` that splits the line on spaces and returns the second word, using `?` on `get(1)` so an empty or one-word line yields `None`.

Summary

Capa has no exceptions; failure is a value the compiler will not let you ignore. `Option<T>` models a value that might be absent, `Some(v)` or `None`, and turns up wherever a lookup or parse might come up empty. `Result<T, E>` models a failure that carries a reason, `Ok(v)` or `Err(e)`, and is what every operation touching the outside world returns. You handle both with an exhaustive

`match` or with convenience methods like `unwrap_or` and `map`, and you propagate them effortlessly with the `?` operator, whose one rule is that the enclosing function must return a matching shape. You wrote your own failing functions and learned to choose `Option` for mere absence and `Result` when the reason matters. You have now met every feature of Capa's ordinary, pleasant core. From here the language reveals what makes it unlike any other: capabilities.

CHAPTER 9

Your First Capability

Since the very first program in this book you have written `fun main(stdio: Stdio)` and used `stdio.println` to put text on the screen, taking that little `stdio` on faith. This chapter is where the faith ends and the understanding begins. `stdio` is a **capability**, and capabilities are the one idea that makes Capa unlike almost every other language. Everything in Part I so far, values, collections, control flow, functions, types, error handling, has been ordinary and familiar on purpose, so that when this idea arrives you have the room to take it seriously. It is worth it.

What stdio Really Is

When your program starts, the Capa **runtime** creates a single `Stdio` value and hands it to `main` as the `stdio` parameter. That value *is* the authority to write to standard output, the stream of text that appears in your terminal. There is no other way to reach the screen: no global print function, no hidden channel. If a piece of code holds the `Stdio` value, it can print; if it does not, it cannot. The power to print is a thing you possess, not a thing that is simply lying around.

To feel how unusual that is, compare it with the same program in Python, a language Capa otherwise resembles:

```
# Python
def main():
    print("Hello")
```

Python's `main` takes no parameters, yet it can print. The print stream is simply *there*, available to any function in any file in the whole program, with

nothing in the signature hinting at it. The same is true of opening files, reading environment variables, and making network requests. This arrangement has a name: **ambient authority**. The authority lives in the surrounding environment, not in the values a function holds.

Removing Ambient Authority

Ambient authority is convenient, and for decades it was treated as harmless. The trouble shows the moment you run code you did not write, which, through libraries and their dependencies, every real program does. A library advertised as a harmless text helper can, in Python or JavaScript, quietly read your files, inspect your environment variables, and open a network connection, because the language grants it the same ambient authority as your own code. Nothing in its public interface has to admit any of this.

Capa's answer is to abolish ambient authority. Each power becomes an ordinary **value** that must be held to be used. The screen is a `Stdio` value; the filesystem is an `Fs` value; the network is a `Net` value. The runtime hands these to `main`, and from there they travel only where you explicitly pass them. Code that was never given the `Net` value cannot reach the network, not because it promises not to, but because, in the most literal sense, it has no way to.

The Nine Built-in Capabilities

Capa builds in nine capability types, each guarding one kind of access to the outside world:

- `Stdio` - standard output and input: `println`, `print`, `eprintln`, `read_line`.
- `Fs` - the filesystem: `read`, `write`, `exists`, `list_dir`, and more.
- `Net` - the network: `get` (an HTTP request).
- `Env` - environment variables and program arguments: `get`, `args`.

- `Clock` - wall-clock time: `now_secs`, `sleep`.
- `Random` - randomness: `float_unit`, `int_range`, `with_seed`.
- `Db` - a database: exec, query.
- `Proc` - subprocesses: exec.
- `Unsafe` - the escape hatch, required to cross into Python code (Appendix C).

A function that needs one of these declares it as a parameter, with its type, exactly as you would any other argument:

```
fun read_config(fs: Fs) -> Result<String, IoError>
    return fs.read("config.txt")

fun fetch_users(net: Net) -> Result<String, IoError>
    return net.get("https://api.example.com/users")
```

And here is the heart of it: a function that takes **no** capabilities cannot do any I/O at all. Not by convention, not by good manners, but structurally, the names it would need (`fs`, `net`, `stdio`) are simply not in scope, and there is no expression anywhere in the language that conjures a capability from nothing.

How Authority Flows Through a Program

The runtime gives the full set of capabilities to `main`. From there, they move through the program the only way they can: as arguments passed from one function to the next. A helper that prints must be handed `stdio`; a helper that calls *that* helper must also have `stdio` to pass along. You saw this pattern foreshadowed in Chapter 6; now you know why it is mandatory.

```
fun greet(stdio: Stdio, name: String)
    stdio.println("Hello, ${name}")

fun greet_all(stdio: Stdio, names: List<String>)
    for name in names
        greet(stdio, name)
```

```
fun main(stdio: Stdio)
    greet_all(stdio, ["Ana", "Bruno", "Carla"])
```

Both `greet` and `greet_all` declare `Stdio`, because each ultimately calls `println`. This gives a Capa program a remarkable property: **the signatures tell you the exact authority surface**. Want to know whether some function can touch the network? Look at whether `Net` appears anywhere in the chain of signatures it calls. A function whose signature lists no capabilities is *provably pure*: it can compute, but it cannot affect or observe the outside world. When `main` needs several powers, it simply lists them all:

```
fun main(stdio: Stdio, fs: Fs)
    match fs.read("hello.txt")
        Ok(text) ->
            stdio.println(text)
        Err(e) ->
            stdio.eprintln("error: ${e}")
```

The Discipline, in Three Rules

Capabilities are values, but they are not *ordinary* values: the compiler enforces three extra rules that keep authority honest. None is arbitrary; each one shuts a door that ambient authority leaves open.

1. No Aliasing

You cannot copy a capability into a `let` binding. `let dup = stdio` is a compile error. If you could quietly alias a capability, you could stash it somewhere global and use it far from where it was granted, defeating the whole signature-as-contract idea. To give a capability to a helper, pass it directly, `helper(stdio)`, rather than binding it first.

2. No Returning, Storing, or Capturing

A built-in capability cannot be returned from a function, stored as a struct field, or captured by a lambda. Authority flows **downward**, through parameters; it is not allowed to flow back up as a return value, sideways into a data structure, or invisibly into a closure. (Chapter 11 relaxes this slightly for capabilities you define yourself, but the built-ins never bend.)

3. Must Be Used

If a function declares a capability parameter and never calls a method on it, that is a compile error too (silence it with a leading underscore, `_stdio`, when you truly mean to ignore it). A signature is a promise about what a function may do; declaring a capability you never touch is a misleading promise, so Capa forbids it.

Together these three rules guarantee something precise and powerful: the set of capabilities in a function's signature is an **upper bound** on everything that function can do. Nothing it calls, no matter how deep, can exceed the authority visible at the top.

The Payoff: a Machine-Readable Authority Map

Because all of this lives in the type system, the compiler can simply *read off* what each function is allowed to do and write it out. The command `capa --manifest` produces exactly that, a JSON description of every function's authority. For `read_config` above it would report something like:

```
{
  "name": "read_config",
  "declared_capabilities": ["Fs"],
  "provably_excluded_capabilities": [
    "Stdio", "Net", "Env", "Clock", "Random", "Unsafe"
  ]
}
```

Read the second list carefully: the compiler is not guessing or scanning for suspicious calls; it is stating, as a proven fact, that `read_config` can **never** reach the network, the environment, or the clock, because none of those values is in its signature. A reviewer, or an automated tool, can answer *which functions can touch the network?* with certainty, straight from the source. That is the practical reward for moving authority into the types.

> **NOTE Why this matters beyond your own code**
>
> The real danger in modern software is not the code you write but the dependencies you install, often hundreds, written by strangers. In Capa, a library that claims to be a simple data helper cannot secretly phone home: to use the network it would have to put `Net` in its signatures, where it would show up in code review, in dependency diffs, and in the manifest. The attack does not become impossible, but it can no longer hide. This is the problem Capa was built to address.

Two Common Errors

The first appears the moment you use a capability you forgot to declare. The name is simply not in scope:

```
fun main(stdio: Stdio)
    let contents = fs.read("hello.txt")   // fs was never declared
```

```
$ capa --run caps.capa
caps.capa:2:20: error: undefined name 'fs'
   2 |     let contents = fs.read("hello.txt")
                         ^
```

The fix is to add it to `main`: `fun main(stdio: Stdio, fs: Fs)`. The second is the aliasing rule from above, in action:

```
fun main(stdio: Stdio)
    let dup = stdio
    dup.println("hi")
```

```
$ capa --run caps.capa
caps.capa:2:15: error: 'stdio' is a capability and cannot be bound to
a 'let'
                  (capabilities flow through function parameters only)
```

Pass the capability where it needs to go instead of binding it: write `helper(stdio)`, not `let dup = stdio` followed by `helper(dup)`.

Try It Yourself

Each exercise is small, but pay attention to the signatures; they are the real lesson here.

Exercise 9-1 File Length

Write a program whose `main` takes `stdio: Stdio` and `fs: Fs`, reads `hello.txt` using `?` to propagate failure, and prints the length of its contents. (You will need `main` to return `Result<(), IoError>` so that `?` is allowed.)

Exercise 9-2 Pass It Down

Write `print_line(stdio: Stdio, text: String)` and a `print_all(stdio: Stdio, lines: List<String>)` that calls it in a loop, then call `print_all` from `main`. Notice that every function in the chain must declare `Stdio`.

Exercise 9-3 A Pure Function

Write a function `total(numbers: List<Int>) -> Int` that takes no capabilities and sums the list. Call it from `main` and print the result. Observe that `total`'s signature proves it cannot do any I/O, it is provably pure.

Exercise 9-4 Catch the Compiler

Inside a pure function, try to add a `stdio.println(...)` call without giving the function a `Stdio` parameter, and read the `undefined name 'stdio'` error. Then fix it by threading `stdio` in as a parameter and passing it from `main`.

Exercise 9-5 Read the Manifest

Run `capa --manifest yourfile.capa` on the program from 9-2 and find your functions in the output. Confirm that each one lists `Stdio` under its declared capabilities and that `total` from 9-3 declares none.

Summary

The magic word is no longer magic. `stdio` is a `Stdio` capability: a value, handed by the runtime to `main`, that carries the authority to write to the screen. Capa abolishes the ambient authority that other languages grant freely, turning each power, the screen, the filesystem, the network, a database, subprocesses, the environment, the clock, randomness, and the `Unsafe` escape hatch, into a value you must hold to use. Capabilities flow only downward, as parameters, so a function's signature is an honest upper bound on what it can do, and a function with no capabilities is provably pure. Three rules keep this airtight: no aliasing, no returning or storing or capturing, and must-be-used. The compiler then writes the whole authority map out as a manifest, turning a language feature into a security guarantee

anyone can audit. Next we sharpen the tool: a function that calls one specific URL should not hold the entire network. Attenuation is how you narrow a capability before passing it on.

CHAPTER 10

Attenuating Capabilities

The previous chapter gave you an all-or-nothing tool: a function either holds the `Net` capability and can reach the whole internet, or it does not and can reach nothing. Real programs need something in between. A function that fetches from one API should be able to reach *that* API and nothing else; a function that writes to a cache folder should be able to write *there* and nowhere else. This chapter is about **attenuation**: narrowing a capability before you pass it on, so each piece of code holds exactly the authority it needs and not a drop more.

The Principle of Least Authority

There is an old, sound idea in security called the **principle of least authority**: give every part of a system the smallest set of powers it needs to do its job. The less authority a piece of code holds, the less damage it can do if it turns out to be buggy or malicious. Capa lets you apply this principle directly. When you pass a capability down to a helper, you can hand over a *narrowed* version rather than the full one.

Narrowing with restrict_to

Each of the narrowable built-in capabilities carries a method that returns a **fresh, narrower copy** of itself. For the network that method is `restrict_to(host)`: it produces a new `Net` whose authority reaches only the host you name.

```
fun main(net: Net, stdio: Stdio)
    let api = net.restrict_to("api.example.com")
    // `api` can reach api.example.com and nothing else
```

```
    fetch_users(api, stdio)
```

The crucial subtlety is that `api` has the very same type as the `net` it came from: both are `Net`. A function receiving `api` sees a parameter of type `Net` and cannot tell it was narrowed; the narrowing lives *inside the value*, as a private allow-list, not in the type. We will see in a moment why that turns out to be a strength.

> **NOTE Binding a capability to a let, the exception**
>
> In Chapter 9 you learned that a capability cannot be bound to a `let`. Here we write exactly that: `let api = net.restrict_to(...)`. The rule and its exception fit together. What is forbidden is *aliasing* an existing capability (`let dup = stdio`), which would create a second name for the same authority. The result of `restrict_to` is not an alias, it is a **brand-new** capability the method just produced, so binding it is allowed. Only freshly created capabilities may be named with `let`.

The Seven Attenuators

Seven of the built-in capabilities can be narrowed, each with its own suitably named method:

- `net.restrict_to(host)` - reach only this exact host.
- `fs.restrict_to(prefix)` - touch only paths under this directory prefix.
- `db.restrict_to(prefix)` - reach only entries under this prefix.
- `proc.restrict_to(basename)` - spawn only this executable.
- `env.restrict_to_keys([names])` - read only these environment-variable names.
- `clock.restrict_to_after(t)` - report only times after the timestamp `t`.

- `random.with_seed(seed)` - produce a fixed, repeatable sequence from a seed.

The first six genuinely *narrow*, reducing what the capability can do. The last is slightly different: `with_seed` does not remove any power, it makes `Random` **deterministic**, so the same seed always yields the same sequence, which is invaluable for tests you want to run identically every time. The two capabilities left out, `Stdio` and `Unsafe`, have no attenuator at all: there is no narrower way to write to one output stream, and `Unsafe` is the deliberate escape hatch that does not pretend to constrain anything.

A Worked Example

Consider a function that fetches a user record from an API and caches it to a local file. Written naively, it asks for the full `Net` and the full `Fs`:

```
fun cache_user(net: Net, fs: Fs, id: String) -> Result<Unit, IoError>
    let body = net.get("https://api.example.com/users/${id}")?
    fs.write("/var/cache/users/${id}.json", body)?
    return Ok(())

fun main(net: Net, fs: Fs, stdio: Stdio)
    match cache_user(net, fs, "42")
        Ok(_) -> stdio.println("cached")
        Err(e) -> stdio.eprintln("error: ${e}")
```

This works, but `cache_user` is **over-authorised**: it could, by accident or through a malicious edit, reach any website or write to any file on the system. Narrow the two capabilities at the boundary, in `main`, before handing them over:

```
fun main(net: Net, fs: Fs, stdio: Stdio)
    let api = net.restrict_to("api.example.com")
    let cache_dir = fs.restrict_to("/var/cache/users/")
    match cache_user(api, cache_dir, "42")
        Ok(_) -> stdio.println("cached")
        Err(e) -> stdio.eprintln("error: ${e}")
```

Notice what did, and did not, change. The **signature** of `cache_user` is identical, it still takes a `Net` and an `Fs`. What changed is the **authority** of the values passed in. If someone later slips `net.get("https://attacker.com")` into `cache_user`, the narrowed `Net` refuses it at run time, before any request leaves the machine. This is the half of Capa's discipline that the type system alone cannot give you: two functions with identical signatures can hold different authority, and the difference is decided by *whoever built the value passed in*.

Monotonic by Construction

The single most important property of `restrict_to` is what it **cannot** do: once a capability has been narrowed, nothing can widen it again. Narrowing only ever shrinks authority, never grows it. Watch what happens if you try to re-point a narrowed `Net` at a different host:

```
fun main(net: Net, stdio: Stdio)
    let a = net.restrict_to("api.example.com")
    let b = a.restrict_to("other.example.com")
    // b can reach the intersection of {api...} and {other...} =
nothing
```

Each narrowed `Net` carries an internal set of allowed hosts. `restrict_to` computes the **intersection** of the current set with the new host, and an intersection can only get smaller. There is no method anywhere that enlarges the set. The unrestricted `Net` that `main` receives stands for "everything allowed"; every `restrict_to` after that only constrains it. This is what *monotonic by construction* means: any chain of restrictions, however long, always ends somewhere at least as narrow as where it began. A library handed a narrowed capability cannot wriggle back out, no matter what its code attempts.

Fail-Closed at Run Time

The compiler cannot know in advance which exact URLs or paths your program will use, those values may be computed as the program runs, so the check that a call stays inside the allowed set happens at **run time**, and it happens *before* any system call. A call outside the set does not partially execute and then get pulled back; it never reaches the network or disk at all. It simply returns an `Err` you handle like any other failure.

```
fun main(net: Net, stdio: Stdio)
    let api = net.restrict_to("api.example.com")
    match api.get("https://attacker.com/")
        Ok(body) -> stdio.println(body)
        Err(e) -> stdio.eprintln("denied: ${e}")
```

```
$ capa --run cache.capa
denied: NetRestricted: attacker.com is not in the allowed host set
```

This behaviour is called **fail-closed**: when in doubt, deny. No request was made; the forbidden host never saw a single packet. Because the denial arrives as an ordinary `Err`, your program stays in control and can log it, retry elsewhere, or carry on, exactly as it would with any other `Result`.

Making Least Authority a Habit

Attenuation costs almost nothing to apply and pays off quietly forever. A good habit is to narrow capabilities as early and as tightly as you reasonably can, usually right in `main`, or at the boundary where you call into a library, so that everything downstream runs with the least authority that still lets it work. When you call third-party code in particular, ask what it truly needs: a single host, a single directory, a handful of environment variables, and hand it only that. The narrowed value looks identical to the full one, so this discipline never complicates your function signatures; it just steadily shrinks the blast radius of anything that goes wrong.

Try It Yourself

These exercises are best run for real so you can watch a denial happen and handle it.

Exercise 10-1 One Host Only

Write `fun fetch_users(net: Net) -> Result<String, IoError>` that gets `https://api.example.com/users`. In `main`, narrow `net` to `"api.example.com"` before passing it in, and print the result or the error.

Exercise 10-2 Watch It Deny

Add a second call inside `fetch_users` to `https://evil.example.com/`. Keep the narrowing in `main` and run the program. Confirm the second call is denied at run time with a `NetRestricted` error, and that no request to the evil host is made.

Exercise 10-3 Restrict the Filesystem

Write a function that writes a short note to `/tmp/notes/today.txt`. In `main`, narrow `fs` with `restrict_to("/tmp/notes/")` before calling it. Then try to write to `/etc/passwd` from the same narrowed `Fs` and observe the denial.

Exercise 10-4 You Cannot Widen

In `main`, narrow `net` to `"api.example.com"`, then call `restrict_to` again on the result with a *different* host. Try to fetch from each host through the doubly-narrowed value and confirm that both are now denied, the intersection is empty.

Exercise 10-5 Deterministic Random

Using `random.with_seed(42)`, print five random integers with `int_range(1, 100)`. Run the program twice and confirm you get the very same five numbers each time, the seeded sequence is repeatable.

Summary

Attenuation is the value-level half of Capa's capability discipline, and it lets you honour the principle of least authority in code. The `restrict_to` family of methods returns a fresh, narrower copy of a capability with the same type, so you can hand a helper authority over one host, one directory, or a few environment variables instead of everything, and unlike an ordinary capability, this freshly made value may be bound to a `let`. Narrowing is monotonic by construction: intersections only shrink, nothing can widen a capability back, and chaining disjoint restrictions ends at authority over nothing. Calls outside the allowed set fail closed at run time, before any system call, returning an ordinary `Err`. Two functions with the same signature can therefore hold very different power, decided by whoever built the value. So far every capability has been one of the built-in nine. In the next chapter you define your own.

CHAPTER 11

Defining Your Own Capabilities

The nine built-in capabilities describe the *primitive* powers a program can hold: the screen, the filesystem, the network, and so on. But the powers your program is really about are usually higher-level, send an email, save a user, charge a card, publish a message. Describing those only as "`Net` plus maybe `Fs`" is far too coarse for a caller to reason about. Capa lets you define **your own capabilities**, as real to the compiler as `Net` or `Fs`, so the authority surface of your code can speak the language of your domain. This is the feature that makes Capa libraries unlike libraries anywhere else.

Declaring a Capability

You declare a new capability with the `capability` keyword, followed by the methods it offers. The declaration is a **contract**: it says what anyone holding this capability is able to do, without saying how.

```
capability SendEmail
    fun send(self,
        to: String,
        subject: String,
        body: String) -> Result<Unit, IoError>
```

From the compiler's point of view `SendEmail` is now a capability, subject to the same discipline as the built-ins: it must arrive through a parameter, it cannot be aliased into a `let`, and a function that declares it must use it. A

function that wants to send mail must receive a `SendEmail`; one that does not, cannot.

Implementing a Capability

A capability declaration is only a contract; something has to fulfil it. An **implementor** is a struct paired with an `impl` block that provides the declared method. Here is an SMTP-based mailer:

```
type SmtpMailer {
    server: String,
    net: Net
}

impl SendEmail for SmtpMailer
    fun send(self,
        to: String,
        subject: String,
        body: String) -> Result<Unit, IoError>
        // a real implementation would speak SMTP over self.net
        return Ok(())
```

Look closely at the struct: it has a `net: Net` field. In Chapter 9 you learned that built-in capabilities **cannot** be stored in struct fields, precisely to stop authority being smuggled sideways. The rule relaxes for one specific, opt-in case: a struct that **implements a user-defined capability** is allowed to hold built-in capabilities as fields. The reason it is safe is that the struct itself now becomes a capability under the discipline, so the authority it wraps cannot leak, it can only be exercised through the methods you declared. This is called the **cap-bearing** pattern.

The Factory Function

How do you create an `SmtpMailer` in the first place, given that it holds a `Net`? Here is the second relaxation: while a built-in capability can never be returned from a function, a **user-defined** one can. So you write an ordinary

factory function that takes the built-in capabilities it needs and returns a fresh implementor:

```
fun make_mailer(net: Net, server: String) -> SmtpMailer
    return SmtpMailer { server: server, net: net }
```

Inside `make_mailer`, the built-in `net` flows into the struct's field, and from that point on the `SmtpMailer` takes over the discipline. This factory is the one doorway through which the high-level capability comes into being.

Using It

From a caller's seat, a user-defined capability behaves exactly like a built-in: declare it as a parameter, call its methods, propagate failures with `?`.

```
fun notify_user(mailer: SendEmail, user_id: String) -> Result<Unit,
IoError>
    mailer.send("${user_id}@example.com", "Welcome", "Hello")?
    return Ok(())

fun main(net: Net, stdio: Stdio)
    let mailer = make_mailer(net, "smtp.example.com")
    match notify_user(mailer, "42")
        Ok(_) -> stdio.println("sent")
        Err(e) -> stdio.eprintln("error: ${e}")
```

Now read the signature of `notify_user`. It declares `SendEmail`, **not** `Net`. Anyone reading it learns the meaningful fact, *this function sends email*, and is spared the irrelevant one, that the transport underneath happens to be the network. The implementation choice is invisible at the call site, which is exactly the point: `SendEmail` is the contract, and the contract is what callers reason about.

> **NOTE The manifest speaks your language**

> When the compiler emits the manifest, `notify_user` appears with `declared_capabilities: ["SendEmail"]`, with no `Net` in sight. An auditor reading the report sees the authority at the level they actually care about, "this sends email", and can drill into a `user_defined_capabilities` section to learn which concrete types implement it. The high-level contract, not the low-level transport, is what surfaces.

Why This Matters for Libraries

This is the unique contribution of Capa. In an ordinary language, a library's authority surface is described in its README and enforced by nothing, you simply trust that the "email" library only sends email. In Capa that surface is a type the compiler enforces and the SBOM records. Library authors get a stable contract ("to use me, give me a `SendEmail`"); callers get real reasoning power ("anything that does not take a `SendEmail` cannot send mail"); auditors get a report whose `declared_capabilities` are meaningful at the level a human cares about.

And the pattern scales to any domain. A database client exposes `capability QueryDB`; a message-queue library exposes `capability PublishMessage`; a token signer exposes `capability SignToken`. None of these could ever be built-ins, the language cannot anticipate every domain, so instead Capa makes it cheap for libraries to add precisely the capabilities their domain needs.

Traits: the Same Idea Without Authority

A capability is a special kind of **trait**, a named set of methods that types can implement. A plain `trait` declares a shared interface for ordinary, authority-free behaviour; the syntax mirrors a capability, but the discipline does not treat it as authority.

```
trait Describe
    fun describe(self) -> String

impl Describe for Rect
    fun describe(self) -> String
        return "${self.width} by ${self.height}"
```

The difference is entirely about authority: a `trait` is a way to share behaviour across types, while a `capability` is a trait that the compiler tracks as a power a function must be granted. Use a `trait` for "these types all know how to describe themselves"; reach for a `capability` when the methods represent an effect on the outside world that you want to appear in signatures and the manifest.

Two Common Errors

The first guards the cap-bearing relaxation. If a struct holds a built-in capability but does **not** implement any user-defined capability, Capa rejects it, the relaxation is opt-in, and you have not opted in:

```
type Holder {
    net: Net
}
```

```
$ capa --run mail.capa
mail.capa:1:6: error: type 'Holder' has a capability field 'net' but
does not
               implement any user-defined capability; capabilities
cannot
               appear as struct fields outside the cap-bearing
pattern
```

The fix is to either drop the capability field or add an `impl SomeCapability for Holder`. The second error is a mismatch between an implementor and its contract: the method you provide must match the declaration exactly, same parameters, same return type.

```
impl SendEmail for SmtpMailer
    fun send(self, to: String) -> Result<Unit, IoError>   // too few
params
        return Ok(())
```

```
$ capa --run mail.capa
mail.capa: error: impl SendEmail for SmtpMailer: method 'send' has
the wrong
           signature; expected (self, String, String, String) ->
Result<Unit,
           IoError>, got (self, String) -> Result<Unit, IoError>
```

Match the declared signature exactly and the error disappears. You may add extra helper methods to the struct freely; the contract only requires that the declared method is present with the right shape.

