kaikai

Simple programming with kaikai

A language for humans and agents

Eduardo Díaz

English edition · Draft

This book ships in two editions: Spanish and English. Both are first-class citizens; neither is a translation of the other. This is the English edition.

Typography: New Computer Modern (body), Helvetica Neue (headings), JetBrains Mono (code). Set with Typst.

Contents

Prologue

> *A* kaikai *is a single thread that draws a figure between the hands. In Rapa Nui culture, this corresponds to the whole practice: the thread, the figure, and the* pata'u ta'u *— the chant that goes with it.*
>
> *This book borrows that image.* Simple *comes from Latin* simplex*: "one fold, one thread."* Complex *comes from* complectere*: "braided together." The book's argument, and the language's bet, is that simplicity is not the opposite of hard, it is the opposite of entangled.*
>
> *The etymology and the simple/easy distinction come from Rich Hickey in his talk* Simple Made Easy *(Strange Loop, 2011), available at InfoQ.*
>
> *Hickey's thesis runs through this whole book: simple is an objective property of things (how many threads compose them); easy is a property relative to the observer (how close it is to what they already know). Kaikai bets on the first even when it costs the second.*

I've wanted to write a programming language since I learned how to build compilers in college. Once I got into industry I ended up solving more than a few problems by inventing small languages for them — what we now call domain-specific languages, or DSLs.

Language design fascinates me. I love learning new languages, comparing them, asking why their authors made the choices they made. Along the way I also figured out why we have so many programming languages: I gave a talk about it years ago, in Spanish, on YouTube. In that same talk I introduced **Ogú**, a language I designed and built years ago.

Ogú is a cartoon character — a caveman, friend of the kid Mampato, created by the Chilean illustrator and cartoonist Themo Lobos. Getting permission from the family to use the character as the language's mascot is still one of the gestures I'm proudest of. The repository goes back to 2010 or so. I announced intentions on the blog, wrote parsers, started over, wrote a grammar, changed it, read and re-read the classic textbooks (the dragon book, *Modern Compiler Implementation*, *Engineering a Compiler*), and made very little progress.

In 2017 I made a serious push: across two months, with sixty hours total split between work and late nights, I built a first version backed by Clojure. Ogú is essentially a transpiler — the compiler translates Ogú syntax into *S-expressions* that Clojure interprets, and the

JVM does the rest. A *"fake it 'til you make it"* solid enough to demo to my conference audience.

But Ogú ended up abandoned. The last commit is from 2021. The GitHub organization is still there for anyone curious. I rewrote the parser in Scala — an exercise in persistence and mild masochism. My problem was code generation, but it was also my ambition.

A friend told me that if I was going to build a language, I'd better answer what was *new* about it.

I learned new languages, the functional family in particular — Haskell, F#. I fell in love with Rust, fought the borrow checker, ended up fluent in lifetime annotations and the promise of memory management without garbage collection.

I also studied category theory and had an epiphany about it, which I wrote up at the time in the post *Revelaciones* (2015, in Spanish). That's where I discovered monads. And right when I was deep in that batter, a post on algebraic effects landed in my hands, alongside Bob Nystrom's essay *"What Color is Your Function?"* I had a second epiphany.

That's how the idea for a new language was born. I called it kaikai.

I initially had in mind the Kai Kai serpent from Mapuche mythology, but I later discovered that in Rapa Nui culture, *kai kai* refers to a game in which string figures are woven with the fingers while a *pata'u ta'u* is chanted — a recited verse. In kai kai, structure and narration go hand in hand, much like a well-typed program.

What sets kaikai apart from earlier attempts, what makes it original, is in this book: I'll let you find it for yourself. What I do want to say here has to do with a central fact in kaikai's making: the use of AI.

The compiler this book describes was built in a month. The way Ogú was built in 60 hours, kaikai was built with the help of Claude Code in roughly a month. Something that could have taken years.

My reasoning was this: the AI has read and understood, better than I have, every paper on algebraic effects, functional programming, language design, and the rest. I might as well use that to my advantage.

I acted as the architect — much of the design already existed in my head — and the AI helped me put that design on disk.

Along the way I invented a method for working with AI in language development at record speed. I documented it under the name ELP: *Empirical Lane Parallelism*. This way of using agents to amplify your development process and build robust software in little time lives in github.com/lnds/elp. I invite you to read it if you want to understand how a compiler as involved as kaikai's could come together in a single month.

A note on how this book was written

This book was written with the assistance of an AI agent — Claude Code from Anthropic — under my direction as author and editor. I decided which chapters existed, in what order, with what voice, with what emphasis. Claude drafted, I edited, corrected, sent back,

edited again. We validated the examples against the compiler. We checked technical claims against the language documentation. We calibrated the book's voice against my blog.

I'm saying this not as an apology but as an acknowledgment of the reality software engineers live in in 2026. The AI wrote parts of it; my judgment decided everything. In *The End of Artisanal Software* I wrote that Jacquard looms are already among us and that the question is not whether we'll use them but how. This book is one concrete answer to that question: what AI enables is not doing without the author, it's writing books one wouldn't have dared start. Without that support this text would still be in my head, the way the sketch of Ogú stayed in my head for years.