Try It Yourself

Define a capability, implement it over a built-in, and call it through its high-level contract.

Exercise 11-1 Store a User

Define `capability StoreUser` with one method `save(self, id: String, payload: String) -> Result<Unit, IoError>`. Implement it with a struct `FileStore` that holds an `Fs` and writes one file per user (path like `id + ".json"`).

Exercise 11-2 Factory and Narrowing

Write a factory `make_file_store(fs: Fs) -> FileStore`. In `main`, narrow the `Fs` with `restrict_to("/var/users/")` *before* passing it to the factory, so the whole store is confined to one directory.

Exercise 11-3 Use the Contract

Write `fun register(store: StoreUser, id: String) -> Result<Unit, IoError>` that saves a small payload through the capability. Note that its signature mentions `StoreUser`, not `Fs`, the transport is hidden behind the contract.

Exercise 11-4 A Plain Trait

Define a `trait Area` with `area(self) -> Float`, and implement it for two different shape structs. Write a function that takes an `Area` and prints its area. Notice this needs no capability at all, computing an area is authority-free.

Exercise 11-5 Read the Manifest

Run `capa --manifest` on your program from 11-3 and confirm that `register` lists `StoreUser` (not `Fs`) under its declared capabilities, and that the implementor appears in the user-defined capabilities section.

Summary

You can now extend the capability system itself. The `capability` keyword declares a new authority as a contract of methods; a struct plus an `impl Cap for Struct` fulfils it, and, through the opt-in cap-bearing relaxation, such a struct may hold built-in capabilities as fields. A factory function, which may return a user-defined capability even though it could never return a built-in one, is the doorway that creates an implementor from the primitives it needs. Callers then declare the high-level capability, `SendEmail`, `StoreUser`, in their signatures, so both code and manifest describe authority in the language of your domain rather than the runtime's. You also met the plain `trait`, the

same shape without the authority. This is the distinctive power of Capa libraries. With the language itself now fully in hand, the next chapter steps up to organising real programs: modules, visibility, and packages.

CHAPTER 12

Modules, Visibility and Packages

Every program in this book has lived in a single file. That is fine for learning, but real projects grow past what one file can comfortably hold, and they lean on code written by other people. Capa handles both with a small, clear set of tools: **modules** split your code across files, **visibility** decides which names a file shares with the rest, and **packages** pull in code from elsewhere. This chapter ties them together.

Splitting a Program Across Files

Each `.capa` file is a **module**. To use the contents of one file from another, you `import` it. Put a helper in `util.capa`:

```
// util.capa
pub fun greet(name: String) -> String
    return "Hello, ${name}"
```

Then use it from `main.capa` in the same folder:

```
// main.capa
import util

fun main(stdio: Stdio)
    stdio.println(greet("Capa"))
```

```
$ capa --run main.capa
Hello, Capa
```

Three things to notice. The `import util` resolves to `./util.capa`, imports are relative to the file that writes them. The imported function is reachable directly by its plain name, `greet(...)`, with no module prefix required. And `greet` was marked `pub`, which is what made it visible at all, as we are about to see.

Private by Default

Remove the `pub` keyword from `greet` and the program stops compiling:

```
// util.capa
fun greet(name: String) -> String
    return "Hello, ${name}"
```

```
$ capa --run main.capa
main.capa:3:19: error: undefined name 'greet'
                (private to module 'util'; mark it 'pub' to expose)
```

In Capa, **every top-level item is private by default**. Only the ones you mark `pub` (functions, types, capabilities, and so on) can be reached from another module. This is the opposite of Python, where everything in a file is importable unless its name starts with an underscore. Capa's default makes you decide, deliberately, what your module's surface is: the `pub` names are your **contract**, the promise other code may depend on, while everything you leave private is implementation detail you can rename or rework freely without breaking anyone.

Folders and Dotted Imports

Larger projects nest files in folders, and the import path mirrors the folder structure with dots. `import util.strings` resolves to `./util/strings.capa`:

```
my-project/
    main.capa
    util/
        strings.capa
        math.capa
```

```
// main.capa
import util.strings
import util.math

fun main(stdio: Stdio)
    stdio.println(capitalise("capa"))   // from util/strings.capa
    stdio.println("${double(7)}")        // from util/math.capa
```

There is one `import` per line, with no wildcard imports. Imports can be **transitive**, a file you import may import others, and the loader follows the whole chain; if two files import each other in a cycle, Capa detects it and reports a precise error rather than looping forever.

Qualified Names and Aliases

You can also call an imported function with its module named explicitly, which is handy when you want the reader to see where a name comes from:

```
import util

fun main(stdio: Stdio)
    stdio.println(util.greet("Capa"))   // same as greet("Capa")
```

When two modules export the same name, qualify or **alias** to tell them apart. `import ... as` gives a module a short local name; without `as`, the default alias is the last segment of the dotted path (so `import util.strings` can be used as `strings.capitalise(...)`).

```
import util.strings as S
import third_party as TP
```

```
fun main(stdio: Stdio)
    stdio.println(S.capitalise("capa"))
    stdio.println(TP.capitalise("capa"))
```

Sharing Modules with CAPA_PATH

The loader looks for an import next to the importing file first. If you keep a folder of modules you reuse across projects, a personal mini-library, point the `CAPA_PATH` environment variable at it and the loader will search there too. Project-local files always win over `CAPA_PATH` entries, so a local file shadows a shared one of the same name.

```
$ export CAPA_PATH=/usr/local/share/capa:./vendor
$ capa --run app.capa
```

Packages: Depending on Other People's Code

For real dependencies, code maintained by others and pinned to a specific version, Capa projects use a **package** file named `capa.toml` at the project root. It names your project and lists what it depends on, with each dependency pinned to an exact git tag:

```
# capa.toml
[package]
name = "my-project"
version = "0.1.0"
capa = ">=1.0.0"

[dependencies]
capa_log = { git = "https://github.com/nelsonduarte/capa_log", tag =
"v0.1.2" }
```

Three commands cover daily work. `capa install` reads `capa.toml`, fetches each dependency into a local `./vendor/` folder, and writes a `capa.lock` file that pins the exact resolved version (down to a content hash) so that every

later install is bit-for-bit reproducible. `capa add <name>` adds a dependency from the command line, resolving its short name through a signed registry. And `capa --run` runs your program with the vendored dependencies on the path.

```
$ capa add capa_log
$ capa install
$ capa --run main.capa
```

Test-only and tooling dependencies go in a separate `[dev-dependencies]` table; they are pulled in when you build the project itself but never inflicted on anyone who depends on *your* package. The lock file's content hashes, together with optional signature verification on the registry, are what make a Capa dependency tree something you can reproduce and audit rather than merely hope about.

> **NOTE Imports make the authority surface visible**
>
> Modules and capabilities reinforce each other. When you pull in a Capa library, its capability surface travels with it: a library whose functions declare no capabilities is *provably* pure, while one that needs the network must say `Net` in its signatures, where you, and the manifest, can see it. So the same `import` that brings in code also brings in an honest, machine-checkable statement of what that code is allowed to do, the supply-chain guarantee from Chapter 9, now spanning your dependencies.

Two Common Errors

The first is calling a private name from outside its module, the missing-`pub` case from earlier, which Capa diagnoses specifically rather than as a generic typo:

```
error: undefined name 'helper'
```

```
        (private to module 'util'; mark it 'pub' to expose)
```

The second is a name conflict, when two imported modules both export the same name and Capa cannot tell which you mean:

```
import a    // exports `helper`
import b    // also exports `helper`
```

```
$ capa --run main.capa
error: name conflict: 'helper' declared in a.capa and b.capa.
       Either rename one or pick the import explicitly.
```

Resolve it by renaming one of the functions, or by aliasing one import (`import a as A`) and reaching for the conflicting name qualified (`A.helper(...)`).

Designing Good Modules

A few habits keep a multi-file project healthy. Give each module a clear, single responsibility, string handling in one file, the data model in another, so a reader knows where to look. Keep the `pub` surface as small as the module's job allows; every public name is a promise you will have to keep, while private ones stay yours to change. And let the imports at the top of a file serve as a quick table of contents for its dependencies, which is easiest when each module pulls in only what it truly uses.

Try It Yourself

These exercises use more than one file, so make a small folder for each.

> **Exercise 12-1 Two Files**

Create `mathutil.capa` with a `pub fun square(n: Int) -> Int`, and a `main.capa` that imports it and prints a few squares. Then remove the `pub` and read the error you get.

Exercise 12-2 A Folder of Helpers

Make a `text/` folder containing `case.capa` (with a `pub` function that upper-cases a string) and `pad.capa` (with a `pub` function that surrounds a string in brackets). Import both into `main.capa` with dotted imports and use them together.

Exercise 12-3 Qualified Calls

Import a module two ways, plainly and `as` an alias, and call the same function both by its bare name and through the alias, confirming they do the same thing.

Exercise 12-4 Split the Mailer

Take the email example from Chapter 11 and split it across files: `mailer.capa` exports `pub capability SendEmail` and `pub fun make_mailer(...)`; `users.capa` exports a function that takes a `SendEmail` and sends a notification; `main.capa` wires them together. Notice that only the names you marked `pub` cross the file boundary.

Exercise 12-5 A Dependency

Create a `capa.toml` for a small project and add a dependency on a registry library (for example `capa_log`) with `capa add`. Run `capa install`, then import and use the library from `main.capa`.

Summary

Your code is no longer confined to one file. Each `.capa` file is a module; `import` brings another module's names into scope, plainly or qualified or under an alias, and folders map to dotted import paths. Visibility is private by default, so only the items you mark `pub` form your module's contract while everything else stays free to change, the reverse of Python's open-by-default rule. For code from outside your project, `CAPA_PATH` shares modules across projects and a `capa.toml` with `capa install` pins real dependencies to exact, reproducible versions. Best of all, importing a library also imports its honest capability surface, extending Capa's supply-chain guarantee across everything you depend on. Two chapters of Part I remain. The next returns to security at the value level: not just which effects a function may use, but where its data is allowed to flow.

CHAPTER 13

Information-Flow Control

Capabilities answer one security question: *which effects may a function use?* Can it touch the network, the filesystem, the screen? But there is a second question they do not address: *where is a particular piece of data allowed to go?* A function might legitimately hold both an API key and the `Net` capability, and still you would not want the key to be the thing it sends over that network. This chapter is about Capa's answer to the second question, **information-flow control** (IFC): a way to label sensitive data and have the compiler prove it never escapes.

Two Labels: public and secret

IFC works with a tiny two-level scale of sensitivity. Every value is either `@public` (ordinary, may go anywhere) or `@secret` (sensitive, must not leak). `@public` sits below `@secret`. You mark the data that matters, a password, a card number, an API token, by annotating its type, a parameter, or a struct field with `@secret`, and the compiler tracks the label through everything that touches it.

```
fun handle(token: @secret String)
    // `token` is secret; the compiler now follows it everywhere
    ...
```

One source of secrets is automatic: reading an environment variable with `env.get` returns a `@secret` value with no annotation needed at all, because environment variables so often hold keys and tokens. That single default heads off one of the most common leaks, an API key accidentally written to a log.

Labels Spread by Themselves

You do not have to annotate every downstream value by hand. A secret label **propagates** automatically to anything derived from a secret, by a rule called **join**: combine a public value with a secret one and the result is secret. This applies through arithmetic, string interpolation, field reads, and function calls, anything that lets data flow from one value into another.

```
fun main(stdio: Stdio)
    let token: @secret String = "abc123"
    let header = "Bearer ${token}"   // header is now @secret by join
    ...
```

Here `header` was built by interpolating a secret into an otherwise public string, so the whole of `header` becomes secret. There is no way to quietly "wash off" the label by mixing the secret into something larger; the taint follows the data wherever it goes.

Sinks: Where Data Leaves the Program

A **sink** is any point where data leaves your program for the outside world. The built-in sinks are the obvious exits: `Stdio`'s `print`, `println`, and `eprintln`; `Net`'s `get` and `post`; `Fs`'s `write`; and a database's `exec` and `query`. The single rule of IFC is this: **a `@secret` value may never reach a sink.** Try it, and the compiler stops you.

```
fun log_token(stdio: Stdio, token: @secret String)
    stdio.println("token is ${token}")   // a secret reaches a public
sink
```

```
$ capa --run secrets.capa
secrets.capa:2:5: information-flow violation: a @secret value reaches
                  Stdio.println, a public sink
```

The interpolated message is secret (by join), and `println` is a public sink, so the flow is forbidden. Notice what this buys you: the leak is caught at compile time, in a function that legitimately holds both `stdio` and the secret. Capabilities alone could not catch it, the function is allowed to print, just not *this*.

Warn, Then Enforce

By default an information-flow violation is a **warning**: Capa points it out but still builds, so you can adopt IFC gradually on an existing codebase. When you want the guarantee to be ironclad, mark a function with the `@strict_ifc()` attribute, and violations inside it become hard **errors** that refuse to compile.

```
@strict_ifc()
fun handle(stdio: Stdio, token: @secret String)
    stdio.println(token)   // now a compile error, not just a warning
```

Under `@strict_ifc` the compiler also catches **implicit flows**, the sneakier case where a secret influences control flow rather than data. Printing a fixed message *inside an `if` guarded by a secret condition* still leaks one bit of the secret, whether the branch was taken, and strict mode rejects that too.

The One Sanctioned Exit: declassify

Sometimes a secret legitimately must become public, you genuinely intend to display the last four digits of a card, or a redacted summary. For that there is exactly one sanctioned bridge: the `declassify` function. It turns a `@secret` value into a `@public` one, and it **demands a written reason**.

```
fun main(stdio: Stdio, card: @secret String)
    let safe = declassify(mask_card(card),
                          reason: "PCI: only last 4 digits retained")
    stdio.println(safe)   // allowed: the value was deliberately
declassified
```

The required `reason` is not a comment; it is part of the record. **Every `declassify` site is recorded in the SBOM** under the function's `declassifications` list, with its reason, the value involved, and its exact source position (pos); the top-level summary.declassification_sites counts them. The result is a machine-checkable disclosure log: a precise, auditable answer to *where, and why, does this program reveal sensitive data?* The proof travels with the manifest, not buried in a code-review thread.

> **NOTE Secrets do not launder through containers**
>
> You cannot escape a label by hiding a secret inside a structure. A secret stashed in a struct, a list, a tuple, a map, or a set, or one iterated over in a `for` loop, stays secret; the label follows the data out the other side. The only way from secret to public is `declassify`, in plain sight, with a reason.

Honest Boundaries

It is worth knowing where the analysis draws its lines, because Capa is careful not to overpromise. IFC is **intra-procedural**: it tracks flows within a single function, so for a secret to cross into another function, that function must accept it through an explicit `@secret` parameter. This is by design, it keeps every signature honest about the sensitive data it receives, rather than letting secrecy leak silently across call boundaries. A struct field *declared* `@secret` keeps its label per field, even when you read or destructure it, so it cannot be laundered through a public sibling; secrets placed at run time into undeclared fields, or inside lists, maps, and tuples, are tracked more coarsely, as a whole. Knowing these edges lets you lean on the guarantee exactly where it is strongest.

Why This Matters

Put the two axes together and Capa answers a question almost no mainstream language can answer by construction: *can this function read secret X and send it over the network?* Because `env.get` is secret by default, an API key cannot slip into a log line by accident. Because card data can be marked `@secret`, a payments core can **prove** at compile time that a card number never reaches a log or a network call unless it was first masked and deliberately declassified, and the SBOM then lists every point of deliberate disclosure. Capabilities tell you what a function *can do*; information-flow control tells you where its data *can go*. Together they make the security posture of a program something you can read off the types, not something you hope a reviewer noticed.

Try It Yourself

Run these and read the warnings or errors carefully; learning to recognise an information-flow violation is the whole skill.

Exercise 13-1 Catch a Leak

Write `fun show(stdio: Stdio, secret_word: @secret String)` that tries to `println` the secret directly. Run it and read the information-flow violation, then add `@strict_ifc()` above the function and watch the warning become a hard error.

Exercise 13-2 Join in Action

Inside a function with a `@secret name`, build a greeting `"Hello, ${name}"` and try to print it. Confirm the violation still fires, the greeting inherited the secret label by join, even though the rest of the string is public.

Exercise 13-3 Declassify with a Reason

Write a `mask(secret: @secret String) -> String` that returns only the first character followed by `"***"`. In `main`, `declassify` its result with a clear `reason` and print the masked value successfully.

Exercise 13-4 An Environment Secret

Recall that `env.get` returns a secret value. Read a variable, then attempt to print it directly and observe the violation. Decide whether printing it is ever appropriate, and if so, declassify it with an honest reason.

Exercise 13-5 No Laundering

Put a `@secret` value into a one-element list, then take it back out and try to print it. Confirm that hiding the secret in a container did not remove its label.

Summary

Information-flow control is Capa's second security axis, governing not which effects a function may use but where its data may travel. Values carry one of two labels, `@public` or `@secret`; you annotate sensitive data, and `env.get` is secret by default. The secret label propagates automatically by join through arithmetic, interpolation, field reads, and calls, so anything derived from a secret stays secret. A `@secret` value reaching a sink, `Stdio`, `Net`, `Fs.write`, a database, is a violation, a warning by default and a hard error under `@strict_ifc()`, which also catches implicit flows through secret-guarded branches. The single sanctioned way from secret to public is `declassify`, which demands a reason and records every use in the SBOM as an auditable disclosure log; secrets cannot be laundered through containers, and the

analysis stays honest by tracking flows within a function and requiring explicit `@secret` parameters across boundaries. With this you have seen the core of the language. One chapter of Part I remains: how to test the programs you write.

CHAPTER 14

Testing Your Code

A program you cannot test is a program you are afraid to change. **Tests** are small pieces of code that check your real code does what you claim, automatically, as often as you like. They catch mistakes the moment you make them and give you the confidence to refactor, knowing that if you break something a test will tell you. Capa's testing story is refreshingly simple: a test is just a program that either succeeds or stops with an error. This final chapter of Part I shows you how to write and run them.

The Idea: Succeed, or Panic

Capa needs almost no new machinery for testing, because it reuses something you have already met: `panic`. Calling `panic(message)` stops the program immediately, prints `panic: ` followed by your message to the error stream, and exits with a non-zero status that signals *failure*. A program that runs to the end without panicking exits with status zero, which signals *success*.

```
fun withdraw(balance: Int, amount: Int) -> Int
    if amount > balance
        panic("withdraw of ${amount} exceeds balance ${balance}")
    return balance - amount
```

That single pair of outcomes, finish cleanly or panic, is the whole basis of testing in Capa. A **test passes when its program exits successfully and fails when it panics.**

Writing a Test

Tests live in a `tests/` folder at the root of your project, in files whose names start with `test_`. Each test file is an ordinary Capa program with a `main`; it exercises some code and panics if the result is not what it should be. Suppose you have a `double` function to check:

```
// tests/test_math.capa
fun double(n: Int) -> Int
    return n * 2

fun main(stdio: Stdio)
    if double(3) != 6
        panic("double(3) should be 6")
    if double(0) != 0
        panic("double(0) should be 0")
    stdio.println("all checks passed")
```

Each `if` is a check: if reality disagrees with the expectation, the test panics with a message explaining what went wrong. If every check passes, `main` reaches the end and the program exits successfully.

Running Your Tests

The `capa test` command finds every `tests/test_*.capa` file in your project, runs it, and reports which passed and which failed. When a test fails, its captured output is printed inline so you can see exactly what happened.

```
$ capa test
running 1 test
  test_math ... ok

1 passed, 0 failed
```

Change `double` to `n * 3` and rerun, and the same command reports the failure, with your panic message, so you know precisely which expectation broke.

A Reusable Check Helper

Writing `if ... panic(...)` for every check is repetitive. A small helper makes tests read better: it prints a tick for a passing check and panics for a failing one.

```
fun check(stdio: Stdio, name: String, condition: Bool)
    if condition
        stdio.println("ok: ${name}")
    else
        panic("FAILED: ${name}")

fun main(stdio: Stdio)
    check(stdio, "double of 3", double(3) == 6)
    check(stdio, "double of 0", double(0) == 0)
```

> **NOTE The capa_test library**
>
> You do not have to hand-roll this. The registry library `capa_test` provides a tiny set of ready-made assertions for exactly this purpose. Because it is needed only while testing, you add it under the `[dev-dependencies]` table of your `capa.toml` (the dev table from Chapter 12), so it is pulled in for your own test runs but never forced on anyone who depends on your package.

Testing Functions That Can Fail

Functions that return an `Option` or a `Result` are tested the same way: call them, inspect the outcome, and panic if it is wrong. Pattern matching makes the intent clear, and you should test both the success and the failure path.

```
fun safe_divide(a: Float, b: Float) -> Option<Float>
    if b == 0.0
        return None
    return Some(a / b)

fun main(stdio: Stdio)
```

```
    match safe_divide(10.0, 2.0)
        Some(r) -> check(stdio, "10/2 is 5", r == 5.0)
        None -> panic("10/2 should be Some")
    match safe_divide(1.0, 0.0)
        Some(_) -> panic("1/0 should be None")
        None -> check(stdio, "1/0 is None", true)
```

Pure functions like `safe_divide` are the easiest things in the world to test: they take no capabilities, so a test just feeds them inputs and checks outputs, with no files, network, or clock to arrange. This ease is one more quiet reward of keeping logic pure and pushing the capabilities to the edges of your program.

Testing on Both Backends

Capa can run your code two ways: through its default Python backend and through a WebAssembly backend. The test runner can target either, or both at once.

```
$ capa test            # the default (Python) backend
$ capa test --wasm     # the WebAssembly backend
$ capa test --both     # run on both, and compare their output
```

The most interesting of these is `capa test --both`. It runs every test on both backends and then **compares their output**, reporting any difference as its own kind of failure. Because Capa promises that a program produces byte-for-byte identical results on either backend, this is the cheapest possible check that your code does not accidentally depend on one of them. For most projects, plain `capa test` is the daily command; reach for `--both` when backend parity matters.

What Makes a Good Test

A few habits make tests genuinely useful rather than mere ceremony. Test one behaviour per check, with a message that names it, so a failure points

straight at the cause. Cover the **edge cases**, the empty list, the zero, the missing key, since that is where bugs hide, not in the happy path you already had in mind. Write a test the moment you fix a bug, encoding the very mistake you just made so it can never silently return. And keep tests fast and independent: each should set up what it needs and not rely on another having run first. Above all, run `capa test` often, ideally after every meaningful change, so that breakage is caught while the cause is still fresh in your mind.

Try It Yourself

Create a `tests/` folder and put each of these in a `test_*.capa` file, then run `capa test`.

Exercise 14-1 First Test

Write a function `add(a: Int, b: Int) -> Int`, and a test that panics unless `add(2, 3)` equals `5` and `add(-1, 1)` equals `0`. Run `capa test` and confirm it passes; then break `add` on purpose and watch the test fail.

Exercise 14-2 Use a Helper

Add the `check(stdio, name, condition)` helper from this chapter and rewrite your tests from 14-1 to use it, so each check prints a labelled `ok` line or panics with a clear name.

Exercise 14-3 Test the Edges

Write `first_or(xs: List<Int>, default: Int) -> Int` that returns the first element or the default. Test it on a non-empty list **and** on the empty list, the edge case is the whole point.

Exercise 14-4 Test a Result

Take the `check_age` function from Chapter 8 (returning `Result<Int, String>`) and write tests covering a valid age, a negative age, and an under-18 age, matching on the result each time.

Exercise 14-5 Both Backends

Run any of your test files with `capa test --both` and confirm the output is identical on both backends. (If you have not installed the WebAssembly backend, note what the command reports.)

Summary

Testing in Capa rests on one simple idea: a test is a program that succeeds by finishing and fails by calling `panic`. You write tests as `tests/test_*.capa` files whose `main` exercises your code and panics when an expectation is unmet, run them all with `capa test`, and read the inline output of any that fail. A small `check` helper, or the `capa_test` library added under `[dev-dependencies]`, makes the checks read cleanly; pure functions are trivially testable because they need no capabilities, and functions returning `Option` or `Result` are tested by matching both paths. The runner can target the Python backend, the WebAssembly backend, or both at once with `capa test --both`, which verifies the two produce identical output. With testing, Part I is complete: you have learned the core of the language, from values to capabilities to information flow, and how to organise and verify a program built from it. In Part II we put all of it to work, building real projects from start to finish.

PART II

Projects

Three complete programs, built end to end.

CHAPTER 15

Project 1: A Grade-Book Tool

Welcome to Part II. You have learned the core of the language; now you will use it. The remaining chapters build complete, working programs from start to finish, the way you would write them in practice, weaving together values, collections, control flow, functions, types, error handling, capabilities, and tests. Read with your hands: type the code, run it, break it, fix it. That is where the learning sets.

Our first project is a **grade-book tool**: a small command-line program that reads student scores from a file, computes each student's average and pass/fail status, writes a report to a second file, and prints a summary to the screen. It is modest enough to finish in three chapters, yet it touches most of Part I. This chapter plans the project and builds the part that turns raw text into typed data.

The Brief

The tool reads a file named `scores.txt`, with one student per line in the shape `Name,score1,score2,score3` (any number of scores). It computes each student's average, marks them as passing when the average is at least `10`, writes a report to `report.txt`, and prints the same report to the terminal. Here is the input it should accept:

```
Ana,17,15,19
Bruno,8,11,9
Carla,14,13,15
```

And the output it should produce:

```
Ana: 17.0 (Pass)
Bruno: 9.33 (Fail)
Carla: 14.0 (Pass)

Passed: 2/3
```

Planning the Shape

A good first step in any project is to decide its shape before writing code. We will split this tool across three files, each with one job, which keeps every piece small and, as you will see, keeps the parts that touch the outside world cleanly separated from the parts that do not.

- `parse.capa` turns one line of text into a typed `Student` record. Pure: no capabilities.
- `report.capa` turns a list of students into the report text. Also pure.
- `main.capa` is the entry point: it reads the file, calls the other two, and writes the output. This is the only file that holds capabilities.

Notice the deliberate design: all the *logic*, parsing and reporting, is pure and authority-free, while all the *effects*, reading and writing files, printing, live in `main`. This is a pattern worth forming as a habit, because it makes the logic trivial to test and lets the capability manifest prove that your core computation cannot touch the outside world. This chapter writes `parse.capa`.

> **NOTE Set up the project**
>
> Create a folder for the project, for example with `capa init gradebook`, and `cd` into it. Create `parse.capa`, `report.capa`, and `main.capa` as you reach

each one, plus a `scores.txt` with the three sample lines above so you have something to run against.

The Data Model

Before parsing anything, decide what a parsed line *is*. A student has a name, a list of scores, and two values we will compute from those scores, the average and whether they passed. A struct captures this exactly:

```
// parse.capa
pub type Student {
    name: String,
    scores: List<Int>,
    average: Float,
    passed: Bool
}
```

The `pub` keyword makes `Student` visible to the other files that will import `parse`. We store the derived `average` and `passed` right in the record, so the reporting code later does not have to recompute them, the parser works them out once, when it has the scores in hand.

Parsing One Line

Now the core of this chapter: a function that takes one line of text and produces a `Student`, or nothing if the line is malformed. Because a line might not parse, the return type is `Option<Student>`. Here is the whole function; we will read it piece by piece afterwards.

```
// parse.capa
pub fun parse_line(line: String) -> Option<Student>
    let parts = line.split(",")
    if parts.length() < 2
        return None
```

```
    let name = parts.get(0)?
    var scores: List<Int> = []
    var sum = 0
    var i = 1
    while i < parts.length()
        let token = parts.get(i)?
        let n = parse_int(token)?
        scores.push(n)
        sum = sum + n
        i = i + 1
    if scores.is_empty()
        return None
    let avg = to_float(sum) / to_float(scores.length())
    return Some(Student {
        name: name,
        scores: scores,
        average: avg,
        passed: avg >= 10.0
    })
```

Read it from the top. `line.split(",")` breaks the line into pieces at each comma (Chapter 2); a line with fewer than two pieces has no scores, so we bail out with `None`. `parts.get(0)?` reads the name: `get` returns an `Option` (Chapter 3), and the `?` unwraps it or propagates `None` if somehow absent (Chapter 8).

The `while` loop (Chapter 5) walks the remaining pieces, starting at index `1`. For each, `parse_int(token)?` converts the text to an integer, and here the `?` earns its keep: if any score is not a valid number, the whole `parse_line` returns `None` at once, exactly the behaviour we want for a malformed line. Valid scores are pushed onto the `scores` list and added to a running `sum`.

Finally we guard against a line with a name but no scores, compute the average with an explicit `to_float` on both sides (Capa never mixes `Int` and `Float`, Chapter 2), and return a fully-built `Student`, with `passed` set by the comparison `avg >= 10.0`. Every field is filled in one place, the moment the data is available.

> **NOTE Why Option, not panic**
>
> A malformed line should not crash the whole program; it should simply be skipped. Returning `Option<Student>` lets the caller decide what to do with a line that does not parse, which, as we will see in Chapter 17, is to ignore it and carry on. This is errors-as-values (Chapter 8) doing exactly the job it was made for.

Testing the Parser

The parser is pure, so it is the easiest possible thing to test (Chapter 14): feed it a line, check the record it returns. Put a quick test in `tests/test_parse.capa`.

```
// tests/test_parse.capa
import parse

fun main(stdio: Stdio)
    match parse_line("Ana,18,16")
        Some(s) ->
            if s.name != "Ana"
                panic("name should be Ana")
            if s.average != 17.0
                panic("average should be 17.0")
            if not s.passed
                panic("Ana should pass")
        None -> panic("Ana should parse")
    match parse_line("")
        Some(_) -> panic("empty line should not parse")
        None -> stdio.println("all parse tests passed")
```

```
$ capa test
running 1 test
  test_parse ... ok

1 passed, 0 failed
```

The test checks a well-formed line in detail and confirms that an empty line produces `None`. With the parser proven, we have a solid foundation: text in, typed `Student` out, malformed input safely rejected.

Try It Yourself

Extend the parser. Each task is a small, self-contained change with a test to match.

Exercise 15-1 Trim the Name

Real files have stray spaces. Use the string method `trim` so that a line like `Ana ,17,15` still yields the name `Ana`. Add a test for it.

Exercise 15-2 Reject Negative Scores

A score below zero is surely a mistake. Make `parse_line` return `None` if any parsed score is negative, and add a test that a line with a negative score does not parse.

Exercise 15-3 Highest and Lowest

Add two fields to `Student`, `best: Int` and `worst: Int`, and compute them in `parse_line` while you walk the scores. Test that for `Ana,10,20,15` the best is `20` and the worst is `10`.

Exercise 15-4 A Configurable Pass Mark

Right now passing is hard-coded at `10.0`. Change `parse_line` to take a second parameter `pass_mark: Float` and use it for the `passed` field. Update the test to pass a mark explicitly.

Summary

You started Part II by planning a real program and building its foundation. You designed the project as three files with a deliberate split, pure parsing and reporting logic separated from the capability-holding entry point, a habit that makes code testable and keeps your core provably free of authority. You modelled a parsed line as a `Student` struct that carries both the raw scores and the derived average and pass/fail status, and you wrote `parse_line`, which turns one line into an `Option<Student>`, using `split`, `get`, `parse_int`, and the `?` operator so that any malformed line is cleanly rejected rather than crashing the program. Finally you tested the parser, which was trivial precisely because it is pure. Next we turn a list of these records into the report the tool prints.

CHAPTER 16

Generating the Report

In the last chapter you built `parse.capa`, which turns one line of text into a `Student` record. The next piece of the grade-book tool takes a whole *list* of those records and produces the report text, the lines you saw in the brief, ending with a pass count. Like the parser, this code is **pure**: it takes data in and returns a string, touching nothing in the outside world. That makes it the second of our three files, `report.capa`, and another thing that is delightfully easy to test.

Reusing the Student Type

The report works with the `Student` type defined in `parse.capa`, so the first line of `report.capa` imports that module to bring the type into scope (Chapter 12). Because `Student` was marked `pub`, the import can see it.

```
// report.capa
import parse
```

Building the Report String

The job is to turn a `List<Student>` into one big string. We build it up piece by piece: a line per student, then a blank line, then a summary count. Here is the whole function.

```
// report.capa
pub fun render(students: List<Student>) -> String
    var out = ""
    for s in students
        let status = if s.passed then "Pass" else "Fail"
```

```
        out = out + "${s.name}: ${s.average} (${status})\n"
    let passed = students.filter(fun (s: Student) -> Bool =>
s.passed).length()
    out = out + "\nPassed: ${passed}/${students.length()}\n"
    return out
```

Read it through. We start with an empty string in a `var` called `out`, because we will be changing it as we go (Chapter 2). The `for` loop (Chapter 5) visits each student; for each one, the `if ... then ... else` *expression* (Chapter 5) picks the word `"Pass"` or `"Fail"` and binds it to `status`, and we append a formatted line to `out` using interpolation, with `\n` for a line break (Chapter 2).

After the loop, we count how many students passed. The expression `students.filter(fun (s: Student) -> Bool => s.passed)` keeps only the passing students (Chapter 3), and `.length()` counts them, all in one readable line. We append the summary and return the finished string. Note that the function never recomputes an average or a pass/fail decision: the parser already stored those on each `Student`, so the report simply reads them.

> **NOTE About that long decimal**
>
> When you run the finished tool, Bruno's average prints as `9.333333333333334`, not the tidy `9.33` from the brief. That is just how a `Float` displays by default. Rounding it to two places takes a small helper, and is a satisfying exercise at the end of this chapter; the core tool works correctly either way.

Testing the Report

Because `render` is pure, a test only has to build a couple of `Student` values, call `render`, and check the result. We construct the records by hand here

(filling every field, since structs have no defaults, Chapter 7) rather than going through the parser, so the test exercises *only* the reporting logic.

```
// tests/test_report.capa
import parse
import report

fun main(stdio: Stdio)
    let ana = Student { name: "Ana", scores: [17],
                        average: 17.0, passed: true }
    let bob = Student { name: "Bob", scores: [8],
                        average: 8.0, passed: false }
    let out = render([ana, bob])
    if not out.contains("Ana: 17.0 (Pass)")
        panic("missing Ana line")
    if not out.contains("Bob: 8.0 (Fail)")
        panic("missing Bob line")
    if not out.contains("Passed: 1/2")
        panic("wrong pass count")
    stdio.println("all report tests passed")
```

```
$ capa test
running 2 tests
  test_parse  ... ok
  test_report ... ok

2 passed, 0 failed
```

We check for the substrings we expect rather than the entire string, which keeps the test robust against incidental spacing while still pinning down the important content. With both pure pieces, parsing and reporting, written and tested, only the entry point remains: the file that actually reads and writes to disk. That is the next chapter.

Try It Yourself

Each task enriches the report. Add a test for every one.

Exercise 16-1 A Header Line

Begin the report with a title line such as `=== Grade Report ===` followed by a blank line, before the per-student lines. Update the test to check the header is present.

Exercise 16-2 Class Average

After the pass count, append a line with the average of all the students' averages. Use `fold` over the list to total the averages (Chapter 3), then divide by the count with `to_float`.

Exercise 16-3 Letter Grades

Add a letter to each line based on the average: A for 18+, B for 16+, C for 14+, D for 10+, otherwise F. Write a small pure helper `grade(avg: Float) -> String` with an `if-elif-else` chain and call it from `render`.

Exercise 16-4 Two Sections

Instead of one list, produce two labelled sections, `Passing:` and `Failing:`, by filtering the students twice and looping over each group separately.

Exercise 16-5 Round to Two Places

Write `fun round2(x: Float) -> Float` that returns `x` rounded to two decimals (hint: `to_float(to_int(x * 100.0 + 0.5)) / 100.0`), and use it on the average in `render` so Bruno shows as `9.33`. Test that `round2(9.3333)` equals `9.33`.

Summary

You wrote the second pure piece of the grade-book tool. `report.capa` imports the `Student` type and provides `render`, which folds a list of students into the report string: a `for` loop builds one line per student using the `if ... then ... else` expression for the pass/fail label, and a single `filter().length()` computes the pass count. Because the parser already stored each average and pass flag, the report only reads them. You then tested `render` by constructing a couple of records by hand and checking the output contains the right lines, easy and fast precisely because the function is pure. Two of the three files are done and verified. In the final chapter of this project we write `main.capa`: the entry point that holds the capabilities, reads the input file, calls these pure functions, writes the report, and does it all with least authority and honest error handling.

CHAPTER 17

Files, Capabilities, and Robustness

Two of the three files are written, and both are pure, no files, no screen, no authority of any kind. Now we add the part that connects them to the real world: `main.capa`, the entry point. This is the *only* file that holds capabilities, and it is where the grade-book tool reads `scores.txt` from disk, calls the parser and the reporter, writes `report.txt`, and prints the summary. We will also do it carefully, with least authority and honest error handling, and then read the manifest to see the discipline pay off.

Loading the Students

First, a function that reads the input file and turns it into a `List<Student>`. Reading a file needs the `Fs` capability (Chapter 9) and can fail, so the function takes an `Fs` and returns a `Result` (Chapter 8).

```
// main.capa
import parse
import report

fun load_students(fs: Fs, path: String) -> Result<List<Student>,
IoError>
    let text = fs.read(path)?
    var students: List<Student> = []
    for line in text.split("\n")
        match parse_line(line)
            Some(s) -> students.push(s)
            None -> ()      // skip empty or malformed lines
    return Ok(students)
```

`fs.read(path)?` reads the whole file; the `?` propagates an `IoError` immediately if the file is missing or unreadable. We split the text into lines and run each through `parse_line` from Chapter 15. The `match` is where the parser's `Option<Student>` pays off: a line that parses is pushed onto the list; a line that does not, an empty line, a blank trailing line, a typo, matches `None`, and we do nothing for it (the `()` is the Unit value, our way of saying "skip this one"). One malformed line never derails the whole run.

The Entry Point

Now `main` itself. It declares the two capabilities the whole program is built from, `Fs` and `Stdio`, narrows the filesystem to the current directory, loads the students, renders the report, writes it, and prints it, handling every failure explicitly as it goes.

```
// main.capa
fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    match load_students(work_dir, "scores.txt")
        Err(e) ->
            stdio.eprintln("read failed: ${e}")
        Ok(students) ->
            let output = render(students)
            match work_dir.write("report.txt", output)
                Err(e) -> stdio.eprintln("write failed: ${e}")
                Ok(_) -> stdio.println(output)
```

Three details are worth calling out, because together they are what writing Capa in the small looks like. First, `main` takes only `Fs` and `Stdio`; the entire program is built from those two capabilities and nothing else. Second, `fs.restrict_to("./")` narrows the filesystem to the working directory before any reading or writing (Chapter 10), so even if a future change accidentally introduced a path like `../../etc/passwd`, the narrowed `Fs` would deny it before any system call. Third, every failure is handled through

`Result` and a `match`, there is no exception, no crash, no silent error; a missing file prints a clear message and the program ends cleanly.

Running the Tool

With `scores.txt` in place (the three sample lines from Chapter 15), run the whole thing:

```
$ capa --run main.capa
Ana: 17.0 (Pass)
Bruno: 9.333333333333334 (Fail)
Carla: 14.0 (Pass)

Passed: 2/3
```

The same text is now in `report.txt` in the working directory; check it with `cat report.txt` on macOS or Linux, or `type report.txt` on Windows. The tool is complete: it reads input, parses it into typed records, computes derived values, renders a report, and writes it out, with every error path named in a type.

Reading the Manifest

Now for the reward that only a capability-typed language can offer. Ask the compiler what each function in the finished program is allowed to do:

```
$ capa --manifest main.capa | grep declared_capabilities
      "declared_capabilities": ["Fs"],
      "declared_capabilities": ["Fs", "Stdio"],
      "declared_capabilities": [],
      "declared_capabilities": []
```

Read the four lines against the four functions. `load_students` declares `Fs`; `main` declares `Fs` and `Stdio`; and the last two, `parse_line` and `render`, declare **nothing at all**. The compiler is stating, as a proven fact, that the parsing and reporting logic, the bulk of the program, cannot touch the

filesystem, the network, or any other resource. A reviewer reading this manifest learns, without reading a single line of the bodies, that the core computation is sandboxed from authority by construction, not by convention or by trust.

> **NOTE The shape of a Capa program**
>
> This is the pattern in miniature, and it scales. Push the authority to a thin outer layer (`main` and a handful of I/O helpers), keep the logic pure and provably so, and narrow every capability to the least it needs at the boundary. Bigger programs are just more of the same: more files, more types, more user-defined capabilities for domain operations, with the type system enforcing the discipline the whole way up.

The Whole Project, at a Glance

You have built a real, multi-file program. `parse.capa` defines the `Student` type and turns one line into an `Option<Student>`. `report.capa` renders a list of students into the report string. `main.capa` holds the capabilities, reads the input under a narrowed `Fs`, wires the pure functions together, writes the output, and reports any failure as a value. Tests cover the two pure pieces, and the manifest proves the separation of authority. Every idea from Part I appears somewhere in these three short files.

Try It Yourself

Take the finished tool further. Each task is a realistic enhancement.

> **Exercise 17-1 Filenames from Arguments**
>
> Use the `Env` capability's `args()` to let the user pass the input and output filenames on the command line (for example `capa --run main.capa`

`grades.txt out.txt`), falling back to the defaults when no arguments are given. Remember to add `Env` to `main`'s parameters.

Exercise 17-2 Count the Skips

Have `load_students` also report how many lines were skipped as malformed, and print that count after the summary, so the user knows if part of their file was ignored.

Exercise 17-3 Empty File

Run the tool against an empty `scores.txt`. Decide what *should* happen (perhaps a friendly "no students found" message) and make `main` produce it instead of an empty report.

Exercise 17-4 Tighter Narrowing

Instead of `restrict_to("./")`, create a `data/` folder, put `scores.txt` there, and narrow the `Fs` to `"./data/"`. Confirm the tool still works, and that an attempt to read a file outside `data/` is denied.

Exercise 17-5 End-to-End Test

Write a test that writes a small `scores.txt` through a narrowed `Fs`, runs `load_students` and `render` on it, and checks the report, an end-to-end test that exercises all three files together.

Summary

You finished your first real Capa program. `main.capa` is the entry point and the only file that holds capabilities: `load_students` reads the file with `Fs` and returns a `Result`, skipping malformed lines by matching the parser's `Option`, and `main` narrows the filesystem with `restrict_to`, wires the pure parser and reporter together, writes the report, and handles every failure through `Result` and `match`, with no exceptions and no crashes. Running it produced the report on screen and on disk, and the manifest proved that the parsing and reporting logic declares no capabilities at all, authority confined to a thin outer layer by construction. That shape, pure core, thin effectful edge, least authority at the boundary, is how Capa programs are built at any size. With one project complete, the next builds something that shows off what makes Capa unique: a tool that works with the supply-chain artefacts the compiler itself produces.

CHAPTER 18

Project 2: Reading an SBOM

The second project leans into what makes Capa distinctive. A **Software Bill of Materials**, or **SBOM**, is a list of every component that goes into a piece of software, the way an ingredients label lists what is in a packaged food. Capa's compiler can emit one automatically, recording which capabilities each part of a program holds. In this project we build a tool that *reads* such a document and **audits** it: for each component, it checks whether the capabilities it declares are allowed by a policy, and it fails loudly if any are not. That is exactly the kind of gate you would run in a continuous-integration pipeline to stop a risky dependency from slipping in.

Along the way you will learn to work with **JSON**, the text format SBOMs are written in, using the standard library's JSON support. This chapter reads the document into typed records; the next applies the policy; the last makes it a finished command-line tool.

The Brief

Our tool reads a file `sbom.json` describing the components of a project. Each component has a name, a version, and the list of capabilities it holds. To keep the focus on the language rather than a real-world schema, we use a small, readable shape:

```
{
  "components": [
    { "name": "capa_csv",  "version": "0.1", "capabilities": [] },
```

```
    { "name": "capa_http", "version": "0.2", "capabilities": ["Net"] },
    { "name": "logger",    "version": "1.0", "capabilities": ["Stdio", "Net"] }
  ]
}
```

The tool will read this, list each component and its capabilities, flag any component holding a capability that the policy does not allow, and exit with a failure if it finds even one violation. This chapter writes the part that turns that text into a `List<Component>` we can work with.

A Look at JSON in Capa

JSON is a simple, ubiquitous way to write structured data as text: objects in braces with `"key": value` pairs, arrays in brackets, plus strings, numbers, booleans, and null. Capa's standard library parses JSON with the function `parse_json`, which returns a `Result<JsonValue, String>`, an error string if the text is not valid JSON. A `JsonValue` is a sum type (Chapter 7) with one variant per JSON shape: `JNull`, `JBool`, `JNum`, `JStr`, `JArr`, and `JObj`.

You rarely match on those variants directly. Instead, `JsonValue` offers extractor methods that each return an `Option`: `as_string()`, `as_num()`, `as_bool()`, `as_array()`, and `as_object()`. Each gives `Some` if the value has that shape and `None` otherwise, which is exactly the safety you want when poking around data that came from a file.

```
fun main(stdio: Stdio)
    match parse_json("{ \"name\": \"capa_csv\" }")
        Ok(root) ->
            let obj = root.as_object().unwrap_or(new_map())
            let name = obj.get("name").and_then(
                fun (v: JsonValue) -> Option<String> =>
v.as_string())
            stdio.println(name.unwrap_or("?"))   // capa_csv
        Err(e) -> stdio.eprintln("bad json: ${e}")
```

That is a mouthful for one field, so for the real parser we will lean on the `?` operator (Chapter 8) to thread these `Option`s together cleanly.

The Data Model

As in Project 1, we begin by deciding what a parsed component *is*. A struct captures it:

```
// sbom.capa
pub type Component {
    name: String,
    version: String,
    capabilities: List<String>
}
```

Parsing One Component

A function takes one `JsonValue`, the JSON object for a single component, and produces a `Component`, or `None` if it is missing a field or has the wrong shape. Because every step might fail, `?` does the heavy lifting: any `None` along the way abandons the whole parse.

```
// sbom.capa
pub fun parse_component(j: JsonValue) -> Option<Component>
    let obj = j.as_object()?
    let name = obj.get("name")?.as_string()?
    let version = obj.get("version")?.as_string()?
    let caps_json = obj.get("capabilities")?.as_array()?
    var caps: List<String> = []
    for item in caps_json
        match item.as_string()
            Some(s) -> caps.push(s)
            None -> ()      // ignore non-string entries
    return Some(Component {
        name: name, version: version, capabilities: caps
    })
```

Trace one field: `obj.get("name")` looks up the key and returns `Option<JsonValue>`; the first `?` unwraps it (or bails with `None`); `.as_string()` returns `Option<String>`; the second `?` unwraps that. Two `?` operators turn "look up a key and expect a string" into one clean line. The capabilities are an array of strings, so we walk it, keeping each string and quietly ignoring anything that is not one.

Parsing the Whole Document

Now the top-level function: take the file's text, parse the JSON, dig out the `components` array, and build a `Component` from each entry. This one returns a `Result` because `parse_json` does, and because a caller deserves to know *why* a document was rejected. The handy method `ok_or` converts an `Option` into a `Result` by supplying an error for the `None` case, which lets us keep using `?`.

```
// sbom.capa
pub fun parse_sbom(text: String) -> Result<List<Component>, String>
    let root = parse_json(text)?
    let obj = root.as_object().ok_or("root is not an object")?
    let arr = obj.get("components").ok_or("missing 'components'")?
                 .as_array().ok_or("'components' is not an array")?
    var comps: List<Component> = []
    for item in arr
        match parse_component(item)
            Some(comp) -> comps.push(comp)
            None -> ()      // skip malformed entries
    return Ok(comps)
```

The first `?` propagates a JSON syntax error straight out as the `Result`'s error. After that, each `ok_or(...)?` says "this had better be an object / have a components field / be an array, and if not, fail with this message." The result is a function that is both safe, it cannot crash on bad input, and informative, every failure carries a reason.

Testing the Parser

Like all our parsing code, this is pure, so a test just feeds it a JSON string and checks the records that come back.

```
// tests/test_sbom.capa
import sbom

fun main(stdio: Stdio)
    let text = "{ \"components\": [
        { \"name\": \"capa_http\", \"version\": \"0.2\",
          \"capabilities\": [\"Net\"] } ] }"
    match parse_sbom(text)
        Ok(comps) ->
            if comps.length() != 1
                panic("expected one component")
            let c0 = comps.get(0).unwrap_or(
                Component { name: "", version: "", capabilities: []
})
            if c0.name != "capa_http"
                panic("wrong name")
            if not c0.capabilities.contains("Net")
                panic("missing Net capability")
            stdio.println("sbom parse test passed")
        Err(e) -> panic("should parse: ${e}")
```

The test confirms one component comes back with the right name and the `Net` capability in its list. With reading handled, we have a clean `List<Component>` to work with, and the interesting question can be asked: does any component hold a capability it should not?

Try It Yourself

Extend the reader. JSON from the wild is messy, so robustness is the theme.

> **Exercise 18-1 Count Components**

Write a small program that parses the three-component example from the brief and prints how many components it found and the name of each, one per line.

Exercise 18-2 Total Capabilities

Add a function that takes a `List<Component>` and returns the total number of capability entries across all components (use `fold`, or a loop with a counter). Test it against the brief's data (it should be `3`).

Exercise 18-3 Malformed Entry

Feed `parse_sbom` a document where one component is missing its `version` field. Confirm that the bad entry is skipped (it parses to `None`) while the good ones still come through.

Exercise 18-4 Bad JSON

Pass `parse_sbom` a string that is not valid JSON at all, and confirm it returns an `Err` with a message rather than crashing. Print the message.

Summary

You began an SBOM-auditing tool by teaching it to read. After meeting JSON and the standard library's `parse_json` and `JsonValue`, with its `Option`-returning extractors `as_object`, `as_array`, and `as_string`, you modelled a `Component` struct and wrote two parsers: `parse_component`, which threads several `Option`s together with `?` to build one record, and `parse_sbom`, which uses `ok_or` to turn those `Option`s into a `Result` carrying a clear error for every way the document can be malformed. Both are pure, so testing them

was a matter of feeding in a JSON string and checking the records. With a tidy `List<Component>` in hand, the next chapter defines a policy and checks each component against it.

CHAPTER 19

Checking a Policy

You can now read an SBOM into a `List<Component>`. The point of the tool, though, is to *judge* that list: to decide whether each component is allowed to hold the capabilities it declares. That judgement is a **policy**, and applying it is the subject of this chapter. As with the parser, all of this logic is pure, it takes data and returns data, so it stays easy to test and free of any authority.

What a Policy Is

Our policy is the simplest useful kind: an **allowlist** of capabilities. A component is acceptable if every capability it holds appears on the list; it is in **violation** for each capability it holds that does not. If the allowlist is `Stdio` and `Fs`, then a logging component holding `Stdio` is fine, but one that also reaches for `Net` is flagged, exactly the "why does the logger need the network?" question a reviewer should be asking.