If you find inconsistencies, examples that don't compile, outdated claims, or any other slip: the lapse is mine, not the agent's. The book lives at `kaikailang-org/kaikai-book` and reports are welcome as issues or pull requests.

Conventions

A few paragraphs on how the book is organized and how to read it. None of this is hard, but knowing it upfront saves friction.

Structure

The book is organized into four parts, eighteen chapters and six appendices.

- **Part I (chs. 1–2)** is the landing strip: a tour of the language with runnable programs and a short chapter on how to *think* in kaikai.
- **Part II (chs. 3–11)** covers the core: types, functions, modules, protocols, units of measure, contracts. What you need to read and write everyday kaikai.
- **Part III (chs. 12–15)** gets into what's distinctive: algebraic effects, fiber-based concurrency, actors, typed holes with AI assistance.
- **Part IV (chs. 16–18)** closes with tooling and two case studies.
- **Appendices A–F** stand as reference: compiler bootstrap, Perceus, operators, stdlib effect catalog, glossary, further reading.

If you come from a functional background and only the new material interests you, start with Part III and circle back to the core as needed.

Shape of each chapter

Every chapter opens with a paragraph or two of context: why the topic matters, what problem it solves. Then comes the technical body — dense, with examples. Key chapters close with a **case study** that pulls the concepts together in a realistic program. At the end you'll find numbered **exercises**, between three and eight depending on the chapter's weight.

Numbering and references

- **Chapters** are integers (ch. 7, ch. 12).
- **Sections** carry the chapter number (§7.3, §12.10).
- **Exercises** are cited as *7.3* inside the chapter, or *ch. 7, exercise 3* from elsewhere.
- **Appendices** are letters (appendix A, appendix D), with sections of the form §A.1.

Typography

- **Bold** introduces a new term the first time it appears. If a word is bold, that's the definition.
- *Italics* are for emphasis and the titles of cited works (*The Go Programming Language*, *Learn You a Haskell*).
- `Monospaced text` is for identifiers, inline code, file names and commands.

Code

Every block tagged `kai` is runnable. Copy it into a `.kai` file, compile with `kai run`, and you'll get the output the text promises. If a block carries *no* language tag, it's a terminal session: what comes after the `$` is what you type, what comes below is the output.

```
$ kai run examples/ch01/01_hello.kai
Hello, kaikai
```

Longer examples live under `examples/chNN/` in the book's repository, and the text references them by name when it's worth pulling the whole file.

Notation

The **language** is called `kaikai`, always lowercase, even at the start of a sentence. The **command-line tool** is `kai`: use it to compile (`kai run`), to run tests (`kai test`), to check properties (`kai check`), to measure (`kai bench`).

Language identifiers, keywords, compiler messages and file names stay in English in both editions. Technical words that are already part of the profession's vocabulary — *handler*, *fiber*, *effect row*, *pattern matching* — are used as is, without italics.

Language editions

The book is published in two editions: Spanish and English. Both live in the same repository, in parallel trees. This is the English edition. The Spanish edition is not a translation: the two were written in parallel, with the same structure and the same examples, each in its own native voice.

Living software

The compiler, the stdlib and this book are evolving. Versions move, examples occasionally stop compiling between releases, appendices fall out of date. If you find a discrepancy between the book and the compiler you have installed, please file an issue against the book's repo. The book records up front the compiler version each edition was validated against.

Who should read this book

Programmers with experience in some language — Python, JavaScript, Go, Java, C#, Rust, anything — who are curious about a new one. I don't assume a functional background: if you've never touched Haskell, OCaml or Elixir, the introductory chapters

carry you through. If you come from the functional world, you can skip to Part III and read what's distinctive about kaikai head-on.

This is not a book for absolute beginners. I assume you know what a function is, what a list is, what a type is, what a test is.

Thanks

To everyone who's read my blog for over twenty years: this book exists because that conversation existed. The constancy of those readers — the comments, the emails, the corrections, the discussions that ran first on Twitter and later in newsletters — kept reassuring me that writing was worth it. Without that patient audience, kaikai would have stayed in my head alongside Ogú.

To Themo Lobos, who is no longer with us, for Ogú the caveman and for giving three generations of Chileans the conviction that imagined worlds are built by hand. I recommend his graphic novel *Mata-ki-te-rangi* (which became the first Chilean animated feature film: *Ogú y Mampato en Rapa Nui*). To the authors whose ideas kaikai picks up: Daan Leijen for Koka, Andreas Rossberg and Jonathan Brachthäuser for Effekt, Joe Armstrong for the BEAM spirit, and the academic community that carried algebraic effects from Plotkin and Pretnar to a usable tool. To the friends who used to tease me about Ogú and now ask about kaikai.

And to Anthropic, for building a tool that let me write the compiler and this book in a month and not in the next decade.

The compiler is alive, the language is evolving, and the community is small but attentive. There's room for more.

Chapter 1 · A Tour of kaikai

The best way to get to know a language is to read it and run it. This chapter is a guided tour of kaikai in eight short programs. None of them runs longer than thirty lines, every one of them compiles, and together they cover the shapes you will see again and again in the rest of the book: declarations, algebraic data types, pattern matching, effects, fibers, protocols, units of measure, and inline tests.

We will not explain every detail yet. The point is to leave you with a view of the language from above, and the sense that you can already read kaikai code even when some of the corners are still blurry. The corners come into focus in the chapters that follow.