A `Set<String>` (Chapter 4) is the natural home for an allowlist: order does not matter, duplicates are meaningless, and the one question we ask, *is this capability allowed?*, is precisely what a set answers quickly. A small helper builds one from a list of names.

```
// policy.capa
import sbom

pub fun make_allowed(names: List<String>) -> Set<String>
    let allowed: Set<String> = new_set()
    for n in names
        allowed.add(n)
    return allowed
```

Modelling a Violation

When the tool finds a problem, it should say precisely what it is: which component, and which forbidden capability. A small struct captures one such finding.

```
// policy.capa
pub type Finding {
    component: String,
    capability_name: String
}
```

Checking One Component

Now the core check. Given a component and the allowlist, produce a `Finding` for every capability the component holds that is not allowed. A component with no problems yields an empty list.

```
// policy.capa
pub fun check_component(c: Component, allowed: Set<String>) ->
List<Finding>
    var findings: List<Finding> = []
    for cap in c.capabilities
        if not allowed.contains(cap)
            findings.push(Finding { component: c.name,
capability_name: cap })
    return findings
```

It walks the component's capabilities and, for each one the set does not contain, pushes a `Finding` naming the component and the offending capability. Returning a *list* of findings, rather than a single yes/no, means one component can report several violations at once, and a clean component simply returns `[]`.

Auditing the Whole List

Auditing the full SBOM is then just running that check over every component and gathering all the findings into one list. A nested loop does it: the outer loop over components, the inner loop over the findings each one produces.

```
// policy.capa
pub fun audit(components: List<Component>, allowed: Set<String>) ->
List<Finding>
    var all: List<Finding> = []
    for c in components
        for f in check_component(c, allowed)
            all.push(f)
    return all
```

The result is a single `List<Finding>`: empty when the whole SBOM satisfies the policy, and otherwise one entry per violation. That list is everything the tool needs to report its verdict.

Rendering the Verdict

Finally, turn the findings into human-readable text, the same pattern as the report in Project 1. An empty list is good news; a non-empty one lists each problem and ends with a count.

```
// policy.capa
pub fun render_findings(findings: List<Finding>) -> String
    if findings.is_empty()
        return "OK: all components satisfy the policy\n"
    var out = "POLICY VIOLATIONS:\n"
    for f in findings
        out = out + "  ${f.component} holds disallowed
${f.capability_name}\n"
    out = out + "\n${findings.length()} violation(s) found\n"
    return out
```

Testing the Policy

All of this is pure, so a test builds a few components, runs the audit against an allowlist, and checks the findings. Here we allow only `Stdio` and `Fs`, then audit the three components from the brief.

```
// tests/test_policy.capa
import sbom
import policy

fun main(stdio: Stdio)
    let comps = [
        Component { name: "capa_csv", version: "0.1", capabilities:
[] },
        Component { name: "capa_http", version: "0.2", capabilities:
["Net"] },
        Component { name: "logger", version: "1.0",
                    capabilities: ["Stdio", "Net"] }
    ]
    let allowed = make_allowed(["Stdio", "Fs"])
    let findings = audit(comps, allowed)
    if findings.length() != 2
        panic("expected 2 violations, got ${findings.length()}")
    stdio.println("policy test passed")
```

```
$ capa test
running 3 tests
  test_sbom   ... ok
  test_policy ... ok
  test_report ... ok

3 passed, 0 failed
```

Two violations is exactly right: `capa_http` and `logger` each hold `Net`, which is not on the allowlist, while `capa_csv`, holding nothing, is clean. The judging logic works; all that remains is to wire it to a real file and a real exit code.

Try It Yourself

Make the policy richer. Each task is a focused change to `policy.capa` with a test.

Exercise 19-1 Count Clean Components

Add a function that returns how many components had *no* violations, and include that number in the rendered verdict (for example `2 of 3 components clean`).

Exercise 19-2 A Severity

Add a `severity: String` field to `Finding`. Make holding `Unsafe` or `Net` a `"high"` severity and anything else `"low"`, and show the severity on each line of the report.

Exercise 19-3 Per-Component Exceptions

Sometimes one component is legitimately allowed an extra capability. Extend `audit` to accept a second allowlist of `(component, capability)` pairs that are exempt, and skip findings that match an exemption.

Exercise 19-4 Denylist Instead

Write an alternative `audit_deny` that takes a *denylist*, the capabilities that are forbidden, and flags any component holding one of them. Discuss which model, allowlist or denylist, is safer by default (hint: think about a capability nobody thought to list).

Summary

You wrote the judging half of the SBOM auditor, all of it pure. A policy is an allowlist held in a `Set<String>`, built by `make_allowed`; a `Finding` struct

records one violation as a component-plus-capability pair. `check_component` produces a finding for each capability a component holds that the set does not allow, `audit` gathers those across every component with a nested loop, and `render_findings` turns the result into a verdict, a clean message when the list is empty, an itemised report with a count when it is not. A test against the brief's data confirmed exactly two violations. The logic is complete and verified; in the final chapter of this project we give it a `main` that reads a real file, prints the verdict, and signals success or failure the way a CI pipeline expects.

CHAPTER 20

A CI-Ready Tool

The reading and the judging are done, both pure, both tested. The last piece is the `main.capa` that makes it a real command-line tool: it reads `sbom.json` from disk, runs the audit, prints the verdict, and, crucially for a tool meant to run in a pipeline, **signals failure with its exit code** when the policy is violated. This chapter writes that entry point and finishes the project.

Exit Codes, the CI Handshake

Automated pipelines do not read your program's prose; they read its **exit code**, the single number a program returns when it ends. By universal convention, zero means success and any non-zero value means failure. A continuous-integration system runs your tool and, if it exits non-zero, marks the build as failed and stops the risky change from merging.

Capa makes this easy with something you already know. A program that runs to the end exits zero. And `panic(message)`, from Chapter 14, stops the program and exits non-zero. So the recipe is simple: print the verdict, and if there were any violations, `panic` to fail the build.

Wiring It Together

First a small helper that does the whole pipeline, read, parse, audit, and returns either the findings or an error message. There is one wrinkle worth meeting: `fs.read` fails with an `IoError`, but `parse_sbom` fails with a `String`. To use `?` for both in one function, their error types must agree, so we convert the `IoError` to a `String` with `map_err` (Chapter 8) before propagating it.

```
// main.capa
```

```
import sbom
import policy

fun run(fs: Fs, path: String, allowed: Set<String>) ->
Result<List<Finding>, String>
    let text = fs.read(path).map_err(
        fun (e: IoError) -> String => "read failed: ${e}")?
    let comps = parse_sbom(text)?
    return Ok(audit(comps, allowed))
```

Read it as a tidy three-step pipeline: read the file (turning any I/O error into a message), parse the SBOM (whose error is already a message), and audit the result. Each `?` bails out with a clear `String` reason if its step fails; otherwise `run` returns the list of findings.

The Entry Point

Now `main`. It narrows the filesystem, defines the policy, runs the pipeline, prints the verdict, and ends, either cleanly, or with a `panic` that fails the build when violations were found.

```
// main.capa
fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let allowed = make_allowed(["Stdio", "Fs"])
    match run(work_dir, "sbom.json", allowed)
        Err(e) ->
            stdio.eprintln("error: ${e}")
            panic("audit could not run")
        Ok(findings) ->
            stdio.print(render_findings(findings))
            if not findings.is_empty()
                panic("${findings.length()} policy violation(s)")
```

As in Project 1, `main` holds only `Fs` and `Stdio`, and narrows the filesystem to the working directory before touching it. If the pipeline fails outright, a missing file, malformed JSON, it prints the reason and panics. If it succeeds, it prints the verdict; and when that verdict contains violations, the final

`panic` is what turns a printed report into a failed build. A clean SBOM, by contrast, prints its `OK` line and lets `main` end normally, exit code zero.

Running It Both Ways

Put the three-component example from the brief in `sbom.json` and run the tool. Two components hold `Net`, which the policy forbids, so the audit reports them and the program fails:

```
$ capa --run main.capa
POLICY VIOLATIONS:
  capa_http holds disallowed Net
  logger holds disallowed Net

2 violation(s) found
panic: 2 policy violation(s)
```

The `panic:` line on standard error is Capa's way of exiting non-zero; a CI system seeing that status fails the build. Now edit `sbom.json` to remove the `Net` capabilities (or add `Net` to the allowlist) and run again:

```
$ capa --run main.capa
OK: all components satisfy the policy
```

This time the program prints its `OK` line and ends cleanly, exit code zero, the build passes. The same tool, two outcomes, each communicated to the pipeline through the exit code.

The Manifest, One More Time

As always, the manifest tells the honest story of what this tool can do. Only `run` and `main` carry authority, `Fs` to read the file and `Stdio` to print, while every function that parses JSON, checks the policy, and renders the verdict declares no capabilities at all. An auditor can see at a glance that the tool reads one file and writes to the terminal, and that its entire decision-making

core is sandboxed from the outside world. A program that audits other software's authority is itself a model of least authority.

What You Built

Project 2 is a complete, useful program: a supply-chain gate. `sbom.capa` reads a JSON document into typed `Component` records using `parse_json` and the `?` operator; `policy.capa` judges those components against an allowlist held in a `Set`, producing a list of `Finding`s; and `main.capa` connects them to a real file under a narrowed `Fs`, prints the verdict, and fails the build through its exit code when the policy is broken. It is exactly the kind of tool teams run to keep risky dependencies out, built from the same small ideas as everything else in this book.

Try It Yourself

Turn the tool into something you would actually deploy.

> **Exercise 20-1 Policy from a File**
>
> Instead of hard-coding the allowlist, read it from a `policy.txt` file (one capability per line) using the narrowed `Fs`, and build the `Set` from those lines. Now the policy can change without recompiling.

> **Exercise 20-2 Filenames from Arguments**
>
> Use `Env`'s `args()` to accept the SBOM path on the command line, defaulting to `sbom.json`. Add `Env` to `main` and narrow it with `restrict_to_keys([])` if you use no environment variables.

> **Exercise 20-3 A Summary Line**

Before the verdict, print a one-line summary such as `Audited 3 components against 2 allowed capabilities.` so a reader sees the scope at a glance.

Exercise 20-4 Warn-Only Mode

Add a mode that prints the violations but does *not* panic, so the tool can be run as an advisory report rather than a hard gate. Decide how the caller selects it (an argument, perhaps), and explain when each mode is appropriate.

Summary

You finished the SBOM auditor by giving it a CI-ready entry point. Pipelines judge a tool by its exit code, zero for pass, non-zero for fail, and Capa expresses that with a clean finish for success and `panic` for failure. A `run` helper chains the read, parse, and audit steps with `?`, using `map_err` to reconcile the `IoError` from `fs.read` with the `String` errors from parsing, and `main` narrows the filesystem, applies the policy, prints the verdict, and panics when violations are found so the build fails. The manifest confirms that only the thin outer layer holds authority. Two complete projects now sit behind you, one everyday, one distinctively Capa. The final project leans entirely into the language's second security axis: a small service that handles secrets and proves, at compile time, that they never leak.

CHAPTER 21

Project 3: Handling Secrets

The final project leans entirely on Capa's second security axis, information-flow control (Chapter 13). We build a small **payment service**: it receives card numbers, charges them, and keeps an audit record, all while the compiler **proves** that a card number can never be printed, logged, or stored in the clear. This is the kind of guarantee that standards like PCI DSS demand and that, in most languages, rests entirely on careful reviewers. In Capa it rests on the type system. This chapter sets up the secret data and meets the compiler errors that keep it safe; the next two add a vault capability and prove the whole thing.

The Brief

The service handles a payment: a card number and an amount. The rules are strict and non-negotiable. The card number is sensitive and must never leave the program in full, not to the screen, not to a file, not anywhere. The only form of the card that may be shown or stored is a **masked** one, revealing at most the last four digits. And every place where we deliberately reveal even that much must be recorded, so an auditor can see exactly where disclosure happens. Capa lets us state all three rules in the types and have them enforced at compile time.

Marking the Secret

We model a payment as a struct, and we mark the card field `@secret` right in its type (Chapter 13). A field declared secret keeps its label wherever it is read, so the card cannot be laundered through a public sibling like the amount.

```
// payment.capa
pub type Payment {
    card: @secret String,
    amount: Int
}
```

From now on, any value that comes from `p.card`, or anything derived from it, carries the secret label automatically, and the compiler tracks it for us. The `amount` stays public; only the card is sensitive.

The Leak the Compiler Catches

Suppose we naively try to log the payment as we process it:

```
// payment.capa
pub fun process(stdio: Stdio, p: Payment)
    stdio.println("charging card ${p.card} for ${p.amount}")
```

```
$ capa --run main.capa
payment.capa:2:5: information-flow violation: a @secret value reaches
                  Stdio.println, a public sink
```

The interpolated message contains `p.card`, so by join the whole string is secret (Chapter 13), and `println` is a public sink. The compiler warns (and rejects under @strict_ifc()), exactly the leak we wanted to make impossible, caught before the program ever runs. Note that this function legitimately holds `Stdio`; capabilities alone would happily let it print. It is information-

flow control, not the capability check, that stops *this* particular value from going out.

Masking the Card

We do need to show *something*, a receipt line, a log entry, so we write a function that masks the card down to its last four digits. It takes the secret card and builds a masked string from it.

```
// payment.capa
fun mask_card(card: @secret String) -> String
    let n = card.length()
    let last4 = card.substring(n - 4, n)
    return "****-****-****-${last4}"
```

Here is the subtlety that makes Capa's guarantee airtight. Even though `mask_card` returns a `String`, that string was **built from a secret**, the `substring` of a secret is secret, and interpolating it keeps it secret, so the value the function returns still carries the secret label. Try to print the masked value directly and the compiler stops you again:

```
let masked = mask_card(p.card)
stdio.println(masked)            // still a violation: masked is
@secret
```

This is the right behaviour, not an annoyance. Masking *intends* to disclose part of the secret, and Capa will not let that disclosure happen silently, no matter how harmless it looks. Something must declare, on the record, that this is a deliberate release.

Declassifying, on the Record

That declaration is `declassify` (Chapter 13). It converts a `@secret` value into a public one and demands a written `reason`. Wrapping the masked card in `declassify` is what finally lets it be printed.

```
// payment.capa
pub fun receipt_line(p: Payment) -> String
    let safe = declassify(mask_card(p.card),
                          reason: "PCI: receipt shows last 4 digits
only")
    return "Charged ${safe} for ${p.amount}"
```

Now `safe` is a public `String`, so it can flow into a receipt, a log, or the screen freely. The `reason` is not a comment; the compiler records every `declassify` site in the SBOM under the function's `declassifications` list, complete with its reason and source position (pos) (Chapter 13). The result is a precise, machine-checkable list of every point at which the program deliberately reveals part of a card, the exact thing an auditor needs.

> **NOTE Make it a hard guarantee with @strict_ifc**
>
> By default an information-flow violation is a warning, useful while adopting IFC, but for a payment service you want it to be impossible to ship a leak. Marking the service's functions with the `@strict_ifc()` attribute (Chapter 13) turns every violation into a compile error, and additionally catches *implicit* leaks, where a secret influences which branch runs. For this project, treat `@strict_ifc` as mandatory.

Where We Are

We have the heart of the service already: a `Payment` whose card is `@secret`, a `mask_card` that reduces it to its last four digits while keeping the label, and a `receipt_line` that, through a single audited `declassify`, produces a public string safe to show. The compiler has rejected two naive leaks for us. What remains is to store these receipts somewhere, and to do it through a capability of our own, so the audit trail is as disciplined as the data. That is the next chapter.

Try It Yourself

These exercises are about *making the compiler complain*, then satisfying it honestly. Add `@strict_ifc()` so violations are hard errors.

Exercise 21-1 Provoke the Leak

Write a function that takes a `Payment` and a `Stdio` and tries to print the full card number. Confirm the information-flow violation, then change it to print the masked, declassified form instead and watch it compile.

Exercise 21-2 A Secret Total

Add a `@secret String cvv` field to `Payment`. Try to build a debug string that interpolates both `card` and `cvv` and confirm it is secret. Then write a function that returns only the amount, a public `Int`, and confirm *that* is fine, the non-secret data flows freely.

Exercise 21-3 Honest Reasons

Write two `declassify` calls with genuinely different reasons (for example one for a printed receipt, one for an internal log) and note how each would appear as a separate entry in the disclosure record. Reflect on why forcing a reason is more than bureaucracy.

Exercise 21-4 No Laundering

Try to hide the card by putting it in a one-element `List<String>` and pulling it back out, then printing it. Confirm the label survives the round trip, the secret cannot be washed clean by a container (Chapter 13).

Summary

You laid the foundation of a payment service that cannot leak. By marking the card field `@secret` in the `Payment` type, you made the compiler track it through every derived value. It caught the obvious leak, printing the card, and the subtle one, printing a masked card that still carried the secret label, because masking is itself a disclosure. The single sanctioned release is `declassify`, which turns the masked value public, demands a written reason, and records that reason in the SBOM as an auditable disclosure point; with `@strict_ifc()` every violation becomes a hard error. The logic is in place and provably safe. Next we build a capability of our own, an audit vault, so that storing these receipts is as disciplined as computing them.

CHAPTER 22

A Vault Capability

The payment service can now turn a secret card into a safe receipt line through an audited `declassify`. Those receipts need to go somewhere, an audit ledger on disk, and we could write them with a raw `Fs`. But this project is about discipline, so we will do better: we define a **capability of our own** (Chapter 11), an `AuditVault`, whose only power is to append an entry to the ledger. This buys us two things at once: callers reason about "can write to the audit log" instead of "can touch the filesystem", and, as you will see, the capability's contract structurally guarantees that only safe, declassified data can ever be stored.

Declaring the Vault

The capability declares a single method, `record`, that appends one entry and may fail with an `IoError`. Crucially, the entry is a plain, **public** `String`, that detail is the whole security argument, and we will return to it.

```
// vault.capa
import payment

pub capability AuditVault
    fun record(self, entry: String) -> Result<Unit, IoError>
```

Implementing It over the Filesystem

A concrete implementor is a struct holding the ledger path and an `Fs`, paired with an `impl` (Chapter 11). Because `FileVault` implements a user-defined capability, the cap-bearing relaxation lets it hold the built-in `Fs` as a field.

The `record` method appends by reading what is there, adding the new line, and writing it back.

```
// vault.capa
type FileVault {
    path: String,
    fs: Fs
}

impl AuditVault for FileVault
    fun record(self, entry: String) -> Result<Unit, IoError>
        let existing = self.fs.read(self.path).unwrap_or("")
        return self.fs.write(self.path, existing + entry + "\n")
```

Inside `record`, `self.fs` is the filesystem capability the vault was built with. We read the current ledger (defaulting to an empty string if it does not exist yet, via `unwrap_or`), append the entry and a newline, and write the whole thing back. All the filesystem authority is sealed inside the vault; nobody holding an `AuditVault` can do anything with it except `record`.

The Factory

A factory function creates a vault from the built-in `Fs` it needs. Recall that a user-defined capability, unlike a built-in, may be returned from a function (Chapter 11), which is what makes this possible.

```
// vault.capa
pub fun make_vault(fs: Fs, path: String) -> FileVault
    return FileVault { path: path, fs: fs }
```

Charging a Payment

Now the operation that ties the two halves together: take a payment, produce its safe receipt line, and record it in the vault. Notice the signature, it asks for an `AuditVault`, not an `Fs`.

```
// vault.capa
pub fun charge(vault: AuditVault, p: Payment) -> Result<Unit,
IoError>
    let line = receipt_line(p)   // a public String (declassified in
Ch. 21)
    return vault.record(line)
```

`receipt_line` (from Chapter 21) returns a public `String`, because the card inside it was masked and deliberately declassified with a reason. That public string flows into `vault.record` without complaint. The card itself never comes near the vault.

Why the Contract Is the Guarantee

Here is the elegant part. Suppose a careless change tried to record the raw card instead of the receipt line:

```
pub fun charge(vault: AuditVault, p: Payment) -> Result<Unit,
IoError>
    return vault.record("card: ${p.card}")   // warning (hard error
under @strict_ifc())
```

```
$ capa --run main.capa
vault.capa: information-flow violation: a @secret value is passed
where a
            public String is required (AuditVault.record expects a
public entry)
```

The string `"card: ${p.card}"` is secret by join, but `record` declares its parameter as a public `String`. A secret cannot cross a function boundary into a public parameter (Chapter 13), so the compiler warns (and, under @strict_ifc(), rejects it). In other words, the vault's contract, *I accept public strings only*, makes it **structurally impossible** to store a secret. The only way to get data into the ledger is to declassify it first, in plain sight, with a reason. The discipline is enforced not by a careful programmer, but by the type of `record`.

NOTE Two disciplines, working together

This project shows both halves of Capa meeting in one design. The **capability** side means a function that holds an `AuditVault` can write to the ledger and do nothing else; the filesystem authority is hidden inside the vault. The **information-flow** side means only declassified data can be passed to it. Together they guarantee that the audit log receives exactly what it should, masked, deliberately-released receipt lines, and nothing it should not.

The Manifest's View

As ever, the manifest tells the honest story. `charge` declares `AuditVault`, not `Fs`. A reviewer reading it learns that the function writes to the audit log, the thing they care about, without being told the transport beneath happens to be a file. The low-level `Fs` appears only inside `make_vault` and the vault's implementation, where it is sealed away. The authority surface of the program is described in the language of the domain: payments, receipts, an audit vault.

Try It Yourself

Extend the vault, and keep proving the guarantee holds.

Exercise 22-1 Try to Cheat

Write a version of `charge` that attempts to record the raw `p.card`, and confirm the compiler rejects it. Then fix it to record the receipt line. This is the guarantee in action.

Exercise 22-2 Timestamped Entries

Give `record` a richer entry by prepending a counter or label in `charge` (for example `Entry #1: ...`). Keep everything public, the prefix is not secret, so it composes fine with the declassified receipt.

Exercise 22-3 A Second Implementor

Write a `MemoryVault` that implements `AuditVault` but stores entries in an in-memory `List<String>` instead of a file (useful for tests). Note that `charge` does not change at all, it depends on the `AuditVault` contract, not the implementation.

Exercise 22-4 Narrow the Vault's Fs

In a factory or in `main`, narrow the `Fs` with `restrict_to` to the ledger's directory before building the `FileVault`, so even the vault's own filesystem reach is the smallest it can be.

Summary

You gave the payment service an audit vault of its own. The `AuditVault` capability declares a single `record` method that takes a public `String`; `FileVault` implements it over a hidden `Fs` using the cap-bearing pattern, and a factory builds one from the filesystem capability it needs. The `charge` function records a payment's declassified receipt line through the vault, and its signature advertises `AuditVault`, not `Fs`. The deep point is that `record`'s public-`String` parameter makes storing a secret structurally impossible: a `@secret` value cannot cross into a public parameter, so the only way into the ledger is through a declassify. Capabilities hide the filesystem; information flow forbids the leak; together they make the design safe by construction. In

the last chapter we assemble the whole service in `main`, run it, and read the proof straight out of the SBOM.

CHAPTER 23

The Service, Proven Safe

Everything is in place. `payment.capa` marks the card `@secret` and turns it into a declassified receipt; `vault.capa` defines an `AuditVault` that can store only public strings. This final chapter assembles them in `main.capa`, runs the service, and, best of all, reads the **proof** straight out of the SBOM: a machine-checkable statement that the card leaks nowhere except the single, reasoned point we chose. It also closes Part II.

The Service Operation

A small function performs one payment end to end: charge it, which records the declassified receipt in the vault, then print the same receipt to the screen. We mark it `@strict_ifc()` (Chapter 13) so that any information-flow violation, now or in a future edit, is a hard compile error rather than a warning.

```
// main.capa
import payment
import vault

@strict_ifc()
fun run(stdio: Stdio, v: AuditVault, p: Payment) -> Result<Unit,
IoError>
    charge(v, p)?
    stdio.println(receipt_line(p))
    return Ok(())
```

Both lines handle only the *declassified* receipt: `charge` records it, and `println` prints it. The raw card never appears here, and under `@strict_ifc` the compiler guarantees it cannot, not through this function and not through any branch it guards.

Wiring Up main

`main` holds the two real capabilities, `Fs` and `Stdio`, narrows the filesystem to the ledger directory, builds the vault and a sample payment, and runs the operation.

```
// main.capa
fun main(fs: Fs, stdio: Stdio)
    let ledger_dir = fs.restrict_to("./ledger/")
    let v = make_vault(ledger_dir, "./ledger/audit.log")
    let p = Payment { card: "4111111111111234", amount: 4200 }
    match run(stdio, v, p)
        Ok(_) -> stdio.println("done")
        Err(e) -> stdio.eprintln("charge failed: ${e}")
```

The filesystem is narrowed to `./ledger/` before the vault is built, so even the vault's own reach is the smallest it can be (Chapter 10). The card is supplied as an ordinary string literal; because the `card` field is declared `@secret`, that value takes on the secret label the moment it enters the struct, and the compiler tracks it from there.

Running It

Create a `ledger/` folder, then run the service:

```
$ capa --run main.capa
Charged ****-****-****-1234 for 4200
done
```

The screen shows only the masked card. The file `ledger/audit.log` now holds the same masked receipt line, never the full number. The service charged a payment, produced a receipt, and recorded an audit entry, all without the card escaping in the clear, and the compiler proved that before the program ever ran.