If you want to follow along on your own machine, the source files for this chapter live under `examples/ch01/` in the book repository. Installation of `kai` is covered at the end of the chapter, in §1.9 — if you need it now, jump there first and come back.

1.1 Hello, kaikai

The oldest exercise in the book, in kaikai:

```
fn main() {
  println("Hello, kaikai")
}
```

```
$ kai run examples/ch01/01_hello.kai
Hello, kaikai
```

Four things to notice before moving on:

- Every kaikai program starts at `fn main()`. There is no configuration file, no `package main` declaration, no enclosing class. A function with that name in some file is enough.
- `fn` introduces functions. The keyword is short on purpose — you will type it a lot.
- Curly braces `{ … }` group a block of statements, but a block is also an expression: the last value it produces is the value of the block. We don't lean on this here, but you will use it.
- `println` does not require an `import`. It is available in every program because it writes to standard output through an effect that kaikai installs by default. Chapter 9 opens that box; for now, it just works.

There is no semicolon at the end of the line. There is no `return` for a function that does not return a value. You don't even need to declare a return type on `main` when there is nothing useful to return. All of this is by design: kaikai tries not to make you write the obvious.

1.2 Algebraic types and `match`: FizzBuzz

The classic interview exercise, in kaikai:

```
type Tag
  = Both
  | Fizz
  | Buzz
  | Other(Int)

fn classify(n: Int) : Tag {
  if n % 15 == 0    { Both }
  else if n % 3 == 0 { Fizz }
  else if n % 5 == 0 { Buzz }
  else               { Other(n) }
}

fn label(c: Tag) : String {
  match c {
    Both     -> "FizzBuzz"
    Fizz     -> "Fizz"
    Buzz     -> "Buzz"
    Other(n) -> int_to_string(n)
  }
}

fn loop(i: Int, n: Int) : Unit / Stdout {
  if i <= n {
    println(label(classify(i)))
    loop(i + 1, n)
  }
}

fn main() {
  loop(1, 15)
}
```

What is interesting about this version is not that it prints `1, 2, Fizz, 4, Buzz, ...`. Any language can do that. What is interesting is what we did to get there.

We defined a **sum type**: `Tag` is one of four constructors. Three are bare names (`Both`, `Fizz`, `Buzz`) and one carries a payload (`Other(Int)`). If you come from an imperative language, this looks like an `enum` with associated data. If you come from an object-oriented one, it looks like a sealed class hierarchy. The difference is that this declaration brings no inheritance, no virtual methods, nothing beyond what you see: four ways to construct a value of type `Tag`.

`classify` decides which of the four to build. Look at the `if`: no `then`, no parentheses around the condition, and each branch is a block that produces a value. The `if` itself is an expression that returns a `Tag`, and the body of the function *is* that expression — no `return`, no intermediate assignment. This is what chapter 2 will call **expression, not statement**, and it is one of the few habit changes you will have to make.

`label` consumes a `Tag` with `match`. Each arm is a **pattern** followed by `->` and the expression that pattern produces. The pattern `Other(n)` doesn't only say "this was built with the `Other` constructor"; it also unpacks the payload and binds it to the name `n`, ready to use on the right-hand side. Destructuring, comparing, and declaring a name happen in a single move.

`loop` is recursive. There is no `while`, no `for`. Well — there are conveniences for iteration in chapter 6, but the base is recursion. So that base does not cost your program anything, the language guarantees **mandatory tail-call optimization**: a recursive call in tail position does not consume stack. `loop(1, 1_000_000)` works without blowing up.

One thing that will look strange and that we leave for chapter 9: the signature of `loop` says `: Unit / Stdout`. The part after the slash is the set of **effects** the function uses. `Stdout` means "this function writes to the terminal". Without it, the compiler would not let you call `println` inside. Don't worry about the details yet — the full story is in chapter 11.

1.3 A calculator with a recursive AST

Something with a bit more meat. A small calculator that represents arithmetic expressions as a tree.

```
type Expr
  = Lit(Int)
  | Add(Expr, Expr)
  | Mul(Expr, Expr)
  | Neg(Expr)

fn eval(e: Expr) : Int {
  match e {
    Lit(n)    -> n
    Add(l, r) -> eval(l) + eval(r)
    Mul(l, r) -> eval(l) * eval(r)
    Neg(x)    -> -eval(x)
  }
}

fn main() {
  let e = Add(Lit(2), Mul(Lit(3), Lit(4)))
  println(int_to_string(eval(e)))
}
```

```
$ kai run examples/ch01/03_calculator.kai
14
```

`Expr` is a sum type just like the one in FizzBuzz, with one difference: **it mentions itself in its own constructors**. `Add` takes two `Expr`s. So does `Mul`. `Neg` takes one. As a result, a value of type `Expr` can be a tree of any depth.

That is the key tool for representing languages, configurations, queries, commands, almost any structure with nesting. You will see it often. Chapter 5 dedicates a whole section to this pattern.

`eval` walks the tree with `match`. Each case recurses on the children. Exhaustiveness is checked by the compiler: if you add a constructor to `Expr` and forget an arm in `eval`, it does not compile. This is huge and will save you many hours. Chapter 5 explores it carefully; for now, trust it.

`let` introduces a local binding. The type is inferred from the right-hand side. There is no `var`, no `mutable`, no reassignment: `let e = ...` binds `e` to a value, and that value does not change. If you really need to mutate something, kaikai lets you, but it asks you to declare it (chapter 12). This is the other half of the habit change: **immutability by default**.

1.4 A custom effect with a handler

Until now, every effect we used was `println`, which works because kaikai installs a default handler for it. Let's see what happens when we declare our own.

```
effect Log {
  log(msg: String) : Unit
}

fn greet(name: String) : Unit / Log {
  Log.log("hello, " ++ name)
}

fn main() {
  handle {
    greet("kaikai")
    greet("world")
  } with Log {
    log(msg, resume) -> {
      println("[INFO] " ++ msg)
      resume(())
    }
  }
}
```

```
$ kai run examples/ch01/04_effect.kai
[INFO] hello, kaikai
[INFO] hello, world
```

This is the example most likely to make you slow down. That is deliberate. Algebraic effects are kaikai's distinctive bet, and we want you to see them running before we explain them seriously.

What is going on:

- `effect Log { log(msg: String) : Unit }` declares a new effect called `Log` with one operation, `log`, taking a string and returning nothing.
- `greet` uses that operation. Its signature — `: Unit / Log` — declares that the function has the `Log` effect, without saying *how* the effect is realised. `greet` is agnostic: it doesn't know whether messages go to the terminal, to a file, or nowhere at all.
- The decision happens at `handle … with Log { … }`. There, inside `main`, we say: "for this block, when someone invokes `Log.log(msg)`, run this code". The handler prints the message with an `[INFO]` prefix and hands control back via `resume(())`, which continues the program where it left off.

This looks like try/catch, like a dependency-injection container, like middleware, like callbacks. But **it is one idea** that subsumes all four. If it confuses you the first time around, that's fine. Chapter 11 returns to it with time and several examples before asking you to write a handler of your own.

What is worth keeping from this section: the type of `greet` tells you it needs `Log`. The compiler will not let you call it from a context where `Log` is not handled. Effects are **visible in the type**, not hidden. This solves an old itch of languages that have invisible exceptions.

1.5 Two cooperative fibers

The fifth program of the tour uses concurrency.

```
import spawn

fn worker(tag: String, n: Int) : Unit / Stdout + Spawn {
  if n > 0 {
    println(tag)
    spawn.yield()
    worker(tag, n - 1)
  }
}

fn main() {
  let f = spawn.spawn(() => worker("B", 3))
  worker("A", 3)
  spawn.await(f)
}
```

```
$ kai run examples/ch01/05_concurrent.kai
A
B
A
B
A
B
```

A **fiber** is a unit of cooperative execution. It weighs much less than an OS thread and lives inside the process. `spawn.spawn` schedules a new fiber but does not run it immediately;

the scheduler picks it up at the next cooperation point. `spawn.yield` is exactly that: a point where the current fiber says "I can wait — give someone else a turn".

Without the `spawn.yield` calls, worker A would run all three iterations before giving B a chance. With them, the output ends up interleaved.

The signature of `worker` is `: Unit / Stdout + Spawn`. Two effects: the one we already knew for printing, and `Spawn` for spawning and coordinating fibers. The `+` operator composes effects: a function may carry several at once, declared in its type.

`(() => worker("B", 3))` is a **lambda**: an anonymous function with no arguments that calls `worker`. We pass it to `spawn.spawn` so it runs inside the new fiber.

There is much to say about kaikai's concurrency model — why fibers are isolated, how they cancel, what happens to memory — but all of it lives in chapter 12. What matters for the tour is that the language has structured concurrency as a first-class feature, and it is treated, once again, as an effect.

1.6 Custom-fitted types with protocols

By now you've seen primitive types and sum types. One construct is missing: **records**, which are what most languages call a *struct* — a named-fields aggregate.

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

And with that comes the natural question: how do you "add operations" to a type? For example, how do we tell the compiler that my `Point` knows how to print itself as a string?

kaikai's answer is **protocols**: a named contract with a small set of operations, that any type may satisfy. Conceptually it matches Go interfaces, Rust traits, or Clojure / Elixir protocols.

```
#[derive(Show)]
type Point = { x: Int, y: Int }

fn main() {
  let p = Point { x: 3, y: 4 }
  println(show(p))
}
```

```
$ kai run examples/ch01/07_protocols.kai
Point { x: 3, y: 4 }
```

`Show` is one of the stdlib protocols (`Eq`, `Ord`, `Hash`, `Show`, `Serialize`). Its contract is a single op: given a value, return a `String`. The line `#[derive(Show)]` above the record tells the compiler to **synthesize** a `Show` implementation for `Point`, walking the fields and delegating to each one's `Show`. Since `Int` already implements `Show` in the stdlib, the whole record is covered without writing anything else.

A hand-written implementation would look like:

```
impl Show for Point {
  fn show(p: Point) : String =
    "(" ++ show(p.x) ++ ", " ++ show(p.y) ++ ")"
}
```

And `show(Point { x: 3, y: 4 })` would now return `"(3, 4)"` instead of the record's default format.

The takeaway for the tour: **kaikai picks explicit single-dispatch**, not Haskell-style typeclasses. No constraint inference, no higher-kinded types, no chained ad-hoc parametric polymorphism. One simple mechanism, like Go or Clojure. Chapter 9 develops the idea.