Reading the Proof

Here is the moment the whole project has been building toward. Ask the compiler for the SBOM, and it lists every point where the program deliberately discloses a secret, recorded under each function's `declassifications (counted in summary.declassification_sites)`:

```
$ capa --manifest main.capa
...
"functions": {
  "receipt_line": {
    "declassifications": [
      { "reason": "PCI: receipt shows last 4 digits only", "value":
"receipt", "pos": "payment.capa:14:5" }
    ]
  }
},
"summary": { "declassification_sites": 1 }
```

One entry. The compiler is telling you, as a proven fact, that this program reveals part of a card in exactly **one** place, for a stated reason, and nowhere else. Combine that with the capability manifest, where the payment logic declares no authority and only the vault touches `Fs`, and you have a complete, machine-readable security argument: the card cannot reach a sink except through this one audited disclosure. No reviewer had to spot it; no test had to catch it. It is true by construction, and the proof ships with the build.

> **NOTE What a real reviewer gets**
>
> In a normal language, "the card never leaks" is a claim defended in code review and re-litigated on every change. In Capa it is a property the compiler checks and the SBOM records, alongside the exact, reasoned exceptions. That is the difference between hoping a system is safe and being able to *show* that it is.

Part II in Retrospect

Three projects now sit behind you. The grade-book tool showed the everyday shape of a Capa program: a pure core, a thin effectful edge, capabilities narrowed at the boundary. The SBOM auditor put the language to work on supply-chain safety, reading and judging the very authority data the compiler emits. And this payment service brought both halves of the type system together, capabilities controlling *what* a function may do, information flow controlling *where* its data may go, to make a sensitive system safe by construction. Across all three, the same handful of ideas from Part I did the work; the discipline simply scaled with the program.

Try It Yourself

Round out the service, and keep the proof intact.

Exercise 23-1 Many Payments

Process a `List<Payment>` in a loop, charging each and collecting the receipts. Confirm the ledger ends with one line per payment and the screen never shows a full card.

Exercise 23-2 Keep It Strict

With `@strict_ifc()` on `run`, deliberately add a line that prints `p.card` and confirm the build fails. Remove it and confirm the build passes, your guarantee, demonstrated.

Exercise 23-3 Read Back the Ledger

Add a function that reads `ledger/audit.log` through the narrowed `Fs` and prints how many entries it contains, a simple end-of-day report. Note that the ledger only ever held masked data, so printing it is safe.

Exercise 23-4 Inspect the Sites

Add a second, differently-reasoned `declassify` somewhere (for example a separate internal log line), run `capa --manifest`, and confirm both disclosure sites appear with their distinct reasons.

Summary

You assembled a payment service that is safe by construction and proved it. The `run` operation, marked `@strict_ifc()`, charges a payment and prints its receipt, handling only the declassified form; `main` narrows the filesystem to the ledger directory, builds the vault, and runs the operation. On screen and on disk, only the masked card ever appears. The decisive payoff came from the SBOM: its `declassification_sites` list named the single, reasoned point of disclosure, and the capability manifest confirmed the payment logic holds no authority at all, together, a machine-checkable proof that the card cannot leak. That closes Part II and the body of the book: you have learned the core of the language and built three real programs with it, from an everyday CLI tool to a system whose security the compiler itself vouches for. What remains are the appendices, quick references for installation, editors, and the journey onward from here.

APPENDICES

Appendices

Reference material and where to go from here.

APPENDIX A

Installation and Troubleshooting

Chapter 1 walked through installing Capa as part of writing your first program. This appendix gathers the install options in one place for reference, along with fixes for the problems that most often come up. Capa is a Python 3.10+ tool that transpiles `.capa` source to Python and runs it; you do not need to know any Python to use it.

Option A: One-Line Installer (Recommended)

The installer downloads the latest pre-built binary and places it on your PATH. Open a new terminal afterwards so the change takes effect.

On **Linux** or **macOS**:

```
$ curl -fsSL https://github.com/nelsonduarte/capa-
language/releases/latest/download/install.sh | bash
```

On **Windows** (PowerShell):

```
PS> irm https://github.com/nelsonduarte/capa-
language/releases/latest/download/install.ps1 | iex
```

If PowerShell refuses to run the script and shows an execution-policy error, it is blocking downloaded scripts for safety. Allow signed scripts once with the command below, answer Y to confirm, then run the install command again. Adding -Scope CurrentUser limits the change to your own account, so it needs no administrator rights.

```
PS> Set-ExecutionPolicy RemoteSigned
```

Then confirm it worked:

```
$ capa --version
capa 1.12.0
```

The installer needs no administrator rights and makes no system-wide changes; it drops `capa` into a per-user folder (`~/.local/bin` on Linux/macOS, `%LOCALAPPDATA%\capa` on Windows). To uninstall, delete that file. Rerunning the installer upgrades to the latest release.

Option B: Manual Binary Download

Each release publishes a standalone binary that bundles the compiler and a Python interpreter, no separate Python needed. Download the file for your platform, make it executable, and run it. On Linux:

```
$ curl -L -o capa \
   https://github.com/nelsonduarte/capa-
language/releases/latest/download/capa-linux-x86_64
$ chmod +x capa
$ ./capa --version
```

Each binary ships with a matching `.sha256` file so you can verify the download before running it (`sha256sum -c capa-linux-x86_64.sha256`). On macOS you must also clear the quarantine attribute the browser sets, with `xattr -d com.apple.quarantine capa`, because the binary is not yet Apple-notarised. Move the file into `~/.local/bin` to run `capa` from any directory.

Option C: From Source

If you want the test suite locally or intend to contribute, install from source. You need Python 3.10 or newer and git.

```
$ git clone https://github.com/nelsonduarte/capa-language
$ cd capa-language
$ pip install -e .
```

The editable install registers the `capa` package *and* puts a `capa` command on your PATH, so both `capa <args>` and `python -m capa <args>` work from anywhere. Verify with the test suite (a couple of minutes):

```
$ python -m unittest discover tests
```

To enable the experimental WebAssembly backend, install the extra: `pip install -e '.[wasm]'`, which unlocks `capa --wasm --run` and the Component Model output.

Making capa Available Everywhere: the PATH

The terminal finds programs by searching a list of folders called the **PATH**. If `capa --version` reports *command not found* after installing, the install folder is probably not on your PATH yet. On Windows the installer adds capa to your PATH for you, so this manual step is only needed on Linux and macOS, or after a manual binary download. There the installer prints the exact line to add; the usual fix is to append the folder to your shell's startup file and reload it:

```
# bash
$ echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
$ source ~/.bashrc

# zsh (macOS default)
$ echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc
```

```
$ source ~/.zshrc
```

On Ubuntu the default shell is bash, so the first pair of lines is the one to use: the first appends the folder ~/.local/bin to your PATH for good, by writing the line into ~/.bashrc (the file bash reads every time it starts), and the second reloads that file so the change applies in the terminal you already have open. If your shell is zsh instead, use the ~/.zshrc lines.

Always open a fresh terminal after changing the PATH; the change only affects terminals started afterwards.

Two small commands let you see what is going on. command -v capa (or which capa) prints the full path to the capa program if the terminal can find it, and prints nothing at all if it cannot. echo $PATH prints the folders currently being searched, separated by colons; after the fix above, ~/.local/bin should appear somewhere in that list.

```
$ command -v capa
$ which capa
$ echo $PATH
```

If you installed by downloading the binary by hand, the file only lives where you saved it, so the terminal will not find it from other folders. Move it into a folder that is on the PATH. The simplest choice, which needs no administrator rights, is your personal ~/.local/bin; make the file executable first if you have not already:

```
$ chmod +x capa
$ mkdir -p ~/.local/bin
$ mv capa ~/.local/bin/

# system-wide alternative, for every user (needs sudo)
$ sudo mv capa /usr/local/bin/
```

The difference is reach and permissions: ~/.local/bin belongs to you alone and needs no sudo, while /usr/local/bin is already on the PATH for everyone on the machine but writing there requires administrator rights.

Common Problems

"capa: command not found" / "not recognized"

On Linux and macOS the install folder is not on your PATH yet, so apply the fix above. On Windows the installer adds capa to the PATH for you, so it is usually enough to open a new terminal, since PowerShell reads the updated PATH only in sessions started after the install. As a fallback, a source install lets you run `python -m capa <args>` instead.

"No such file or directory" when running a file

You are in a different folder from the one holding your `.capa` file. Use `cd` to move into the right folder, and list its contents (`ls` on macOS/Linux, `dir` on Windows) to confirm the file is there.

macOS refuses to run the binary

Gatekeeper blocks unsigned downloads. Clear the quarantine attribute with `xattr -d com.apple.quarantine capa`, or allow it under System Settings -> Privacy & Security. (The one-line installer does this for you.)

The first run feels slow

A single-file binary self-extracts on first run, adding a fraction of a second. Later runs are cached and fast. A source install does not have this step.

> **NOTE When in doubt, ask the checker**
>
> For anything about your *code* rather than the install, run `capa --check yourfile.capa`. It runs the full analysis without executing the program and prints precise, source-aligned errors, often the fastest way to find what is wrong.

APPENDIX B

Editors and the Language Server

Any text editor can write Capa, since code is just text. But a little setup gives you colour highlighting and, better still, live error checking as you type. This appendix covers the editor extension, the language server that powers richer features, and a quick reference to the `capa` command-line flags.

Syntax Highlighting in VS Code

The VS Code Extension

The simplest setup for VS Code users is the official extension, published on the Marketplace as `Capa Language` (id `nelsonduarte.capa-language`). It adds syntax highlighting, code snippets (including one for `import`), Python-style auto-indentation, an icon for `.capa` files, and a bundled LSP client that starts the language server for you.

To install it, search for `Capa Language` in the Extensions view, or run from a terminal:

```
$ code --install-extension nelsonduarte.capa-language
```

The bundled client auto-detects the server: it uses the `capa` binary if it is on your PATH (from compiler v1.12.0 onward that binary serves the LSP with no extra dependencies), and otherwise falls back to `python -m capa lsp`. With a pip install of the compiler, the server needs the `[lsp]` extra (`pip install "capa[lsp]"`).

Installing from the Repository

Capa ships a VS Code extension that colours `.capa` files, keywords by category, a distinct colour for the built-in capabilities, string interpolation, and operators. The published extension (described above) is the simplest way to install it. To work from the repository instead, or to track local changes, link the `vscode/` folder from the repository into your extensions directory:

```
# macOS / Linux
$ ln -s "$(pwd)/vscode" ~/.vscode/extensions/capa-language
```

Reload VS Code and any `.capa` file is highlighted. Other editors can use the same TextMate grammar from that folder. If you installed the published extension instead, the same highlighting is already in place, no linking required.

The Language Server

For diagnostics, hover, go-to-definition, and more, Capa includes a **language server** that any LSP-capable editor can talk to (Neovim, Helix, Zed, VS Code, JetBrains, Emacs). The published VS Code extension starts and talks to this server for you (see The VS Code Extension above); for other editors you wire it up by hand. To install the server and launch it directly:

```
$ pip install -e '.[lsp]'   # adds the server dependency
$ capa lsp                  # speaks LSP over stdin/stdout
```

The server runs the very same lexer, parser, and analyzer as the CLI, so what it reports always matches what `capa --check` would say. Its features include:

- **Diagnostics** on every edit, with the same `; did you mean 'X'?` hints the compiler gives.
- **Hover** to see a function's signature, or a binding's type and kind.
- **Go-to-definition** and **find-references** for any name.
- **Document outline** of constants, types, traits, capabilities, functions, and impl blocks.
- **Quick fixes** for the "did you mean?" suggestions, applied in one click.
- **Rename** a symbol across the file, and **completion** of keywords, built-ins, locals, and a type's methods after a dot.
- **Semantic highlighting** that colours parameters, capabilities, types, and variants distinctly.

Editor Configuration

For **Helix**, add to `languages.toml`:

```
[[language]]
name = "capa"
language-servers = ["capa"]
file-types = ["capa"]

[language-server.capa]
command = "capa"
args = ["lsp"]
```

For **Neovim** with `nvim-lspconfig`:

```
require("lspconfig").configs.capa = {
  default_config = {
    cmd = { "capa", "lsp" },
    filetypes = { "capa" },
    root_dir = require("lspconfig.util").root_pattern(".git", "."),
  },
}
require("lspconfig").capa.setup({})
```

If `capa` is not on your PATH, replace `command = "capa"` / `cmd = { "capa", ... }` with the explicit `python -m capa lsp` form.

The capa Command, at a Glance

The compiler exposes a pipeline of modes, each one including the ones before it, plus a few tooling flags. The everyday command is `--run`; the everyday diagnostic is `--check`.

- `capa init [name]` - scaffold a new project (`main.capa`, `README.md`, `.gitignore`).
- `capa file.capa` (no flag) - print the lexer token stream. For debugging the lexer.
- `capa --parse file.capa` - print the parsed syntax tree. For debugging the grammar.
- `capa --check file.capa` - run the full analyzer; prints `OK` or precise errors.
- `capa --transpile file.capa` - emit the equivalent Python to see how a construct lowers.
- `capa --run file.capa` - transpile and execute. The everyday flag.
- `capa --fmt file.capa` / `--fmt-check` - rewrite in canonical style, or verify only.
- `capa --doc file.capa` - build an HTML page from `///` doc comments.
- `capa --manifest` / `--cyclonedx` / `--spdx` / `--vex` / `--provenance` - emit the authority graph in five standard formats.
- `capa --watch file.capa` - re-run on every change. Pairs well with experimentation.
- `capa test` - discover and run `tests/test_*.capa`. Add `--wasm` or `--both` for the other backend.

- `capa lsp` - start the language server.

Every invocation also works as `python -m capa <args>` if the `capa` command is not on your PATH.

APPENDIX C

Capa for Python Programmers

Capa's syntax is deliberately close to Python, so a Python programmer feels at home quickly. But the resemblance hides some important, deliberate differences, most of them in service of the capability discipline and the type system. This appendix lists the differences that matter most, then shows how to bring an existing Python program into Capa gradually.

The Key Differences

Authority is in the signature

The biggest difference. In Python, any function can `print`, `open`, read `os.environ`, or make a request, the authority is ambient. In Capa, a function that needs the screen takes a `Stdio`, one that needs files takes an `Fs`, and a function that takes neither cannot do either. Authority flows in through parameters, and the signature is an honest summary of what the function can do.

Bindings: let and var, not bare assignment

Where Python writes `x = 5` and reassigns freely, Capa distinguishes `let x = 5` (immutable) from `var x = 5` (mutable), and a binding's type never changes. Prefer `let`.

No exceptions; failure is a value

There is no `try`/`except`. A function that might fail returns `Option<T>` (a value or nothing) or `Result<T, E>` (a value or an error), and the compiler makes you handle both cases. The `?` operator propagates failure concisely, the way an exception would, but visibly and in the type.

Types are explicit on signatures, and never coerce

Every function parameter and return value is annotated. Local `let`s are usually inferred. And Capa never mixes number types silently: `1 + 1.0` is an error; convert with `to_float` or `to_int`.

Sum types and match instead of duck typing

Capa models "one of several shapes" with sum types and `match`, which is checked for exhaustiveness, so adding a case turns every unhandled `match` into a compile error rather than a silent bug.

Private by default

Where Python makes every module-level name importable, Capa makes them private unless marked `pub`. The public surface is a deliberate contract.

No overloading, no default arguments

One name is one function, and every parameter is passed at every call. Named arguments (`greet(name: "Ana", age: 30)`) keep long calls readable instead.

> **NOTE Hello, world, side by side**
>
> Python: `def main(): print("Hello")`. Capa: `fun main(stdio: Stdio) stdio.println("Hello")`. The whole philosophy is in that one extra parameter, the authority to print, asked for out loud.

Bringing an Existing Python Program Across

You do not have to rewrite a Python program to get a Capa authority manifest from it. The recommended path is gradual: wrap the Python file in a thin Capa shell, then move one function at a time into typed Capa. At every step the `Unsafe` capability shrinks and the manifest gets more honest, while the Python file itself never changes.

Stage 1: a thin Unsafe shell

Crossing into Python uses `py_import` and `py_invoke`, which both require the `Unsafe` capability, Capa's explicit escape hatch. A first version delegates everything to the original module:

```
fun main(stdio: Stdio, u: Unsafe)
    let mod = py_import(u, "my_program")
    py_invoke(u, mod.main, [])
```

The manifest reports `Unsafe` here, which is the honest signal that the program still escapes Capa's analysis. That is the starting point, not the destination.

Stage 2: move one function at a time

Pick the easiest function, usually one that needs a single built-in capability, and rewrite it in typed Capa, leaving the rest delegating to Python. A file write, for example, becomes:

```
fun save_response(fs: Fs, path: String, content: String) ->
Result<Unit, IoError>
    return fs.write(path, content)
```

Now `Fs` appears explicitly in the manifest, exercised by a typed function rather than hidden in an `Unsafe` block. Repeat for the next function.

Stage 3: fully typed, Unsafe gone

Once the last `py_invoke` is migrated, the Python file is unreferenced and can be deleted. `main` now threads exactly the capabilities the program uses and nothing more, and the manifest is a clean per-function authority bound. You can track the journey with `capa migrate <file.capa>`, which reports the share of functions that are already `Unsafe`-free and suggests the cheapest next step.

> **NOTE Honest limits of migration**
>
> Capa's built-in capability surface is intentionally narrow (basic `Net`, `Fs`, `Env`, `Clock`, `Random` methods). A function that needs a richer Python library, say custom HTTP headers or a database driver, can stay on the `Unsafe` side; that lone `Unsafe` then becomes a precise audit signal pointing at the one place needing human review. The Python file is never made safer by migration, only the Capa side becomes auditable.

A Cheat Sheet

A handful of translations to keep nearby as you start writing Capa:

- `x = 5` → `let x = 5` (or `var x = 5` if it changes).
- `def f(a, b):` → `fun f(a: Int, b: Int) -> Int`.
- `print(s)` → `stdio.println(s)` (and `main` takes `stdio: Stdio`).
- `open(p).read()` → `fs.read(p)` returning `Result<String, IoError>`.
- `os.environ.get(k)` → `env.get(k)` returning a `@secret Option<String>`.
- `d[k]` → `m.get(k)` returning an `Option`; handle the missing case.
- `try/except` → `match` on a `Result`, or the `?` operator.
- `f"{x}"` → `"${x}"` (string interpolation is the same idea).

APPENDIX D

Getting Help and Going Further

You have finished the book: the core of the language and three real projects. This short appendix points you to where to turn when you are stuck, where the reference material lives, and how to get involved with Capa as it grows.

When You Are Stuck

Most problems are quickest to solve without leaving your terminal. In rough order:

- **Read the error.** Capa's messages name the line and column, explain the problem, and often suggest the fix with a `; did you mean 'X'?` hint. They are written to be read.
- **Run `capa --check`.** It performs the full analysis without running anything, the fastest way to ask *is my code valid?* and see every diagnostic at once.
- **Consult the reference and standard library.** The language reference covers syntax and semantics; the standard-library reference lists every built-in type, method, and capability. Keep both open while you write.
- **Read an example.** The repository's `examples/` directory has a small, working program for almost every feature, from `hello.capa` to user-defined capabilities, SBOM parsers, and the CVE case studies that motivated the language.

Where to Ask

Capa is an open project with real, monitored channels on GitHub:

- **GitHub Discussions** for open-ended questions and "how do I...?" help, the right place when you are not sure whether something is a bug, a request, or a question.
- **Issues** for concrete bug reports and feature proposals; guided forms collect the details a maintainer will need.
- **Private security advisories** for anything that looks like a way to bypass the capability discipline, escape attenuation, or otherwise break Capa's guarantees, reported privately so it can be fixed before disclosure.

Contributing

Capa is at a stage where a thoughtful issue, a small fix, or a report from real use genuinely shapes its direction. The short version of getting set up:

```
$ git clone https://github.com/nelsonduarte/capa-language
$ cd capa-language
$ pip install -e .
$ python -m unittest discover tests
```

Open an issue first for anything beyond a small fix, so the direction is agreed before the work; keep one concern per pull request. What helps most right now is not compiler patches but **libraries written in Capa**, small, well-tested packages with an honest capability surface (a parser, a data format, a client), plus new examples and reports of where the tooling diverges from the docs on real programs. The `CONTRIBUTING` guide covers the compiler's architecture and what fits.

Going Further

When you are ready to push past this book, a few directions stand out. Write a small **library** of your own and give it an honest capability surface, that is the single most useful thing the young ecosystem needs. Explore the **supply-chain artefacts** beyond the manifest: the CycloneDX and SPDX SBOMs, the per-function VEX claims, and the SLSA provenance the compiler can emit, and how they map onto frameworks like the CRA, NIS2, and DORA. Try the **WebAssembly backend** to package a capability-confined `.wasm` artefact. And if you maintain Python code whose authority surface you want to make auditable, migrate it gradually, as Appendix C describes.

> **NOTE A closing thought**
>
> Capa exists to make one quiet idea concrete: that a program should have to say what it is allowed to do, and a compiler should be able to check it. You have now written programs where that is true, where the screen, the files, the network, and even individual secrets travel only where the types permit. Take that habit with you, even into languages that do not enforce it; code that is honest about its authority is easier to trust, in any language. Welcome to Capa.

APPENDIX E

Solutions to the Exercises

Worked answers to the Try It Yourself exercises, one per chapter, in the order they appear in the book. Each solution gives a short recap, the code, and a line or two on why it works. Every program was checked against Capa 1.12.0; type it, run it, and compare. Some exercises ask you to provoke a compiler error; for those, the listing is the version that triggers it, and the note explains both the message and the fix.

Chapter 1: Getting Started

Exercise 1-1 Hello, You

Modify hello.capa so it prints a greeting with your own name, for example Hello, Ana!.

```
fun main(stdio: Stdio)
    stdio.println("Hello, Nelson!")
```

A single println call writes one line to the screen. main asks for stdio: Stdio because printing is an authority the function must be handed, not something every function may do.

Exercise 1-2 Two Lines

Add a second stdio.println(...) line to main, indented the same way as the first, so the program prints two separate lines of text.

```
fun main(stdio: Stdio)
    stdio.println("Line one.")
    stdio.println("Line two.")
```

Two println calls indented the same way run top to bottom, so the program prints two separate lines.

Exercise 1-3 Break It on Purpose

Delete the stdio: Stdio part from the signature, leaving fun main(), and run capa --check hello.capa.

```
fun main()
    stdio.println("Hello")
```

Dropping stdio: Stdio from the signature removes stdio from scope, and the checker rejects the program with undefined name 'stdio'. A capability exists only if it is received as a parameter; the fix is to put stdio: Stdio back.

Exercise 1-4 Scaffold a Project

Use capa init practice to create a new project, cd into it, and run main.capa.

```
fun main(stdio: Stdio)
    stdio.println("Hello from Capa!")
```

capa init practice scaffolds a project folder with main.capa, capa.toml, .capa-version, .gitignore, and README.md. The generated main.capa already prints a greeting and runs with capa --run.

Chapter 2: Values and Simple Types

Exercise 2-1 About You

Create three let bindings holding your name (a String), your birth year (an Int), and your height in metres (a Float).

```
fun main(stdio: Stdio)
    let nome = "Ana"
    let ano = 1998
    let altura = 1.67
    stdio.println("${nome} nasceu em ${ano} e mede ${altura}
metros.")
```

Three let bindings hold a String, an Int, and a Float, and string interpolation with ${...} weaves all three into one sentence.

Exercise 2-2 Simple Arithmetic

Bind two integers and print their sum, difference, product, and remainder, each on its own line, each labelled (for example sum = 8).

```
fun main(stdio: Stdio)
    let a = 17
    let b = 5
    stdio.println("soma: ${a + b}")
    stdio.println("diferenca: ${a - b}")
    stdio.println("produto: ${a * b}")
    stdio.println("resto: ${a % b}")
    let media = to_float(a + b) / 2.0
```

```
    stdio.println("media: ${media}")
```

The four operators +, -, *, and % work on integers, and to_float converts the sum so the average comes out as a Float rather than a truncated integer.

Exercise 2-3 Mutating a Counter

Use a var counter starting at 0, add 1 to it three times, and print it after each step so you see 1, 2, 3.

```
fun main(stdio: Stdio)
    var counter = 0
    counter = counter + 1
    stdio.println("${counter}")
    counter = counter + 1
    stdio.println("${counter}")
    counter = counter + 1
    stdio.println("${counter}")
```

A var can be reassigned, so the counter climbs from 1 to 3. Assigning a String to it is rejected with cannot assign String to Int: a var may change value but never type.

Exercise 2-4 String Surgery

Bind the string " Capa Language " (with the extra spaces).

```
fun main(stdio: Stdio)
    let original = "  Capa Language  "
    let trimmed = original.trim()
    stdio.println("trimmed: '${trimmed}'")
    stdio.println("upper: ${trimmed.to_upper()}")
    stdio.println("length apos trim: ${trimmed.length()}")
    stdio.println("original inalterado: '${original}'")
```

String methods like trim and to_upper return new strings; the original binding is never mutated, as the final line confirms.

Exercise 2-5 Mixing on Purpose

Write let bad = 5 + 2.0 and run the program.

```
fun main(stdio: Stdio)
    let bad = to_float(5) + 2.0
    stdio.println("${bad}")
```

Capa never mixes Int and Float, so 5 + 2.0 is rejected. Fix it either by making both operands Float (5.0 + 2.0) or by converting the Int explicitly with to_float(5).

Chapter 3: Lists

Exercise 3-1 Your Week

Make a List<String> of the seven days of the week.

```
fun main(stdio: Stdio)
    let dias: List<String> = ["Segunda", "Terca", "Quarta", "Quinta",
"Sexta", "Sabado", "Domingo"]
    let primeiro = dias.first().unwrap_or("nenhum")
    let ultimo = dias.last().unwrap_or("nenhum")
    stdio.println("primeiro: ${primeiro}")
    stdio.println("ultimo: ${ultimo}")
    stdio.println("total: ${dias.length()}")
```

first() and last() return an Option, so unwrap_or supplies a fallback if the list were empty, and length() reports the count.

Exercise 3-2 Greet Each

Make a list of three friends' names and use a for loop to print a personalised greeting for each one, such as Hi, Ana, nice to see you!.

```
fun main(stdio: Stdio)
    let nomes = ["Ana", "Bruno", "Carla"]
    for nome in nomes
        stdio.println("Ola, ${nome}!")
```

A for loop walks the list, binding each name in turn so the body can print a personalised greeting.

Exercise 3-3 Grow It

Start from an empty list, push the numbers 1 through 5 onto it inside a for i in 1..=5 loop, then print the list's length to confirm it is 5.

```
fun main(stdio: Stdio)
    let xs: List<Int> = []
    for i in 1..=5
        xs.push(i)
    stdio.println("length: ${xs.length()}")
```

Pushing inside a for over the inclusive range 1..=5 grows the list to five elements.

Exercise 3-4 Squares with map

From the list [1, 2, 3, 4, 5], use map to build a list of each number's square, then loop over the result and print each square on its own line.

```
fun main(stdio: Stdio)
    let nums = [1, 2, 3, 4, 5]
    let quadrados = nums.map(fun (x: Int) -> Int => x * x)
    for q in quadrados
        stdio.println("${q}")
```

map applies the lambda to every element and returns a new list, leaving the original untouched.

Exercise 3-5 Only the Big Ones

From a list of test scores, use filter to keep only the scores of 10 or more, and print how many passed.

```
fun main(stdio: Stdio)
    let scores = [4, 12, 9, 15, 10, 7, 20]
    let aprovados = scores.filter(fun (s: Int) -> Bool => s >= 10)
    stdio.println("aprovados: ${aprovados.length()}")
    let total = aprovados.fold(0, fun (acc: Int, s: Int) -> Int =>
acc + s)
    stdio.println("total: ${total}")
```

filter keeps the elements that satisfy the predicate, and fold reduces the survivors to a single total starting from 0.

Exercise 3-6 Count to a Hundred

Using a range, build the numbers 1 to 100, keep only the multiples of three with filter, and print how many there are.

```
fun main(stdio: Stdio)
    let mults = (1..=100).filter(fun (n: Int) -> Bool => n % 3 == 0)
    stdio.println("multiplos de 3 em 1..=100: ${mults.length()}")
```

A range can be filtered directly; counting the multiples of 3 in 1..=100 gives 33.

Chapter 4: Maps and Sets

Exercise 4-1 Capital Cities

Build a Map<String, String> from a few countries to their capitals.

```
fun main(stdio: Stdio)
    let capitais: Map<String, String> = new_map()
    capitais.set("Portugal", "Lisboa")
    capitais.set("Franca", "Paris")
    capitais.set("Japao", "Toquio")
    let p = capitais.get("Portugal").unwrap_or("unknown")
    let j = capitais.get("Japao").unwrap_or("unknown")
    let a = capitais.get("Brasil").unwrap_or("unknown")
    stdio.println("Portugal -> ${p}")
    stdio.println("Japao -> ${j}")
    stdio.println("Brasil -> ${a}")
```

A Map associates keys with values, and get returns an Option so a missing key (Brasil) falls back through unwrap_or("unknown").

Exercise 4-2 Inventory

Make a Map<String, Int> of products to quantities.

```
fun main(stdio: Stdio)
    let stock: Map<String, Int> = new_map()
    stock.set("macas", 12)
    stock.set("bananas", 7)
    stock.set("laranjas", 20)
    for (name, qty) in stock.pairs()
        stdio.println("${name}: ${qty}")
    stdio.println("produtos distintos: ${stock.length()}")
```

pairs() yields each (key, value) tuple for the for loop to destructure, and length() reports how many distinct keys the map holds.

Exercise 4-3 Word Count

Given a list of words (with repeats), build a Map<String, Int> of how many times each word appears, using the lookup-then-update pattern.

```
fun main(stdio: Stdio)
    let palavras = ["sol", "lua", "sol", "estrela", "lua", "sol"]
    let contagens: Map<String, Int> = new_map()
    for w in palavras
        let atual = contagens.get(w).unwrap_or(0)
        contagens.set(w, atual + 1)
    for (palavra, n) in contagens.pairs()
        stdio.println("${palavra}: ${n}")
```

The lookup-then-update pattern reads the current count with unwrap_or(0) and writes back one more, building a word-frequency map in a single pass.

Exercise 4-4 Unique Visitors

Given a list of visitor names with duplicates, add them all to a Set<String> and print how many distinct visitors there were, then list them.

```
fun main(stdio: Stdio)
    let visitantes = ["Ana", "Bruno", "Ana", "Carla", "Bruno",
"Diogo"]
    let distintos: Set<String> = new_set()
    for v in visitantes
        distintos.add(v)
    stdio.println("distintos: ${distintos.length()}")
    for nome in distintos
        stdio.println("  ${nome}")
```

A Set discards duplicates automatically, so its length is the number of distinct visitors.

Exercise 4-5 Seen Before?

Loop over a list of numbers and, using a Set<Int>, print each number the first time you see it and the word repeat every later time.

```
fun main(stdio: Stdio)
    let nums = [3, 1, 3, 7, 1, 9, 7, 3]
    let vistos: Set<Int> = new_set()
    for n in nums
        if vistos.contains(n)
            stdio.println("repeat")
        else
            vistos.add(n)
            stdio.println("${n}")
```

contains tests membership before insertion, so each value prints once on its first appearance and reports repeat thereafter.

Chapter 5: Control Flow

Exercise 5-1 Number Sign

Bind an integer and use an if-elif-else chain to print whether it is positive, negative, or zero.

```
fun classify(n: Int, stdio: Stdio)
    if n > 0
        stdio.println("positive")
    elif n < 0
        stdio.println("negative")
```

```
    else
        stdio.println("zero")

fun main(stdio: Stdio)
    classify(7, stdio)
    classify(-4, stdio)
    classify(0, stdio)
```

An if / elif / else chain selects exactly one branch, classifying the integer as positive, negative, or zero.

Exercise 5-2 Grades

Write a program that turns a numeric score into a letter grade with an if-elif-else chain (A for 90+, B for 80+, C for 70+, D for 60+, otherwise F).

```
fun letter(score: Int) -> String
    if score >= 90
        return "A"
    elif score >= 80
        return "B"
    elif score >= 70
        return "C"
    elif score >= 60
        return "D"
    else
        return "F"

fun main(stdio: Stdio)
    let score = 84
    stdio.println("grade: ${letter(score)}")
    let outcome = if score >= 60 then "pass" else "fail"
    stdio.println("outcome: ${outcome}")
```

The letter function uses statement-form if/elif/else, while the pass/fail decision uses the expression form if ... then ... else, which yields a value directly.

Exercise 5-3 Admission

Given an age and a Bool has_ticket, print Welcome only if the person is at least 18 and holds a ticket, and an explanatory message otherwise.

```
fun main(stdio: Stdio)
    let age = 20
    let has_ticket = true
    if age >= 18 and has_ticket
        stdio.println("Welcome")
    else
        stdio.println("Sorry, you cannot enter")
```

and is true only when both operands are, so Welcome prints exactly when the visitor is old enough and holds a ticket.

Exercise 5-4 Countdown

Use a while loop and a var to count down from 5 to 1, printing each number, then print Lift off!

```
fun main(stdio: Stdio)
    var n = 5
    while n >= 1
        stdio.println("${n}")
        n -= 1
    stdio.println("Lift off!")
```

A while loop with a var counts down, decrementing each pass until the condition fails, then prints the closing line.

Exercise 5-5 First Multiple

Loop over the numbers 1 to 100 and print the first one that is divisible by both 3 and 7, then break.

```
fun main(stdio: Stdio)
    var n = 1
    while n <= 100
        if n % 3 == 0 and n % 7 == 0
            stdio.println("first divisible by 3 and 7: ${n}")
            break
        n += 1
```

break leaves the loop the moment the first number divisible by both 3 and 7 is found, so 21 prints and nothing after it.

Exercise 5-6 Skip the Negatives

Given a list of integers that includes some negative values, loop over it, use continue to skip the negatives, and print the sum of only the non-negative numbers.

```
fun main(stdio: Stdio)
    let numbers = [4, -2, 7, -9, 3, -1, 10]
    var total = 0
    for n in numbers
        if n < 0
            continue
        total += n
    stdio.println("sum of non-negatives: ${total}")
```

continue skips the rest of the current iteration for negative values, so only the non-negative numbers reach the running total.

Chapter 6: Functions

Exercise 6-1 Greeter

Write a function greeting(name: String) -> String that returns a friendly sentence using the name, then call it from main and print the result for two different names.

```
fun greeting(name: String) -> String
    return "Hello, ${name}!"

fun main(stdio: Stdio)
    stdio.println(greeting("Ana"))
    stdio.println(greeting("Bruno"))
```

A function with an explicit String return type builds and returns the greeting, and main prints it for two different names.

Exercise 6-2 Celsius to Fahrenheit

Write to_fahrenheit(celsius: Float) -> Float that returns celsius * 9.0 / 5.0 + 32.0.

```
fun to_fahrenheit(celsius: Float) -> Float
    return celsius * 9.0 / 5.0 + 32.0

fun main(stdio: Stdio)
    stdio.println("0C = ${to_fahrenheit(0.0)}F")
    stdio.println("100C = ${to_fahrenheit(100.0)}F")
    stdio.println("37C = ${to_fahrenheit(37.0)}F")
```

The conversion formula is a one-line return; the Float annotations make the arithmetic unambiguous.

Exercise 6-3 Named Call

Write describe(name: String, age: Int, city: String) -> String and call it once using only named arguments.

```
fun describe(name: String, age: Int, city: String) -> String
    return "${name} is ${age} and lives in ${city}"

fun main(stdio: Stdio)
    let s = describe(name: "Ana", age: 30, city: "Porto")
    stdio.println(s)
```

Named arguments label each value at the call site. A positional argument after a named one is rejected; name them all, or keep the positionals first.

Exercise 6-4 Sum and Count

Write sum_and_count(numbers: List<Int>) -> (Int, Int) that returns both the total and how many numbers there were.

```
fun sum_and_count(numbers: List<Int>) -> (Int, Int)
    var total = 0
    var count = 0
    for n in numbers
        total += n
        count += 1
    return (total, count)

fun main(stdio: Stdio)
    let (total, count) = sum_and_count([10, 20, 30, 40, 50])
    let average = to_float(total) / to_float(count)
    stdio.println("total = ${total}, count = ${count}")
    stdio.println("average = ${average}")
```

The function returns a two-element tuple, and main destructures it with let (total, count), converting with to_float for an honest average.

Exercise 6-5 Printing Helper

Write print_banner(stdio: Stdio, text: String) (returning nothing) that prints the text surrounded by a line of dashes above and below.

```
fun print_banner(stdio: Stdio, text: String)
    stdio.println("--------")
    stdio.println(text)
    stdio.println("--------")

fun main(stdio: Stdio)
    print_banner(stdio, "Hello")
```

A function that returns nothing still threads stdio: Stdio so it can print; main passes its own stdio down.

Exercise 6-6 Apply a Function

Write apply_to_all(numbers: List<Int>, f: Fun(Int) -> Int) -> List<Int> that returns a new list with f applied to each element (use the list's map).

```
fun double(x: Int) -> Int
    return x * 2

fun apply_to_all(numbers: List<Int>, f: Fun(Int) -> Int) -> List<Int>
    return numbers.map(f)

fun main(stdio: Stdio)
    let xs = [1, 2, 3, 4]
    let doubled = apply_to_all(xs, double)
    stdio.println("doubled[0]=${doubled[0]}
doubled[3]=${doubled[3]}")
    let squared = apply_to_all(xs, fun (x: Int) -> Int => x * x)
    stdio.println("squared[3]=${squared[3]}")
```

A function-typed parameter f: Fun(Int) -> Int accepts both a named function and an inline lambda, so apply_to_all works with either.

Chapter 7: Structs and Sum Types

Exercise 7-1 A Book

Define a struct Book with fields title: String, author: String, and year: Int.

```
type Book {
    title: String,
    author: String,
    year: Int
}

fun main(stdio: Stdio)
    let b1 = Book { title: "Capa", author: "Nelson", year: 2026 }
    let b2 = Book { title: "SICP", author: "Abelson", year: 1985 }
    stdio.println("${b1.title} by ${b1.author} (${b1.year})")
    stdio.println("${b2.title} by ${b2.author} (${b2.year})")
    match b1
        Book { title, author, year } ->
            stdio.println("destructured: ${title} / ${author} /
${year}")
```

A struct groups named fields; the values are read with dot access for the sentences and unpacked with a match on Book { ... } for the destructured line.

Exercise 7-2 Circle Methods

Define a struct Circle with a radius: Float.

```
type Circle {
    radius: Float
}

impl Circle
    fun area(self) -> Float
```

```
        return 3.14159 * self.radius * self.radius
    fun diameter(self) -> Float
        return self.radius * 2.0

fun main(stdio: Stdio)
    let c = Circle { radius: 3.0 }
    stdio.println("area = ${c.area()}")
    stdio.println("diameter = ${c.diameter()}")
```

An impl block attaches methods to a struct, each taking self to read the fields, so area and diameter compute from the radius.

Exercise 7-3 Constructor

Add a constructor function unit_circle() -> Circle that returns a circle of radius 1.0, and use it in main.

```
type Circle {
    radius: Float
}

impl Circle
    fun area(self) -> Float
        return 3.14159 * self.radius * self.radius

fun unit_circle() -> Circle
    return Circle { radius: 1.0 }

fun main(stdio: Stdio)
    let c = unit_circle()
    stdio.println("unit radius = ${c.radius}")
    stdio.println("unit area = ${c.area()}")
```

A constructor is just a function that returns a struct value, here a Circle with radius 1.0.

Exercise 7-4 Traffic Light

Define a payloadless sum type Light with variants Red, Amber, and Green.

```
type Light =
    Red
    Amber
    Green

fun action(light: Light) -> String
    return match light
        Red -> "stop"
        Amber -> "wait"
        Green -> "go"

fun main(stdio: Stdio)
    stdio.println(action(Red))
    stdio.println(action(Amber))
    stdio.println(action(Green))
```

A payloadless sum type lists its variants, and match maps each to an action. The check is exhaustive: drop a variant and the compiler refuses the match until every case is handled.

Exercise 7-5 Shapes

Using the Shape sum from this chapter, write perimeter(shape: Shape) -> Float with a match (circle: 2.0 * 3.14159 * r; rectangle: 2.0 * (w + h); square: 4.0 * side).

```
type Shape =
    Circle(Float)
    Rectangle((Float, Float))
    Square(Float)

fun perimeter(shape: Shape) -> Float
    return match shape
        Circle(r) -> 2.0 * 3.14159 * r
        Rectangle((w, h)) -> 2.0 * (w + h)
```

```
        Square(side) -> 4.0 * side

fun main(stdio: Stdio)
    stdio.println("circle   = ${perimeter(Circle(1.0))}")
    stdio.println("rect     = ${perimeter(Rectangle((3.0, 4.0)))}")
    stdio.println("square   = ${perimeter(Square(5.0))}")
```

Each variant of Shape carries its own payload, and the match arms destructure those payloads to compute the perimeter.

Exercise 7-6 Events

Extend the Event type with a new variant Signup(String).

```
type Event =
    Login(String)
    Logout(String)
    Failure((Int, String))
    Signup(String)

fun handle(e: Event) -> String
    return match e
        Login(user) -> "${user} logged in"
        Logout(user) -> "${user} logged out"
        Failure((code, msg)) -> "error ${code}: ${msg}"
        Signup(user) -> "${user} signed up"

fun main(stdio: Stdio)
    stdio.println(handle(Login("ana")))
    stdio.println(handle(Logout("ana")))
    stdio.println(handle(Failure((404, "not found"))))
    stdio.println(handle(Signup("bruno")))
```

Adding the Signup variant forces a matching arm everywhere Event is matched; exhaustiveness checking turns a forgotten case into a compile error rather than a runtime surprise.

Chapter 8: Errors as Values

Exercise 8-1 Safe Lookup

Build a Map<String, Int> of a few products to prices.

```
fun price_of(stock: Map<String, Int>, name: String) -> Int
    return stock.get(name).unwrap_or(0)

fun main(stdio: Stdio)
    let stock: Map<String, Int> = new_map()
    stock.set("apple", 120)
    stock.set("pear", 95)

    let apple = price_of(stock, "apple")
    let banana = price_of(stock, "banana")
    stdio.println("apple: ${apple}")
    stdio.println("banana: ${banana}")
```

get returns an Option, and unwrap_or(0) turns a missing product into the price 0, so the function always returns an Int.

Exercise 8-2 Parse or Default

Write read_number(text: String) -> Int that returns the parsed integer, or -1 if the text is not a number.

```
// Version A: sentinel -1 on invalid input.
fun read_number(text: String) -> Int
    match parse_int(text)
        Some(n) -> return n
        None -> return -1

// Version B: honest Option<Int>, the caller decides what "invalid"
means.
fun read_number_opt(text: String) -> Option<Int>
    return parse_int(text)
```

```
fun main(stdio: Stdio)
    let a_ok = read_number("42")
    let a_bad = read_number("oops")
    stdio.println("A: ${a_ok}")
    stdio.println("A: ${a_bad}")

    match read_number_opt("42")
        Some(n) -> stdio.println("B: got ${n}")
        None -> stdio.println("B: not a number")

    match read_number_opt("oops")
        Some(n) -> stdio.println("B: got ${n}")
        None -> stdio.println("B: not a number")
```

The sentinel version returns -1 for bad input, which cannot be told apart from a real -1; the Option<Int> version returns None, letting the caller decide what missing means.

Exercise 8-3 Safe Divide Chain

Using the safe_divide from this chapter, write fun divide_twice(a: Float, b: Float, c: Float) -> Option<Float> that divides a by b, then that result by c, using ?

```
fun safe_divide(a: Float, b: Float) -> Option<Float>
    if b == 0.0
        return None
    return Some(a / b)

// divide_twice computes (a / b) / c. The ? operator short-circuits:
// if either division hits a zero divisor, the whole thing is None.
fun divide_twice(a: Float, b: Float, c: Float) -> Option<Float>
    let first = safe_divide(a, b)?
    return safe_divide(first, c)

fun main(stdio: Stdio)
    match divide_twice(100.0, 2.0, 5.0)
        Some(r) -> stdio.println("result: ${r}")
        None -> stdio.println("undefined (division by zero)")
```

```
    match divide_twice(100.0, 0.0, 5.0)
        Some(r) -> stdio.println("result: ${r}")
        None -> stdio.println("undefined (division by zero)")

    match divide_twice(100.0, 2.0, 0.0)
        Some(r) -> stdio.println("result: ${r}")
        None -> stdio.println("undefined (division by zero)")
```

The ? operator unwraps a Some or short-circuits to None, so divide_twice yields None if either division hits a zero divisor.

Exercise 8-4 Validated Result

Write parse_grade(text: String) -> Result<Int, String> that parses the text and returns Err("not a number") if parsing fails, Err("out of range") if the number is not between 0 and 100, and Ok(n) othe

```
// parse_grade distinguishes two failure modes from success, the
// classic reason to reach for Result over Option: the Err carries
WHY.
fun parse_grade(text: String) -> Result<Int, String>
    match parse_int(text)
        None -> return Err("not a number")
        Some(n) ->
            if n < 0 or n > 100
                return Err("out of range")
            return Ok(n)

fun report(stdio: Stdio, text: String)
    match parse_grade(text)
        Ok(n) -> stdio.println("'${text}' -> grade ${n}")
        Err(msg) -> stdio.println("'${text}' rejected: ${msg}")

fun main(stdio: Stdio)
    report(stdio, "87")
    report(stdio, "hello")
    report(stdio, "150")
    report(stdio, "0")
```

Result carries the reason for failure: Err("not a number") and Err("out of range") are distinct, which Option could not express.

Exercise 8-5 Propagate It

Write fun second_word(line: String) -> Option<String> that splits the line on spaces and returns the second word, using ?

```
// second_word splits on spaces and returns the second token, if any.
// get(1) returns Option<String>; ? unwraps it or bails out with
None.
fun second_word(line: String) -> Option<String>
    let words = line.split(" ")
    let w = words.get(1)?
    return Some(w)

fun main(stdio: Stdio)
    match second_word("hello there world")
        Some(w) -> stdio.println("second: ${w}")
        None -> stdio.println("no second word")

    match second_word("lonely")
        Some(w) -> stdio.println("second: ${w}")
        None -> stdio.println("no second word")
```

get(1) returns an Option, and ? either unwraps the second word or returns None for a line that is too short.

Chapter 9: Your First Capability

Exercise 9-1 File Length

Write a program whose main takes stdio: Stdio and fs: Fs, reads hello.txt using ?

```
// main returns a Result so it can use ? on the file read. If the
read
// fails, the error propagates and the program exits non-zero, fail-
closed.
fun main(stdio: Stdio, fs: Fs) -> Result<Unit, IoError>
    let text = fs.read("hello.txt")?
    stdio.println("hello.txt is ${text.length()} bytes")
    return Ok(())
```

Returning Result<Unit, IoError> lets main use ? on the file read; if the read fails the error propagates and the program exits non-zero, fail-closed.

Exercise 9-2 Pass It Down

Write print_line(stdio: Stdio, text: String) and a print_all(stdio: Stdio, lines: List<String>) that calls it in a loop, then call print_all from main.

```
fun print_line(stdio: Stdio, text: String)
    stdio.println(text)

fun print_all(stdio: Stdio, lines: List<String>)
    for line in lines
        print_line(stdio, line)

fun main(stdio: Stdio)
    let lines = ["first", "second", "third"]
    print_all(stdio, lines)
```

The Stdio capability flows explicitly through every signature, from main down to print_line, so authority is always visible in the types.

Exercise 9-3 A Pure Function

Write a function total(numbers: List<Int>) -> Int that takes no capabilities and sums the list.

```
// total takes no capability: it only reads its argument and does
// arithmetic. Its manifest entry is [], pure. main holds the Stdio.
fun total(numbers: List<Int>) -> Int
    var sum = 0
    for n in numbers
        sum += n
    return sum

fun main(stdio: Stdio)
    let t = total([1, 2, 3, 4, 5])
    stdio.println("total: ${t}")
```

total takes no capability at all: it only reads its argument and does arithmetic. Its manifest entry is the empty list, the signature of a pure function.

Exercise 9-4 Catch the Compiler

Inside a pure function, try to add a stdio.println(...) call without giving the function a Stdio parameter, and read the undefined name 'stdio' error.

```
// BROKEN: shout claims to be pure (no Stdio parameter) yet tries to
// print. There is no `stdio` in scope, so the analyzer rejects it.
fun shout(text: String)
    stdio.println(text)

fun main(stdio: Stdio)
    shout("hello")
```

A function without stdio: Stdio has no stdio in scope, so the call is rejected with undefined name 'stdio'. The fix is to thread the capability in through the signature, the only way to gain authority in Capa.

Exercise 9-5 Read the Manifest

Run capa --manifest yourfile.capa on the program from 9-2 and find your functions in the output.

```
capa --manifest 9-3.capa
```

capa --manifest prints one entry per function. total shows declared_capabilities of the empty list and main shows Stdio: the manifest is a per-function SBOM of authority, read straight from the signatures.

Chapter 10: Attenuating Capabilities

Exercise 10-1 One Host Only

Write fun fetch_users(net: Net) -> Result<String, IoError> that gets https://api.example.com/users.

```
// fetch_users sees a plain Net. It cannot tell whether the Net was
// narrowed: the restriction lives in the value, enforced at runtime.
fun fetch_users(net: Net) -> Result<String, IoError>
    return net.get("https://api.example.com/users")

fun main(stdio: Stdio, net: Net) -> Result<Unit, IoError>
    // Narrow authority to a single host before handing it off.
    let scoped = net.restrict_to("api.example.com")
    let body = fetch_users(scoped)?
    stdio.println("got ${body.length()} bytes")
    return Ok(())
```

fetch_users sees a plain Net and cannot tell it has been narrowed; restrict_to in main scopes the authority to a single host before handing it off, enforced at runtime.

Exercise 10-2 Watch It Deny

Add a second call inside fetch_users to https://evil.example.com/.

```
// fetch_two reaches two different hosts. The Net it receives has
been
// narrowed to api.example.com, so the second call
(analytics.example.com)
// is outside the allowed set and is denied at runtime, fail-closed.
fun fetch_two(net: Net) -> Result<String, IoError>
    let users = net.get("https://api.example.com/users")?
    // This host is NOT in the restriction set, runtime raises
NetRestricted.
    let stats = net.get("https://analytics.example.com/track")?
    return Ok("${users.length()} + ${stats.length()}")

fun main(stdio: Stdio, net: Net) -> Result<Unit, IoError>
    let scoped = net.restrict_to("api.example.com")
    match fetch_two(scoped)
        Ok(s) -> stdio.println("ok: ${s}")
        Err(e) -> stdio.eprintln("denied: ${e}")
    return Ok(())
```

The narrowed Net allows api.example.com but denies any other host. The second call is refused at runtime, fail-closed, with no packet leaving for the unauthorised host.

Exercise 10-3 Restrict the Filesystem

Write a function that writes a short note to /tmp/notes/today.txt.

```
// save_note writes under a directory it does not get to choose: the
// prefix restriction is baked into the Fs value it receives.
```

```
fun save_note(fs: Fs, text: String) -> Result<Unit, IoError>
    return fs.write("/tmp/notes/today.txt", text)

fun main(stdio: Stdio, fs: Fs) -> Result<Unit, IoError>
    // Narrow to the notes directory. A later attempt to read or
write
    // /etc/passwd through this Fs is denied at runtime
(FsRestricted),
    // even though save_note only ever sees a bare `Fs`.
    let notes_fs = fs.restrict_to("/tmp/notes/")
    save_note(notes_fs, "buy milk\n")?
    stdio.println("note saved")
    return Ok(())
```

restrict_to bakes a path prefix into the Fs value. A write under the allowed prefix succeeds; an attempt on /etc/passwd is denied at runtime with the allowed prefixes named.