1.7 Units of measure

kaikai ships an uncommon feature for *mainstream* languages: **units of measure**. F# has had them since 2010 and almost no other language offers them out of the box. The idea is to mark a number with a unit (`Real<USD>`, `Real<m/s>`, `Int<Seconds>`) and let the compiler reject incompatible mixes.

```
unit USD
unit EUR

fn main() {
  let price : Real<USD> = 1.50<USD>
  let total : Real<USD> = price + 2.00<USD>
  println("total = #{total}")
}
```

```
$ kai run examples/ch01/08_units.kai
total = 3.5 USD
```

`unit USD` declares a unit. `1.50<USD>` is an annotated literal. `Real<USD>` is the type of a real with that unit. And if you try:

```
let mix = price + 1.00<EUR>     # error: USD ≠ EUR
```

the compiler complains before the program runs. This catches an entire class of bugs that usually surface in production: the classic Mars Climate Orbiter[1], adding balances in different currencies, passing a timeout in milliseconds where seconds were expected.

The best part of the scheme is that **units are erased at compile time**. The binary `kai build` produces operates on plain `Real`, no overhead. Same promise as effects: information in the type, zero cost at runtime.

There is much more to say — generic units, unit algebra (`m/s^2`, `kg * m / s^2`), explicit conversions, and a very useful variant called *branded types* that tags strings and integers

[1]NASA's Mars Climate Orbiter spacecraft was lost in September 1999 as it entered the Martian atmosphere. The root cause: one software module computed thrust in pound-force per second (imperial units) while another read the same value as newtons per second (metric units). Nobody had labeled the units at the interface. The mission cost USD 327 million.

with names like `UserId` or `OrderId` so the compiler won't let you confuse them. All of that lives in chapter 10. For now, knowing it exists is enough.

1.8 Inline tests

kaikai treats tests as first-class citizens: they live in the same file as the code they exercise, with their own syntax beside the functions.

```
fn square(n: Int) : Int = n * n

test "square of zero" {
  assert square(0) == 0
}

test "square preserves positives" {
  assert square(7) == 49
}

test "square of negatives" {
  assert square(-5) == 25
}
```

```
$ kai test examples/ch01/06_tests.kai
  ok   square of zero
  ok   square preserves positives
  ok   square of negatives

3/3 tests passed
```

`test "..." { ... }` is a top-level block. Inside, you use `assert` to write assertions — if one fails, the test fails and the runner moves on. In a normal build (`kai run`, `kai build`), `test` blocks are ignored: they don't add weight to the binary you ship.

There are two close relatives that share the same shape:

- `check "..." with x: T { ... }` declares a **property** the runner verifies with random inputs. This is what other languages call property-based testing.
- `bench "..." { ... }` is a benchmark: the runner runs the block many times and reports nanoseconds per iteration.

The three forms complement each other: `test` for fixed cases, `check` for invariants that must hold over any input, `bench` to measure performance instead of guessing. Chapter 7 treats each one in detail.

1.9 Installing and running `kai`

To run any of the programs above you need the `kai` binary. The project lives at github.com/kaikailang-org/kaikai.

From a fresh checkout, all you need is a C compiler:

```
$ make all
$ make test
```

`make all` builds stage 0 (written in C), stage 1 (written in kaikai-minimal and compiled by stage 0), and the `kai` binary in the repo root. `make test` runs the stage 0, stage 1, and phase 4 test suites to confirm the build is healthy.

From there, the three commands you'll use throughout the book are:

```
$ kai run file.kai     # compile and run
$ kai build file.kai -o name   # produce a native binary
$ kai test file.kai    # run the `test "..." { ... }` blocks in the file
```

`kai run` is the workhorse while you read this book. Edit a file, run it, look at the output, edit again.

Chapter 16 covers the rest of the tooling: `fmt`, `lsp`, `watch`, editor integration. For now, `run` is enough.

1.10 How the rest of the book is laid out

We saw, without going deep, almost everything that makes kaikai distinctive. The rest of the book takes each piece and treats it seriously.

- **Part II — The Language** (chapters 3 to 10) covers basic types, compound types, sum types and `match`, functions, testing and benchmarking, modules, protocols, and units of measure. It is the solid, predictable half.
- **Part III — What Sets It Apart** (chapters 11 to 14) takes algebraic effects, fiber-based concurrency, actors, and the language's bet around LLMs. This is the half where kaikai earns its novelty.
- **Part IV — Practice** (chapters 15 and 16) covers the tooling and closes with an integrating case study.
- Before all that, **chapter 2** softens a few assumptions if you come from an imperative world: expressions vs statements, immutability by default, `Option` instead of `null`, visible effects. It is short but useful.

If you come from Haskell, OCaml, Elixir, or Scala, you can skip chapter 2 and even skim Part II; what is new for you lives in Part III. If you come from Python, Go, Java, JavaScript, or C#, read chapter 2 carefully and do the exercises in Part II.

Either way: the example sources are in `examples/` in the book repository. Compile everything. Run everything. The only way to learn a language is to write it.

Chapter 2 · Thinking in kaikai

Chapter 1 showed you the language from above. Before getting into the detail of types, functions, and modules, it is worth pausing on a handful of habits kaikai asks of you that you probably did not bring with you from Python, Java, Go, JavaScript, or C#.

This is the shortest chapter in the book, and you can skip it. If you have already programmed in Haskell, OCaml, Elixir, or Scala, what follows will sound familiar — head to Part II and we'll meet you in chapter 3. If you come from an imperative background, give it the twenty minutes it asks for. They will save you a lot of friction across the next hundred and fifty pages.

We are not going to open up the theory of each idea — that's the job of the chapters that follow. What we want is to name the habit change, show it, and give you the words to recognize it when it shows up.

2.1 Expressions, not statements

In most of the languages you probably know, code is built from two kinds of pieces:

- **Expressions**, which produce a value: `x + 1`, `f(2)`, `a == b`.
- **Statements**, which do not produce a value but execute something: a two-armed `if`, a `for`, a `return`, an assignment.

Statements have to be lined up in a sequence. Expressions do not: they compose by nesting.

kaikai erases that boundary. **Almost everything is an expression.** An `if` produces a value. A `match` produces a value. A block `{ ... }` produces a value — the value of its last expression. A function does not need a `return` because its body *is* the expression that gets returned.

Compare two ways of writing the same thing. The classic imperative form:

```
String s;
if (x > 0) {
  s = "positive";
} else {
  s = "non-positive";
}
print(s);
```

In kaikai:

```
let s = if x > 0 { "positive" } else { "non-positive" }
println(s)
```

The difference is not in line count. It's in how you think about the code. The imperative version forces you to **declare `s` first**, because the `if` cannot return anything; then it **assigns to `s` in each branch**. In kaikai, the `if` *is* the value, and `s` binds directly to the result. Declaration and assignment merge. There is no assignment at all: `s` is bound once and never changes.

The practical consequences show up quickly:

- **Fewer lines, no loss of clarity.** Three steps (declare, decide, assign) collapse into one.
- **Fewer temporaries.** If you only need a value to pass into the next call, you build it inline.
- **Fewer "forgot to initialise" bugs.** No declared-but-uninitialised variables.
- **Smoother refactoring.** An expression can be lifted into a function or replaced by another expression without disturbing the surrounding context; a statement, less so.

You see the same with `match`. In most languages with a `switch`, each `case` is a statement that runs a block and then either breaks or falls through, depending on the rules. In kaikai, `match` is an expression that returns a value, and each arm is the expression that value might be. You saw this in chapter 1, in `label` and in `eval`. You will see it constantly.

A related point: the body of a function can take two shapes, and the choice between them communicates intent.

```
fn double(x: Int) : Int = x * 2

fn classify(x: Int) : String {
  if x < 0 { "negative" }
  else if x == 0 { "zero" }
  else { "positive" }
}
```

Use `=` and a single expression when the function is direct. Use `{ … }` when there are several steps or visual separation helps. The compiler accepts both; the difference is for the reader.

A useful consequence of treating everything as an expression is that **chained transformations** become comfortable. From the Elixir world, kaikai borrows the pipe operator `|>`:

```
xs |> filter(is_even) |> map(double) |> list.length
```

is equivalent to `list.length(map(filter(xs, is_even), double))`. The piped form reads left-to-right, in the order the transformations happen, and avoids intermediate names. It is only possible because each step is an expression that can be composed with the next.

`|>` is the most general of three pipe operators kaikai provides for chaining. The other two — `|` (map over lists) and `||` (flat-map) — are shortcuts for the cases that appear over and over. Chapter 6 covers the three in detail.

2.2 Immutability by default

`let x = 5` binds `x` to the value `5`. That's it. You cannot write `x = 6` later. If your code seems to need to "change `x`", in kaikai that means one of two things:

- You actually need a *new* value derived from the first. The right move is to bind another name, or to redefine `x` in a nested scope.
- You actually need *visible* mutation. That's a real but small case, and kaikai gives it to you, but it asks you to declare it. Mutation of an array, for example, lives under the `Mutable` effect, which appears in the signature of any function that uses it. We'll cover this calmly in chapter 12.

Why take this path? Because most of the time, "changing a variable" is a habit inherited from the imperative model, not a real need. When you program with values that don't change:

- **Reasoning becomes local.** If `x` was bound to `5` on line 12, it is still `5` on line 30. Period. No need to hunt for who modified it in between.
- **Concurrency becomes simpler.** Two fibers can read the same value without synchronization; nobody is going to overwrite it.
- **Bugs go away.** A whole category of errors ("I expected X, but at the end of the method it was Y") just doesn't exist.
- **Tests are simpler.** A pure function — input, output, no hidden state — is tested by giving it inputs and comparing outputs. That's all.

A note on vocabulary. The usual way to bind a name is `let`, which is immutable. For the cases where you really need a local mutable cell — a counter, an accumulator, a cursor — kaikai gives you `var`, along with two companion sugars: `@name` to read the cell and `name := v` to write it.

```
var n = 0
n := @n + 1
println(int_to_string(@n))  # 1
```

Here is the interesting part: `var` is not really a new construct in the language. It is **syntactic sugar** over the `State` effect. The line `var n = 0` rewrites to a `handle ... with State[Int](0) as n { ... }` covering the rest of the block. `@n` rewrites to `n.get()`, and `n := v` to `n.set(v)`. The base language is the same algebraic-effects machinery from chapter 11; what changes is the face it shows for common cases.