Exercise 10-4 You Cannot Widen

In main, narrow net to "api.example.com", then call restrict_to again on the result with a different host.

```
// Narrowing is monotonic: it can only shrink authority. Narrowing to
// host A and then to host B yields a Net whose allowed set is the
// intersection {A} n {B} = {} (empty). Both hosts are then denied.
// We use `allows` (a pure query, no IO) so the demo is offline.
fun main(stdio: Stdio, net: Net)
    let a = net.restrict_to("api.example.com")
    let b = a.restrict_to("cdn.example.com")

    let a_api = a.allows("api.example.com")
    let b_api = b.allows("api.example.com")
    let b_cdn = b.allows("cdn.example.com")

    stdio.println("a allows api.example.com?  ${a_api}")
    stdio.println("b allows api.example.com?  ${b_api}")
    stdio.println("b allows cdn.example.com?  ${b_cdn}")
```

Narrowing only shrinks authority. Restricting to host A and then to host B leaves the intersection, which is empty, so afterwards both hosts are denied; the pure allows query shows this offline.

Exercise 10-5 Deterministic Random

Using random.with_seed(42), print five random integers with int_range(1, 100).

```
// with_seed pins the PRNG so draws are reproducible: same seed, same
// sequence, every run. Run the program twice and the five numbers
match.
fun main(stdio: Stdio, random: Random)
    let rng = random.with_seed(42)
    var i = 0
    while i < 5
        stdio.println("${rng.int_range(1, 100)}")
        i += 1
```

with_seed pins the PRNG, so the same seed produces the same sequence on every run: deterministic, reproducible random.

Chapter 11: Defining Your Own Capabilities

Exercise 11-1 Store a User

Define capability StoreUser with one method save(self, id: String, payload: String) -> Result<Unit, IoError>.

```
// A user-defined capability: anyone holding a StoreUser can save a
user,
// and nobody else can. The concrete FileStore wraps an Fs (allowed
// because cap-bearing structs may hold built-in caps as fields).
capability StoreUser
```

```
    fun save(self, id: String, payload: String) -> Result<Unit,
IoError>

type FileStore {
    dir: String,
    fs: Fs
}

impl StoreUser for FileStore
    fun save(self, id: String, payload: String) -> Result<Unit,
IoError>
        let path = "${self.dir}/${id}.txt"
        return self.fs.write(path, payload)

// Factory: a capability-bearing struct must be produced by a call,
not
// bound directly from a struct literal in a `let`.
fun new_file_store(fs: Fs) -> FileStore
    return FileStore { dir: "/tmp/users", fs:
fs.restrict_to("/tmp/users/") }

fun main(stdio: Stdio, fs: Fs)
    let store = new_file_store(fs)
    match store.save("alice", "Alice Smith")
        Ok(_) -> stdio.println("saved alice")
        Err(e) -> stdio.eprintln("save failed: ${e}")
```

A user-defined capability declares a method, and a struct implements it; the FileStore wraps an Fs as a field. A capability-bearing struct must come from a factory call, not a bare let binding.

Exercise 11-2 Factory and Narrowing

Write a factory make_file_store(fs: Fs) -> FileStore.

```
capability StoreUser
    fun save(self, id: String, payload: String) -> Result<Unit,
IoError>

type FileStore {
```

```
    dir: String,
    fs: Fs
}

impl StoreUser for FileStore
    fun save(self, id: String, payload: String) -> Result<Unit,
IoError>
        let path = "${self.dir}/${id}.txt"
        return self.fs.write(path, payload)

// Factory: consumes a low-level Fs and produces the high-level cap.
// The Fs is narrowed inside, so a FileStore can only ever touch
/var/users/.
fun make_file_store(fs: Fs) -> FileStore
    return FileStore { dir: "/var/users", fs:
fs.restrict_to("/var/users/") }

fun main(stdio: Stdio, fs: Fs)
    // Restrict before the factory ever sees the Fs: the store
inherits
    // an already-narrowed authority.
    let scoped = fs.restrict_to("/var/users/")
    let store = make_file_store(scoped)
    match store.save("bob", "Bob Jones")
        Ok(_) -> stdio.println("saved bob")
        Err(e) -> stdio.eprintln("save failed: ${e}")
```

The factory narrows the Fs to /var/users/ before building the store, so every FileStore it produces inherits an already-scoped authority.

Exercise 11-3 Use the Contract

Write fun register(store: StoreUser, id: String) -> Result<Unit, IoError> that saves a small payload through the capability.

```
capability StoreUser
    fun save(self, id: String, payload: String) -> Result<Unit,
IoError>

type FileStore {
```

```
    dir: String,
    fs: Fs
}

impl StoreUser for FileStore
    fun save(self, id: String, payload: String) -> Result<Unit,
IoError>
        let path = "${self.dir}/${id}.txt"
        return self.fs.write(path, payload)

fun make_file_store(fs: Fs) -> FileStore
    return FileStore { dir: "/var/users", fs:
fs.restrict_to("/var/users/") }

// register takes the StoreUser capability, not an Fs. Its manifest
lists
// StoreUser only: it has exactly the authority to save users, no
more.
fun register(store: StoreUser, id: String) -> Result<Unit, IoError>
    return store.save(id, "registered: ${id}")

fun main(stdio: Stdio, fs: Fs)
    let store = make_file_store(fs.restrict_to("/var/users/"))
    match register(store, "carol")
        Ok(_) -> stdio.println("registered carol")
        Err(e) -> stdio.eprintln("register failed: ${e}")
```

register takes the StoreUser capability, not an Fs, so its manifest lists exactly the authority to save users and nothing more about the filesystem.

Exercise 11-4 A Plain Trait

Define a trait Area with area(self) -> Float, and implement it for two different shape structs.

```
// A trait (plain polymorphism, NOT a capability): two shapes compute
// their area differently, and one function accepts any Area.
trait Area
    fun area(self) -> Float
```

```
type Rectangle {
    width: Float,
    height: Float
}

type Circle {
    radius: Float
}

impl Area for Rectangle
    fun area(self) -> Float
        return self.width * self.height

impl Area for Circle
    fun area(self) -> Float
        return 3.14159 * self.radius * self.radius

// Accepts any value whose type implements Area; dispatch is dynamic.
fun print_area(stdio: Stdio, shape: Area)
    stdio.println("area = ${shape.area()}")

fun main(stdio: Stdio)
    let r: Area = Rectangle { width: 3.0, height: 4.0 }
    let c: Area = Circle { radius: 2.0 }
    print_area(stdio, r)
    print_area(stdio, c)
```

A trait is plain polymorphism, not a capability: two structs implement Area differently, and one function accepts any Area with dynamic dispatch. The values bind to let with no restriction.

Exercise 11-5 Read the Manifest

Run capa --manifest on your program from 11-3 and confirm that register lists StoreUser (not Fs) under its declared capabilities, and that the implementor appears in the user-defined capabilities sect

```
capa --manifest 11-3.capa
```

The manifest shows register with StoreUser, not Fs. The filesystem authority is sealed inside FileStore by the factory, making the boundary auditable function by function.

Chapter 12: Modules, Visibility and Packages

Exercise 12-1 Two Files

Create mathutil.capa with a pub fun square(n: Int) -> Int, and a main.capa that imports it and prints a few squares.

```
// mathutil.capa
pub fun square(n: Int) -> Int
    return n * n
```

```
// main.capa
import mathutil

fun main(stdio: Stdio)
    stdio.println("2^2 = ${square(2)}")
    stdio.println("5^2 = ${square(5)}")
    stdio.println("9^2 = ${square(9)}")
```

Only pub symbols cross a module boundary. With pub, main can call square; remove pub and the import fails with undefined name 'square' (private to module).

Exercise 12-2 A Folder of Helpers

Make a text/ folder containing case.capa (with a pub function that upper-cases a string) and pad.capa (with a pub function that surrounds a string in brackets).

```
// text/case.capa
pub fun shout(s: String) -> String
    return s.to_upper()
```

```
// text/pad.capa
pub fun bracket(s: String) -> String
    return "[${s}]"
```

```
// main.capa
import text.case
import text.pad

fun main(stdio: Stdio)
    let word = "capa"
    stdio.println(bracket(shout(word)))
```

A folder of small modules is imported with dotted paths, and the public functions compose: bracket(shout(word)) uppercases then wraps the word.

Exercise 12-3 Qualified Calls

Import a module two ways, plainly and as an alias, and call the same function both by its bare name and through the alias, confirming they do the same thing.

```
// greet.capa
pub fun hello(name: String) -> String
    return "Hello, ${name}!"
```

```
// main.capa
import greet as G

fun main(stdio: Stdio)
    stdio.println(hello("bare"))      // caminho simples
```

```
    stdio.println(G.hello("via G"))   // caminho qualificado pelo
alias
```

import greet as G exposes the function both bare (hello) and qualified (G.hello); one import already offers both call paths.

Exercise 12-4 Split the Mailer

Take the email example from Chapter 11 and split it across files: mailer.capa exports pub capability SendEmail and pub fun make_mailer(...); users.capa exports a function that takes a SendEmail and se

```
// mailer.capa
pub capability SendEmail
    fun send(self, to: String, body: String) -> Result<Unit, IoError>

pub type ConsoleMailer {
    stdio: Stdio
}

impl SendEmail for ConsoleMailer
    fun send(self, to: String, body: String) -> Result<Unit, IoError>
        self.stdio.println("to ${to}: ${body}")
        return Ok(())

pub fun make_mailer(stdio: Stdio) -> ConsoleMailer
    return ConsoleMailer { stdio: stdio }
```

```
// users.capa
import mailer

pub fun welcome(m: SendEmail, name: String) -> Result<Unit, IoError>
    return m.send(name, "Welcome to Capa, ${name}")
```

```
// main.capa
import mailer
```

```
import users

fun main(stdio: Stdio)
    let m = make_mailer(stdio)
    let r = welcome(m, "alice")
    match r
        Ok(_) -> stdio.println("done")
        Err(e) -> stdio.eprintln("failed: ${e}")
```

The capability, its concrete type, and the factory are all pub, so other files can use them while the impl stays sealed in mailer.capa. The authority chain is visible in the signatures.

Exercise 12-5 A Dependency

Create a capa.toml for a small project and add a dependency on a registry library (for example capa_log) with capa add.

```
# capa.toml
[package]
name = "my-project"
version = "0.1.0"
capa = ">=0.8.4"          # versao minima de Capa (opcional)

[dependencies]
capa_log = { git = "https://github.com/nelsonduarte/capa_log", tag =
"v0.1" }
```

A minimal capa.toml names the package and lists dependencies. capa add declares a dependency (and installs it by default), and capa install resolves them, vendors the code, and writes a capa.lock for reproducible builds.

Chapter 13: Information-Flow Control

Exercise 13-1 Catch a Leak

Write fun show(stdio: Stdio, secret_word: @secret String) that tries to println the secret directly.

```
// show_strict.capa
@strict_ifc()
fun show(stdio: Stdio, secret_word: @secret String)
    stdio.println(secret_word)

fun main(stdio: Stdio)
    show(stdio, "hunter2")
```

Leaking a @secret to a public sink is a warning by default and an error under @strict_ifc(). The only sanctioned bridge is declassify(value, reason: "..."); @strict_ifc() is what turns the advice into an obligation.

Exercise 13-2 Join in Action

Inside a function with a @secret name, build a greeting "Hello, ${name}" and try to print it.

```
// greet.capa
@strict_ifc()
fun greet(stdio: Stdio, name: @secret String)
    let line = "Hello, ${name}"
    stdio.println(line)

fun main(stdio: Stdio)
    greet(stdio, "Ada")
```

String interpolation joins labels, so a string built from a @secret name is itself @secret. Printing it is still a violation: the label survives the join.

Exercise 13-3 Declassify with a Reason

Write a mask(secret: @secret String) -> String that returns only the first character followed by "***".

```
// mask.capa
fun mask(secret: @secret String) -> String
    let head = secret.substring(0, 1)
    return "${head}***"

fun main(stdio: Stdio)
    let pw: @secret String = "swordfish"
    let shown = declassify(mask(pw), reason: "show only first letter,
rest masked")
    stdio.println(shown)
```

mask still returns a @secret value, because the label propagates through substring and interpolation. declassify is the single audited bridge from @secret to @public, and it requires a literal reason for the SBOM to record.

Exercise 13-4 An Environment Secret

Recall that env.get returns a secret value.

```
// ok.capa
fun mask(s: @secret String) -> String
    return "${s.substring(0, 1)}***"

fun main(stdio: Stdio, env: Env)
    match env.get("API_KEY")
        Some(key) ->
            let shown = declassify(mask(key), reason: "log only a
masked API key for diagnostics")
            stdio.println("API_KEY = ${shown}")
        None -> stdio.println("no API_KEY set")
```

env.get is secret-by-default, so printing the value directly is a violation. Routing only the masked key through declassify with an honest reason discloses just what is intended.

Exercise 13-5 No Laundering

Put a @secret value into a one-element list, then take it back out and try to print it.

```
// launder.capa
fun main(stdio: Stdio)
    let pw: @secret String = "swordfish"
    let box: List<String> = [pw]
    let back = box[0]
    stdio.println(back)
```

Putting a @secret into a list and indexing it back out does not strip the label: collections are treated as whole aggregates, so there is no laundering.

Chapter 14: Testing Your Code

Exercise 14-1 First Test

Write a function add(a: Int, b: Int) -> Int, and a test that panics unless add(2, 3) equals 5 and add(-1, 1) equals 0.

```
// mathadd.capa
pub fun add(a: Int, b: Int) -> Int
    return a + b
```

```
// tests/test_add.capa
import mathadd
```

```
fun main(stdio: Stdio)
    if add(2, 3) != 5
        panic("add(2, 3) should be 5")
    if add(-1, 1) != 0
        panic("add(-1, 1) should be 0")
    stdio.println("add tests passed")
```

A test is a main that panics on a failed expectation; capa test discovers tests/test_*.capa and runs them. Break add and the panic makes the test fail with the message shown inline.

Exercise 14-2 Use a Helper

Add the check(stdio, name, condition) helper from this chapter and rewrite your tests from 14-1 to use it, so each check prints a labelled ok line or panics with a clear name.

```
// mathadd.capa
pub fun add(a: Int, b: Int) -> Int
    return a + b
```

```
// tests/testutil.capa, helper partilhado (nao e um teste)
pub fun check(stdio: Stdio, name: String, condition: Bool)
    if condition
        stdio.println("ok: ${name}")
    else
        panic("FAILED: ${name}")
```

```
// tests/test_add.capa, reescrito com o helper check
import mathadd
import testutil

fun main(stdio: Stdio)
    check(stdio, "add(2, 3) == 5", add(2, 3) == 5)
```

```
    check(stdio, "add(-1, 1) == 0", add(-1, 1) == 0)
```

A shared helper lives in tests/testutil.capa, which does not match test_*.capa and so is imported rather than run. check prints ok or panics, making each test a single readable line.

Exercise 14-3 Test the Edges

Write first_or(xs: List<Int>, default: Int) -> Int that returns the first element or the default.

```
// lists.capa
pub fun first_or(xs: List<Int>, default: Int) -> Int
    match xs.first()
        Some(v) -> return v
        None -> return default
```

```
// tests/test_first_or.capa
import lists

fun main(stdio: Stdio)
    let nonempty: List<Int> = [10, 20, 30]
    if first_or(nonempty, -1) != 10
        panic("first_or on non-empty list should be 10")
    let empty: List<Int> = []
    if first_or(empty, -1) != -1
        panic("first_or on empty list should fall back to default")
    stdio.println("first_or tests passed")
```

Testing both a non-empty and an empty list exercises both arms of first_or, so the default path is covered as well as the happy path.

Exercise 14-4 Test a Result

Take the check_age function from Chapter 8 (returning Result<Int, String>) and write tests covering a valid age, a negative age, and an under-18 age, matching on the result each time.

```
// age.capa
pub fun check_age(age: Int) -> Result<Int, String>
    if age < 0
        return Err("age cannot be negative")
    if age < 18
        return Err("must be at least 18")
    return Ok(age)
```

```
// tests/test_check_age.capa
import age

fun main(stdio: Stdio)
    match check_age(25)
        Ok(a) ->
            if a != 25
                panic("valid age should round-trip the value")
        Err(e) -> panic("25 should be valid, got error: ${e}")

    match check_age(-3)
        Ok(_) -> panic("negative age should be rejected")
        Err(e) ->
            if e != "age cannot be negative"
                panic("wrong message for negative age: ${e}")

    match check_age(15)
        Ok(_) -> panic("age under 18 should be rejected")
        Err(e) ->
            if e != "must be at least 18"
                panic("wrong message for under-18: ${e}")

    stdio.println("check_age tests passed")
```

match on the Result asserts both the value on success and the exact message on each failure, covering the valid, negative, and under-18 cases.

Exercise 14-5 Both Backends

Run any of your test files with capa test --both and confirm the output is identical on both backends.

capa test --both runs each test on the Python and Wasm backends and requires exit 0 on both plus byte-identical stdout. Differing output is reported as DIVERGED, the cheapest way to prove a library behaves the same on both backends.

Chapter 15: A Grade-Book Tool

Exercise 15-1 Trim the Name

Real files have stray spaces.

```
    let name = parts.get(0)?.trim()
```

```
import parse

fun main(stdio: Stdio)
    match parse_line("Ana ,17,15")
        Some(s) -> stdio.println("name=[${s.name}]")
        None -> stdio.println("None")
```

trim removes the stray whitespace, so a line like Ana ,17,15 parses the name as Ana. The ? on get(0) unwraps the field or bails out with None.

Exercise 15-2 Reject Negative Scores

A score below zero is surely a mistake.

```
    while i < parts.length()
        let token = parts.get(i)?
        let n = parse_int(token)?
        if n < 0
            return None
        scores.push(n)
        sum = sum + n
        i = i + 1
```

```
import parse

fun main(stdio: Stdio)
    match parse_line("Dan,10,-3,12")
        Some(s) -> stdio.println("parsed ${s.name}")
        None -> stdio.println("rejected")
    match parse_line("Eva,10,3,12")
        Some(s) -> stdio.println("parsed ${s.name}")
        None -> stdio.println("rejected")
```

Rejecting a negative score returns None for the whole line the moment one appears, so a malformed row never becomes a Student. Note that logical negation is not, not the exclamation mark.

Exercise 15-3 Highest and Lowest

Add two fields to Student, best: Int and worst: Int, and compute them in parse_line while you walk the scores.

```
pub type Student {
    name: String,
    scores: List<Int>,
    average: Float,
    passed: Bool,
    best: Int,
    worst: Int
}

pub fun parse_line(line: String) -> Option<Student>
```

```
    let parts = line.split(",")
    if parts.length() < 2
        return None
    let name = parts.get(0)?
    var scores: List<Int> = []
    var sum = 0
    var best = 0
    var worst = 0
    var seen = false
    var i = 1
    while i < parts.length()
        let token = parts.get(i)?
        let n = parse_int(token)?
        scores.push(n)
        sum = sum + n
        if not seen
            best = n
            worst = n
            seen = true
        else
            if n > best
                best = n
            if n < worst
                worst = n
        i = i + 1
    if scores.is_empty()
        return None
    let avg = to_float(sum) / to_float(scores.length())
    return Some(Student { name: name, scores: scores, average: avg,
passed: avg >= 10.0, best: best, worst: worst })
```

```
import parse

fun main(stdio: Stdio)
    match parse_line("Ana,10,20,15")
        Some(s) -> stdio.println("best=${s.best} worst=${s.worst}")
        None -> stdio.println("None")
```

A seen flag initialises best and worst to the first score, then each later score updates them in the same pass, since Capa has no built-in list min or max.

Exercise 15-4 A Configurable Pass Mark

Right now passing is hard-coded at 10.0.

```
pub fun parse_line(line: String, pass_mark: Float) -> Option<Student>
    ...
    return Some(Student { name: name, scores: scores, average: avg,
passed: avg >= pass_mark })
```

```
fun load_students(fs: Fs, path: String, pass_mark: Float) ->
Result<List<Student>, IoError>
    let text = fs.read(path)?
    var students: List<Student> = []
    for line in text.split("\n")
        match parse_line(line, pass_mark)
            Some(s) -> students.push(s)
            None -> ()
    return Ok(students)

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    match load_students(work_dir, "scores.txt", 10.0)
        Err(e) -> stdio.eprintln("read failed: ${e}")
        Ok(students) ->
            let output = render(students)
            match work_dir.write("report.txt", output)
                Err(e) -> stdio.eprintln("write failed: ${e}")
                Ok(_) -> stdio.println(output)
```

Threading pass_mark: Float through parse_line and its caller makes the passing threshold a parameter rather than a hard-coded constant.

Chapter 16: Generating the Report

Exercise 16-1 A Header Line

Begin the report with a title line such as === Grade Report === followed by a blank line, before the per-student lines.

```
pub fun render(students: List<Student>) -> String
    var out = "=== Grade Report ===\n\n"
    for s in students
        let status = if s.passed then "Pass" else "Fail"
        out = out + "${s.name}: ${s.average} (${status})\n"
    let passed = students.filter(fun (s: Student) -> Bool =>
s.passed).length()
    out = out + "\nPassed: ${passed}/${students.length()}\n"
    return out
```

Seeding the output string with a title line and a blank line puts a header above the per-student lines.

Exercise 16-2 Class Average

After the pass count, append a line with the average of all the students' averages.

```
pub fun render(students: List<Student>) -> String
    var out = ""
    for s in students
        let status = if s.passed then "Pass" else "Fail"
        out = out + "${s.name}: ${s.average} (${status})\n"
    let passed = students.filter(fun (s: Student) -> Bool =>
s.passed).length()
    out = out + "\nPassed: ${passed}/${students.length()}\n"
    if not students.is_empty()
        let total = students.fold(0.0, fun (acc: Float, s: Student) -
> Float => acc + s.average)
        let class_avg = total / to_float(students.length())
        out = out + "Class average: ${class_avg}\n"
```

```
    return out
```

fold sums the averages starting from 0.0, and dividing by the count (guarded against an empty list) appends the class average.

Exercise 16-3 Letter Grades

Add a letter to each line based on the average: A for 18+, B for 16+, C for 14+, D for 10+, otherwise F.

```
pub fun grade(avg: Float) -> String
    if avg >= 18.0
        return "A"
    elif avg >= 16.0
        return "B"
    elif avg >= 14.0
        return "C"
    elif avg >= 10.0
        return "D"
    else
        return "F"

pub fun render(students: List<Student>) -> String
    var out = ""
    for s in students
        let status = if s.passed then "Pass" else "Fail"
        let letter = grade(s.average)
        out = out + "${s.name}: ${s.average} (${status},
${letter})\n"
    let passed = students.filter(fun (s: Student) -> Bool =>
s.passed).length()
    out = out + "\nPassed: ${passed}/${students.length()}\n"
    return out
```

A pure helper grade maps an average to a letter with if/elif/else, and render calls it per student so each line carries its grade.

Exercise 16-4 Two Sections

Instead of one list, produce two labelled sections, Passing: and Failing:, by filtering the students twice and looping over each group separately.

```
pub fun render(students: List<Student>) -> String
    let passing = students.filter(fun (s: Student) -> Bool =>
s.passed)
    let failing = students.filter(fun (s: Student) -> Bool => not
s.passed)
    var out = "Passing:\n"
    for s in passing
        out = out + "  ${s.name}: ${s.average}\n"
    out = out + "Failing:\n"
    for s in failing
        out = out + "  ${s.name}: ${s.average}\n"
    out = out + "\nPassed:
${passing.length()}/${students.length()}\n"
    return out
```

Filtering twice, once for passing and once for failing, produces two labelled sections from the same list.

Exercise 16-5 Round to Two Places

Write fun round2(x: Float) -> Float that returns x rounded to two decimals (hint: to_float(to_int(x * 100.0 + 0.5)) / 100.0), and use it on the average in render so Bruno shows as 9.33.

```
pub fun round2(x: Float) -> Float
    return to_float(to_int(x * 100.0 + 0.5)) / 100.0

pub fun render(students: List<Student>) -> String
    var out = ""
    for s in students
        let status = if s.passed then "Pass" else "Fail"
        out = out + "${s.name}: ${round2(s.average)} (${status})\n"
    let passed = students.filter(fun (s: Student) -> Bool =>
s.passed).length()
```

```
    out = out + "\nPassed: ${passed}/${students.length()}\n"
    return out
```

round2 scales by 100, rounds via to_int, and scales back, so an average like 9.3333 prints as 9.33.

Chapter 17: Files, Capabilities, and Robustness

Exercise 17-1 Filenames from Arguments

Use the Env capability's args() to let the user pass the input and output filenames on the command line (for example capa --run main.capa grades.txt out.txt), falling back to the defaults when no argu

```
fun main(fs: Fs, stdio: Stdio, env: Env)
    let args = env.args()
    let in_path = args.get(0).unwrap_or("scores.txt")
    let out_path = args.get(1).unwrap_or("report.txt")
    let work_dir = fs.restrict_to("./")
    match load_students(work_dir, in_path)
        Err(e) -> stdio.eprintln("read failed: ${e}")
        Ok(students) ->
            let output = render(students)
            match work_dir.write(out_path, output)
                Err(e) -> stdio.eprintln("write failed: ${e}")
                Ok(_) -> stdio.println(output)
```

env.args() supplies the input and output filenames, and unwrap_or fills in defaults. Program arguments are passed after -- on the command line.