What matters for your mental model is that this rewrite happens **inside the block**: the `handle` opens and closes right there, so the `State` effect does not leak into the function's signature. A function with a `var` inside has the same signature it would have without it.

More visible mutations — writing to an array that lives beyond the block, sending to another actor's mailbox, modifying memory observed from outside — *do* show up in the signature, under effects like `Mutable`, `Actor`, or whichever fits. We'll see that distinction in chapter 12.

The practical rule is simple: use `let` by default; if you need a local variable that changes, `var` with `@` and `:=`; if what you want to mutate is something visible from outside, you are in effects territory and you'll have to declare them.

2.3 `Option` and `Result` instead of `null` and exceptions

The oldest question in language design: what does a function do when it cannot return what it promised?

The C, Java, Python, JavaScript answer (and many others) is to **lie**: the function says it returns a `User`, but in some cases it returns a magic value called `null` (or `None`, or `nil`) which **is not a user**, and which the type system does not distinguish from a real one. The caller has to remember to check. Tony Hoare, who invented the null reference back in 1965, called that decision his "billion-dollar mistake".

The other traditional answer is to throw an exception. The function does not return anything and, without telling the type system, it diverts control to some distant `catch`. Which `catch`? Depends. Sometimes there isn't one and the program dies.

kaikai picks a third path, old in the ML family but still uncommon outside of it: **encode the possibility of failure in the return type**.

```
type Option[a] = None | Some(a)
type Result[e, a] = Err(e) | Ok(a)
```

A function that may not find its result returns `Option[User]`: either `Some(usr)` when found, or `None` otherwise. A function that may fail in several ways returns `Result[Error, User]`: either `Ok(usr)`, or `Err(reason)`. In both cases, the type forces you to consider both possibilities.

Compare:

```
# Python: the caller has to remember
def find(id: int) -> User:
    ...  # sometimes returns None, sometimes not, check the docs

usr = find(7)
print(usr.name)   # crash if usr is None
```

```
# kaikai: the type says so
fn find(id: Int) : Option[User] = ...

let r = find(7)
match r {
  Some(usr) -> println(usr.name)
  None      -> println("not found")
}
```

In the kaikai version, `match` is exhaustive: if you forget the `None` branch, it does not compile. The compiler reminds you of what Python leaves to your memory.

What about exceptions? kaikai has an equivalent mechanism — the `Fail` effect, and more generally the algebraic effects from chapter 11 — but there too, what can fail appears in the type. The "invisible exceptions" that in Java or Python can spring out of any call do not exist in kaikai. If a function can jump elsewhere, its signature says so.

This changes how programming feels. Instead of wrapping every external call in a `try` just in case, you read the signature, see what can fail, and decide right there what to do.

A note on `!`

In kaikai, the postfix `!` operator applies to an `Option` or a `Result` and propagates the negative case: if the value is `Ok(v)` or `Some(v)`, the expression evaluates to `v` and the program continues; if it is `Err(e)` or `None`, the current function returns right there with that `Err` or `None`, handing it to the caller.

```
fn load() : Result[Error, User] {
  let id = parse_id(input)!      # if it fails, propagate
  let data = read_file(id)!      # likewise
  Ok(build_user(id, data))
}
```

This is the same as Rust's `?`. And it is worth saying what it **isn't**: in Elixir, naming a function `File.read!` is a convention for the version that *raises* an exception instead of returning an `{:ok, _} | {:error, _}` tuple. In kaikai the convention is the opposite: `!` is not part of a name, it is an operator, and it **never raises**. It only unpacks the happy case and propagates the negative one through the return type. If you have seen plenty of `File.read!` in Elixir code and it made you nervous, you can relax: in kaikai that same syntax is on the safe side.

2.4 Pattern matching as a control-flow tool

In an imperative language, deciding what to do based on the shape of a value usually takes three steps:

1. **Check the shape** with `if`, `instanceof`, `is`, `typeof`, or a discriminator field.
2. **Get at the data** carried by that shape, with casts, `as`, or named accesses that assume step 1 worked.
3. **Do something** with that data.

kaikai folds the three into a single construct: `match`.

```
match expr {
  Lit(n)    -> n
  Add(l, r) -> eval(l) + eval(r)
  Mul(l, r) -> eval(l) * eval(r)
  Neg(x)    -> -eval(x)
}
```

Each arm is a **pattern** followed by the expression that pattern produces. `Lit(n)` doesn't only say "this was built with `Lit`"; it also declares that `n` is the `Int` carried inside, ready to use on the right-hand side. Check, unpack, bind — in one move.

Patterns nest. If you have an `Option[Result[Error, Int]]` and you want to distinguish the three possible shapes, you write:

```
match x {
  None            -> "not present"
  Some(Err(reason)) -> "failed: " ++ reason
```

```
  Some(Ok(value))  -> "ok: " ++ int_to_string(value)
}
```

Patterns can also carry **guards** — extra conditions evaluated after the structural match — and wildcards (`_`) when you don't care about the value.

The crucial part: the compiler checks **exhaustiveness**. If `Tag` has four constructors and your `match` covers three, it does not compile. If you add a fifth constructor to a type and there are thirty `match` sites in the codebase, those thirty sites become compile errors that tell you exactly where to come back. This turns a feared refactor into a tedious but safe one.