Exercise 17-2 Count the Skips

Have load_students also report how many lines were skipped as malformed, and print that count after the summary, so the user knows if part of their file was ignored.

```
fun load_students(fs: Fs, path: String) -> Result<(List<Student>,
Int), IoError>
    let text = fs.read(path)?
    var students: List<Student> = []
    var skipped = 0
    for line in text.split("\n")
        match parse_line(line)
            Some(s) -> students.push(s)
            None ->
                if not line.trim().is_empty()
                    skipped = skipped + 1
    return Ok((students, skipped))

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    match load_students(work_dir, "scores.txt")
        Err(e) -> stdio.eprintln("read failed: ${e}")
        Ok(loaded) ->
            let (students, skipped) = loaded
            let output = render(students)
            match work_dir.write("report.txt", output)
                Err(e) -> stdio.eprintln("write failed: ${e}")
                Ok(_) ->
                    stdio.println(output)
                    stdio.println("Skipped: ${skipped}")
```

Returning a tuple (List<Student>, Int) lets load_students report how many non-empty lines failed to parse, printed after the summary.

Exercise 17-3 Empty File

Run the tool against an empty scores.txt.

```
fun main(fs: Fs, stdio: Stdio)
```

```
    let work_dir = fs.restrict_to("./")
    match load_students(work_dir, "scores.txt")
        Err(e) -> stdio.eprintln("read failed: ${e}")
        Ok(students) ->
            if students.is_empty()
                stdio.println("no students found")
            else
                let output = render(students)
                match work_dir.write("report.txt", output)
                    Err(e) -> stdio.eprintln("write failed: ${e}")
                    Ok(_) -> stdio.println(output)
```

Checking is_empty before rendering lets the tool print no students found instead of an empty report.

Exercise 17-4 Tighter Narrowing

Instead of restrict_to("./"), create a data/ folder, put scores.txt there, and narrow the Fs to "./data/".

```
fun main(fs: Fs, stdio: Stdio)
    let data_dir = fs.restrict_to("./data/")
    match load_students(data_dir, "data/scores.txt")
        Err(e) -> stdio.eprintln("read failed: ${e}")
        Ok(students) ->
            let output = render(students)
            match data_dir.write("data/report.txt", output)
                Err(e) -> stdio.eprintln("write failed: ${e}")
                Ok(_) -> stdio.println(output)
```

Narrowing the Fs to ./data/ still works because paths resolve against the working directory and are then validated against the prefix, so they must include the data/ segment.

Exercise 17-5 End-to-End Test

Write a test that writes a small scores.txt through a narrowed Fs, runs load_students and render on it, and checks the report, an end-to-end test that exercises all three files together.

```
import parse
import report

fun load_students(fs: Fs, path: String) -> Result<List<Student>,
IoError>
    let text = fs.read(path)?
    var students: List<Student> = []
    for line in text.split("\n")
        match parse_line(line)
            Some(s) -> students.push(s)
            None -> ()
    return Ok(students)

fun assert(cond: Bool, msg: String)
    if not cond
        panic("assertion failed: ${msg}")

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    // 1. escreve a fixtura atraves do Fs estreitado
    match work_dir.write("e2e_scores.txt",
"Ana,17,15,19\nBruno,8,11,9\n")
        Err(e) -> panic("setup write failed: ${e}")
        Ok(_) -> ()
    // 2. carrega + renderiza
    match load_students(work_dir, "e2e_scores.txt")
        Err(e) -> panic("load failed: ${e}")
        Ok(students) ->
            assert(students.length() == 2, "expected 2 students")
            let out = render(students)
            assert(out.contains("Ana: 17.0 (Pass)"), "Ana line
present")
            assert(out.contains("Bruno: 9.333333333333334 (Fail)"),
"Bruno line present")
            assert(out.contains("Passed: 1/2"), "summary present")
            stdio.println("all assertions passed")
```

An end-to-end test writes a fixture through a narrowed Fs, runs the load-and-render pipeline, and asserts on the report with a panic-on-failure helper.

Chapter 18: Reading an SBOM

Exercise 18-1 Count Components

Write a small program that parses the three-component example from the brief and prints how many components it found and the name of each, one per line.

```
import sbom

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let text = match work_dir.read("sbom.json")
        Ok(s) -> s
        Err(e) ->
            stdio.eprintln("read failed: ${e}")
            return
    match parse_sbom(text)
        Err(e) -> stdio.eprintln("parse failed: ${e}")
        Ok(comps) ->
            stdio.println("Found ${comps.length()} component(s):")
            for c in comps
                stdio.println("  ${c.name}")
```

Parsing the example and looping over the components prints the count and each name; parse_sbom returns a Result, handled with match.

Exercise 18-2 Total Capabilities

Add a function that takes a List<Component> and returns the total number of capability entries across all components (use fold, or a loop with a counter).

```
import sbom

fun total_capabilities(components: List<Component>) -> Int
    return components.fold(0, fun (acc: Int, c: Component) -> Int =>
acc + c.capabilities.length())

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let text = match work_dir.read("sbom.json")
        Ok(s) -> s
        Err(e) ->
            stdio.eprintln("read failed: ${e}")
            return
    match parse_sbom(text)
        Err(e) -> stdio.eprintln("parse failed: ${e}")
        Ok(comps) -> stdio.println("Total capabilities:
${total_capabilities(comps)}")
```

fold accumulates each component's capability count, so the three components total 3 capability entries.

Exercise 18-3 Malformed Entry

Feed parse_sbom a document where one component is missing its version field.

```
import sbom

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let text = match work_dir.read("sbom_bad.json")
        Ok(s) -> s
        Err(e) ->
            stdio.eprintln("read failed: ${e}")
            return
    match parse_sbom(text)
        Err(e) -> stdio.eprintln("parse failed: ${e}")
        Ok(comps) ->
            stdio.println("Parsed ${comps.length()} valid
component(s):")
```

```
            for c in comps
                stdio.println("  ${c.name} @ ${c.version}")
```

No code change is needed: parse_component already uses ? on every field, so a component missing version parses to None and parse_sbom skips it while the good entries survive.

Exercise 18-4 Bad JSON

Pass parse_sbom a string that is not valid JSON at all, and confirm it returns an Err with a message rather than crashing.

```
import sbom

fun main(stdio: Stdio)
    let garbage = "this is not json {{{"
    match parse_sbom(garbage)
        Err(e) -> stdio.println("rejected as expected: ${e}")
        Ok(comps) -> stdio.println("unexpectedly parsed
${comps.length()} component(s)")
```

parse_sbom calls parse_json with ? on its first line, so invalid input returns Err with a message instead of crashing: the parse error is a value, not an exception.

Chapter 19: Checking a Policy

Exercise 19-1 Count Clean Components

Add a function that returns how many components had no violations, and include that number in the rendered verdict (for example 2 of 3 components clean).

```
import sbom
import policy

fun count_clean(components: List<Component>, allowed: Set<String>) ->
Int
    var clean = 0
    for c in components
        if check_component(c, allowed).is_empty()
            clean = clean + 1
    return clean

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let allowed = make_allowed(["Stdio", "Fs"])
    let text = match work_dir.read("sbom.json")
        Ok(s) -> s
        Err(e) ->
            stdio.eprintln("read failed: ${e}")
            return
    match parse_sbom(text)
        Err(e) -> stdio.eprintln("parse failed: ${e}")
        Ok(comps) ->
            let findings = audit(comps, allowed)
            stdio.print(render_findings(findings))
            let clean = count_clean(comps, allowed)
            stdio.println("${clean} of ${comps.length()} components
clean")
```

Reusing check_component, a component is clean when its findings list is empty; counting the clean ones feeds the verdict line.

Exercise 19-2 A Severity

Add a severity: String field to Finding.

```
pub type Finding {
    component: String,
    capability_name: String,
    severity: String
}
```

```
fun severity_of(cap: String) -> String
    if cap == "Unsafe" or cap == "Net"
        return "high"
    return "low"

pub fun check_component(c: Component, allowed: Set<String>) ->
List<Finding>
    var findings: List<Finding> = []
    for cap in c.capabilities
        if not allowed.contains(cap)
            findings.push(Finding { component: c.name,
capability_name: cap, severity: severity_of(cap) })
    return findings

pub fun render_findings(findings: List<Finding>) -> String
    if findings.is_empty()
        return "OK: all components satisfy the policy\n"
    var out = "POLICY VIOLATIONS:\n"
    for f in findings
        out = out + "  [${f.severity}] ${f.component} holds
disallowed ${f.capability_name}\n"
    out = out + "\n${findings.length()} violation(s) found\n"
    return out
```

A severity field on Finding, set by a small severity_of helper, marks Net and Unsafe high and everything else low, shown on each reported line.

Exercise 19-3 Per-Component Exceptions

Sometimes one component is legitimately allowed an extra capability.

```
pub type Exception {
    component: String,
    capability_name: String
}

fun is_excepted(f: Finding, exceptions: List<Exception>) -> Bool
    for e in exceptions
```

```
        if e.component == f.component and e.capability_name ==
f.capability_name
            return true
    return false

pub fun audit(components: List<Component>, allowed: Set<String>,
exceptions: List<Exception>) -> List<Finding>
    var all: List<Finding> = []
    for c in components
        for f in check_component(c, allowed)
            if not is_excepted(f, exceptions)
                all.push(f)
    return all
```

```
fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let allowed = make_allowed(["Stdio", "Fs"])
    // capa_http esta autorizado a manter Net (aprovado pela
seguranca); logger nao.
    let exceptions = [Exception { component: "capa_http",
capability_name: "Net" }]
    let text = match work_dir.read("sbom.json")
        Ok(s) -> s
        Err(e) ->
            stdio.eprintln("read failed: ${e}")
            return
    match parse_sbom(text)
        Err(e) -> stdio.eprintln("parse failed: ${e}")
        Ok(comps) -> stdio.print(render_findings(audit(comps,
allowed, exceptions)))
```

An exceptions list of (component, capability) pairs and an is_excepted predicate let audit skip approved findings; the exception is by exact pair, so other components keep their findings.

Exercise 19-4 Denylist Instead

Write an alternative audit_deny that takes a denylist, the capabilities that are forbidden, and flags any component holding one of them.

```
import sbom
import policy

// Modelo denylist: levanta um Finding para cada capability que
aparece na
// denylist, independentemente do resto. Contrasta com `audit`, que
marca
// tudo o que NAO esta numa allowlist.
fun audit_deny(components: List<Component>, denied: Set<String>) ->
List<Finding>
    var all: List<Finding> = []
    for c in components
        for cap in c.capabilities
            if denied.contains(cap)
                all.push(Finding { component: c.name,
capability_name: cap })
    return all

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let denied = make_allowed(["Net"])
    let text = match work_dir.read("sbom.json")
        Ok(s) -> s
        Err(e) ->
            stdio.eprintln("read failed: ${e}")
            return
    match parse_sbom(text)
        Err(e) -> stdio.eprintln("parse failed: ${e}")
        Ok(comps) -> stdio.print(render_findings(audit_deny(comps,
denied)))
```

A denylist marks any listed capability instead of anything missing from an allowlist. The allowlist is safer by default: a new, unforeseen capability is denied until approved, which is the right posture for supply-chain threats.

Chapter 20: A CI-Ready Tool

Exercise 20-1 Policy from a File

Instead of hard-coding the allowlist, read it from a policy.txt file (one capability per line) using the narrowed Fs, and build the Set from those lines.

```
import sbom
import policy

fun load_policy(fs: Fs, path: String) -> Result<Set<String>, String>
    let text = fs.read(path).map_err(fun (e: IoError) -> String =>
"read failed: ${e}")?
    var names: List<String> = []
    for line in text.split("\n")
        let cap = line.trim()
        if not cap.is_empty()
            names.push(cap)
    return Ok(make_allowed(names))

fun run(fs: Fs, sbom_path: String, allowed: Set<String>) ->
Result<List<Finding>, String>
    let text = fs.read(sbom_path).map_err(fun (e: IoError) -> String
=> "read failed: ${e}")?
    let comps = parse_sbom(text)?
    return Ok(audit(comps, allowed))

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let outcome = load_policy(work_dir, "policy.txt").and_then(fun
(allowed: Set<String>) -> Result<List<Finding>, String> =>
run(work_dir, "sbom.json", allowed))
    match outcome
        Err(e) -> stdio.eprintln("error: ${e}")
        Ok(findings) -> stdio.print(render_findings(findings))
```

load_policy reads policy.txt through the same narrowed Fs, one capability per line, and builds the allowed set, so the policy is read under the same trust boundary as the SBOM.

Exercise 20-2 Filenames from Arguments

Use Env's args() to accept the SBOM path on the command line, defaulting to sbom.json.

```
import sbom
import policy

fun run(fs: Fs, path: String, allowed: Set<String>) ->
Result<List<Finding>, String>
    let text = fs.read(path).map_err(fun (e: IoError) -> String =>
"read failed: ${e}")?
    let comps = parse_sbom(text)?
    return Ok(audit(comps, allowed))

fun main(fs: Fs, stdio: Stdio, env: Env)
    let work_dir = fs.restrict_to("./")
    let allowed = make_allowed(["Stdio", "Fs"])
    let path = match env.args().first()
        Some(p) -> p
        None -> "sbom.json"
    stdio.println("auditing ${path}")
    match run(work_dir, path, allowed)
        Err(e) -> stdio.eprintln("error: ${e}")
        Ok(findings) -> stdio.print(render_findings(findings))
```

env.args().first() supplies the SBOM path with a default of sbom.json; arguments follow -- on the command line.

Exercise 20-3 A Summary Line

Before the verdict, print a one-line summary such as Audited 3 components against 2 allowed capabilities.

```
import sbom
import policy

fun run(fs: Fs, path: String, allowed: Set<String>) ->
Result<List<Component>, String>
```

```
    let text = fs.read(path).map_err(fun (e: IoError) -> String =>
"read failed: ${e}")?
    return parse_sbom(text)

fun main(fs: Fs, stdio: Stdio)
    let work_dir = fs.restrict_to("./")
    let allowed = make_allowed(["Stdio", "Fs"])
    match run(work_dir, "sbom.json", allowed)
        Err(e) -> stdio.eprintln("error: ${e}")
        Ok(comps) ->
            stdio.println("Audited ${comps.length()} components
against ${allowed.length()} allowed capabilities.")
            stdio.print(render_findings(audit(comps, allowed)))
```

comps.length() and allowed.length() feed a one-line summary printed before the verdict; run returns the components so main can both count and audit them.

Exercise 20-4 Warn-Only Mode

Add a mode that prints the violations but does not panic, so the tool can be run as an advisory report rather than a hard gate.

```
import sbom
import policy

fun run(fs: Fs, path: String, allowed: Set<String>) ->
Result<List<Finding>, String>
    let text = fs.read(path).map_err(fun (e: IoError) -> String =>
"read failed: ${e}")?
    let comps = parse_sbom(text)?
    return Ok(audit(comps, allowed))

// O chamador escolhe o modo com `--warn` depois de `--`.
// Modo gate (default): as violacoes causam exit nao-zero (panic),
// apropriado para CI, onde um SBOM mau tem de bloquear o build.
// Modo warn-only: as violacoes sao impressas mas o programa sai com
0,
// apropriado para um relatorio consultivo ou um periodo de migracao
em
```

```
// que se quer visibilidade sem ainda partir o pipeline.
fun main(fs: Fs, stdio: Stdio, env: Env)
    let work_dir = fs.restrict_to("./")
    let allowed = make_allowed(["Stdio", "Fs"])
    let warn_only = env.args().contains("--warn")
    match run(work_dir, "sbom.json", allowed)
        Err(e) ->
            stdio.eprintln("error: ${e}")
            panic("audit could not run")
        Ok(findings) ->
            stdio.print(render_findings(findings))
            if not findings.is_empty()
                if warn_only
                    stdio.println("(warn-only: not failing the
build)")
                else
                    panic("${findings.length()} policy violation(s)")
```

Selecting --warn after -- makes violations print without panicking. Gate mode (the default) fails the build, right for CI; warn-only suits a migration period where you want visibility without blocking yet.

Chapter 21: Handling Secrets

Exercise 21-1 Provoke the Leak

Write a function that takes a Payment and a Stdio and tries to print the full card number.

```
import payment

@strict_ifc()
fun print_card(stdio: Stdio, p: Payment) -> Unit
    stdio.println("Card: ${p.card}")

fun main(stdio: Stdio)
    let p = Payment { card: "4111111111111234", amount: 4200 }
    print_card(stdio, p)
```

Printing the full card is an information-flow violation: an error under @strict_ifc(), a warning without it. Printing the already-declassified receipt line compiles cleanly.

Exercise 21-2 A Secret Total

Add a @secret String cvv field to Payment.

```
pub type Payment {
    card: @secret String,
    cvv: @secret String,
    amount: Int
}

// mask_card e receipt_line ficam iguais

pub fun amount_of(p: Payment) -> Int
    return p.amount
```

```
import payment

@strict_ifc()
fun show_amount(stdio: Stdio, p: Payment) -> Unit
    stdio.println("Amount: ${amount_of(p)}")

fun main(stdio: Stdio)
    let p = Payment { card: "4111111111111234", cvv: "123", amount:
4200 }
    show_amount(stdio, p)
```

A debug string mixing card and cvv is @secret, because one secret field taints the whole interpolated string; the violation points at the println. Returning only the public amount is allowed.

Exercise 21-3 Honest Reasons

Write two declassify calls with genuinely different reasons (for example one for a printed receipt, one for an internal log) and note how each would appear as a separate entry in the disclosure record.

```
pub fun audit_line(p: Payment) -> String
    let last4 = declassify(mask_card(p.card), reason: "internal fraud
audit: last 4 retained for reconciliation")
    return "AUDIT card=${last4} amount=${p.amount}"
```

```
import payment

fun main(stdio: Stdio)
    let p = Payment { card: "4111111111111234", amount: 4200 }
    stdio.println(receipt_line(p))
    stdio.println(audit_line(p))
```

Two declassify calls with genuinely different reasons appear as separate entries in the disclosure record. Forcing a reason is what makes each leak auditable after the fact, not mere bureaucracy.

Exercise 21-4 No Laundering

Try to hide the card by putting it in a one-element List<String> and pulling it back out, then printing it.

```
import payment

@strict_ifc()
fun try_launder(stdio: Stdio, p: Payment) -> Unit
    let box: List<String> = [p.card]
    let leaked = box.get(0).unwrap_or("")
    stdio.println("Card: ${leaked}")

fun main(stdio: Stdio)
    let p = Payment { card: "4111111111111234", amount: 4200 }
```

```
    try_launder(stdio, p)
```

Boxing the card in a one-element list and reading it back does not launder it: the value that comes out of get is still @secret, so it cannot be smuggled past the checker.

Chapter 22: A Vault Capability

Exercise 22-1 Try to Cheat

Write a version of charge that attempts to record the raw p.card, and confirm the compiler rejects it.

```
@strict_ifc()
pub fun charge(vault: AuditVault, p: Payment) -> Result<Unit,
IoError>
    return vault.record("card: ${p.card}")
```

Recording the raw p.card is flagged: a warning without @strict_ifc(), an error with it. The check is interprocedural, naming AuditVault.record because the leak reaches a public sink inside it. The fix records the declassified receipt line.

Exercise 22-2 Timestamped Entries

Give record a richer entry by prepending a counter or label in charge (for example Entry #1: ...).

```
pub fun charge(vault: AuditVault, p: Payment, n: Int) -> Result<Unit,
IoError>
    let line = "Entry #${n}: " + receipt_line(p)
    return vault.record(line)
```

```
import payment
import vault

fun main(fs: Fs, stdio: Stdio)
    let ledger_dir = fs.restrict_to("./ledger/")
    let v = make_vault(ledger_dir, "./ledger/audit.log")
    let p = Payment { card: "4111111111111234", amount: 4200 }
    match charge(v, p, 1)
        Ok(_) -> stdio.println("done")
        Err(e) -> stdio.eprintln("failed: ${e}")
```

Prefixing a public counter and label to the already-declassified receipt line keeps the whole entry public, so no information-flow warning fires.

Exercise 22-3 A Second Implementor

Write a MemoryVault that implements AuditVault but stores entries in an in-memory List<String> instead of a file (useful for tests).

```
pub type MemoryVault {
    entries: List<String>
}

impl AuditVault for MemoryVault
    fun record(self, entry: String) -> Result<Unit, IoError>
        self.entries.push(entry)
        return Ok(())

pub fun make_memory_vault() -> MemoryVault
    let entries: List<String> = []
    return MemoryVault { entries: entries }
```

```
import payment
import vault

fun main(stdio: Stdio)
```

```
    let mv = make_memory_vault()
    let p = Payment { card: "4111111111111234", amount: 4200 }
    match charge(mv, p)
        Ok(_) ->
            let count = mv.entries.length()
            let first = mv.entries.get(0).unwrap_or("?")
            stdio.println("recorded ${count} entries: ${first}")
        Err(e) -> stdio.eprintln("failed: ${e}")
```

A MemoryVault implements the same AuditVault contract but stores entries in a list, ideal for tests. charge does not change, because it depends on the contract, not the concrete vault.

Exercise 22-4 Narrow the Vault's Fs

In a factory or in main, narrow the Fs with restrict_to to the ledger's directory before building the FileVault, so even the vault's own filesystem reach is the smallest it can be.

```
pub fun make_vault_in(fs: Fs, dir: String) -> FileVault
    let scoped = fs.restrict_to(dir)
    return FileVault { path: dir + "audit.log", fs: scoped }
```

```
import payment
import vault

fun main(fs: Fs, stdio: Stdio)
    let v = make_vault_in(fs, "./ledger/")
    let p = Payment { card: "4111111111111234", amount: 4200 }
    match charge(v, p)
        Ok(_) -> stdio.println("done")
        Err(e) -> stdio.eprintln("failed: ${e}")
```

Moving restrict_to into a factory means callers never have to remember it: the FileVault holds an already-narrowed Fs, so its authority is exactly write to the ledger directory.

Chapter 23: The Service, Proven Safe

Exercise 23-1 Many Payments

Process a List<Payment> in a loop, charging each and collecting the receipts.

```
import payment
import vault

@strict_ifc()
fun run_all(stdio: Stdio, v: AuditVault, payments: List<Payment>) ->
Result<Unit, IoError>
    for p in payments
        charge(v, p)?
        stdio.println(receipt_line(p))
    return Ok(())

fun main(fs: Fs, stdio: Stdio)
    let ledger_dir = fs.restrict_to("./ledger/")
    let v = make_vault(ledger_dir, "./ledger/audit.log")
    let payments: List<Payment> = [
        Payment { card: "4111111111111234", amount: 4200 },
        Payment { card: "5500000000005678", amount: 999 },
        Payment { card: "340000000009012", amount: 12000 }
    ]
    match run_all(stdio, v, payments)
        Ok(_) -> stdio.println("all done")
        Err(e) -> stdio.eprintln("failed: ${e}")
```

The loop charges each payment and prints its declassified receipt line, so the screen never shows a full card and the ledger ends with one masked line per payment. Under @strict_ifc() a stray ${p.card} would still fail the build.

Exercise 23-2 Keep It Strict

With @strict_ifc() on run, deliberately add a line that prints p.card and confirm the build fails.

```
@strict_ifc()
fun run(stdio: Stdio, v: AuditVault, p: Payment) -> Result<Unit,
IoError>
    charge(v, p)?
    stdio.println(receipt_line(p))
    stdio.println("debug card: ${p.card}")
    return Ok(())
```

With @strict_ifc() on run, a line that prints p.card is a build error; remove the attribute and the same leak is only a warning. The attribute is the switch that decides whether a leak stops the build.

Exercise 23-3 Read Back the Ledger

Add a function that reads ledger/audit.log through the narrowed Fs and prints how many entries it contains, a simple end-of-day report.

```
import payment
import vault

pub fun count_entries(fs: Fs, path: String) -> Int
    let text = fs.read(path).unwrap_or("")
    if text == ""
        return 0
    var count = 0
    for line in text.split("\n")
        if line != ""
            count = count + 1
    return count

fun main(fs: Fs, stdio: Stdio)
    let ledger_dir = fs.restrict_to("./ledger/")
    let n = count_entries(ledger_dir, "./ledger/audit.log")
    stdio.println("ledger has ${n} entries")
```

count_entries reads ledger/audit.log through the narrowed Fs. Everything written went through the declassified receipt line, so what is read back is masked: the masking happened on write and cannot be undone on read.

Exercise 23-4 Inspect the Sites

Add a second, differently-reasoned declassify somewhere (for example a separate internal log line), run capa --manifest, and confirm both disclosure sites appear with their distinct reasons.

A second, differently-reasoned declassify shows up under capa --manifest. Each function carries its own declassifications array of reason, value, and pos, and summary.declassification_sites is a count, not a global array.

Capa

THE CAPABILITY-TYPED PROGRAMMING LANGUAGE

Make authority visible.

Most languages let any line of code do anything, open your files, read your secrets, reach the network, with nothing in sight to say so. Capa is different. Every function must declare the authority it holds, and the compiler checks it.

This book teaches the language from absolute zero. If you have never written a line of code, you are in exactly the right place.

Inside, you will:

- Learn values, types, collections, control flow, functions and error handling
- Meet the ideas that make Capa unique: capabilities, attenuation and information flow
- Handle errors as values and prove where your secrets are allowed to go
- Build three real projects, from a CLI tool to a service the compiler proves safe

---

About the author

Nelson Duarte teaches computer science, programming in Python and C#, cybersecurity, networks, and Linux and Windows systems, and is the creator of the Capa language. He wrote this book in the same patient, example-driven style he uses in the classroom.

capa-language.com

Nelson Duarte · MIT or Apache-2.0