Without pattern matching, languages handle this case with visitor patterns, class hierarchies, or chains of `if / else if / else`. Each works, but all of them lose the link `match` keeps between the type of the value and the way you decide on it.

After a few weeks with kaikai, `match` becomes one of those tools you don't want to give up.

2.5 Pure functions and visible effects

The four ideas above converge on a bigger one, which is the language's central bet: **separate the pure from what touches the world**, and have the type system police that distinction.

A *pure* function is a function whose result depends only on its arguments. Calling it with the same arguments always returns the same value. It does not print. It does not read from disk. It does not send messages. It does not look at a clock. It does not roll a die.

Pure functions are easy to test, easy to reason about, easy to parallelize, easy to cache. The catch is that a program made only of pure functions does nothing useful: it never talks to the world.

kaikai puts effects in the type system. You saw this in chapter 1: a function that writes to stdout says `: Unit / Stdout` in its signature. A function that uses `Log` says `: … / Log`. A function that may fail and bail out says `: … / Fail`. And a pure function, one that does not touch the world, says nothing after the `/`. Its signature is just `fn f(x: Int) : Int`.

The effect on the right-hand side of `/` is not decoration. It is a constraint: the compiler will not let you call an effectful function from a context where those effects are not being handled. Some handler up the stack has to take ownership. This solves, all at once and at the language level, several old itches:

- The **invisible exceptions** that Java and Python permit because no signature declares them.
- The **cancellation handling** that in `async` / `await` languages becomes a thread of plumbing inside the business logic, rather than a separate mechanism.
- The **dependency injection** that in OO languages requires whole containers to do what an effect handler does in five lines.
- **Logging**, **config access**, **the clock**, **the database**: every dependency that traditionally sneaks in hidden can be an effect, declared in the type, and chosen by the caller.

If you've never seen this before, it sounds too ambitious to be true. It is and it isn't. Chapter 11 gives it the space it deserves. For now, it is enough to know that the signatures you see with `/ something` are not noise: they are information about what that function can do to your program.

2.6 A short genealogy

kaikai did not appear from nowhere. It inherits decisions from several language families, and it helps to know which ones to understand why things look the way they do.

- **From ML (1973), OCaml, and Haskell** come algebraic data types, pattern matching, Hindley-Milner type inference, and `Option` / `Result`. Old, good ideas that "modern" languages like Rust and Swift now rediscover without always acknowledging their grandparents.
- **From Erlang (1986) and Elixir** come isolated processes with private memory, the idea that concurrency is built on passing messages rather than sharing memory, and the supervision model with `link` and `monitor`. In kaikai those processes are called fibers and actors.
- **From Elixir** also comes the pipe operator `|>` we saw earlier.
- **From Koka (Daan Leijen, Microsoft Research)** and **Effekt (Andreas Rossberg, Jonathan Brachthäuser)** come algebraic effects with effect rows in the type system. kaikai follows Effekt closely on how handlers are expressed, and Koka on some internal decisions.
- **From Koka** also comes **Perceus**, the compile-time-optimized reference counting scheme kaikai uses to manage memory without a garbage collector or a borrow checker.
- **From Go** kaikai takes the decision to ship a single binary as compiler, formatter, and test runner; the primacy of programs that build and run fast; and the discipline of keeping few forms in the language.
- **From Rust**, kaikai learns what *not* to do: sum types and pattern matching are lessons Rust teaches well. The borrow checker, on the other hand, kaikai prefers to avoid — Perceus plus isolated fibers solve the problem without asking the programmer to internalize lifetimes.

None of these decisions is new. What kaikai attempts is a coherent combination: algebraic types + algebraic effects + Perceus + BEAM-style fibers, in a language that compiles fast to native code and that an experienced programmer can read without taking a course first.

The rest of the book digs into each of those decisions. If you made it this far, you have the map.

Exercises

2.1. Go back to `02_fizzbuzz.kai` from chapter 1. Identify all the **expressions** that return a value. How many are there? Are there any strict **statements** — something that runs for its side effect without producing anything useful — beyond the calls to `println`?

2.2. In your day-to-day language, write a short version of the following: a function `classify_age(n)` that returns the string `"child"`, `"young"`, `"adult"`, or `"senior"` based on ranges. How

many local variables did you use? How many assignments? Now rewrite it in pseudo-kaikai (it doesn't have to compile) using `if` as an expression.

2.3. Find a function in your work codebase that returns `null` or throws an exception to signal "no result". Write down in a comment what its signature would be in kaikai using `Option` or `Result`. What do you gain? What do you lose?

Chapter 3 · Basic types and expressions

Time to get to the primitives. This chapter covers kaikai's seven basic types, the shape of literals, the operators that handle them, and how values bind to names with `let`. It also pins down something you've already seen in passing: **`if` and blocks are expressions**, not statements.

If you read chapter 2, what follows is the concrete content of those warnings. If you skipped it and you come from an imperative background, go back. It will save you friction.

3.1 The seven primitive types

kaikai has exactly seven primitive types:

Type	What it's for	Example literal
`Int`	Signed 64-bit integers	`42`, `-7`, `1_000_000`
`Real`	Double-precision IEEE 754 floats	`3.14`, `-0.5`, `1e10`
`Bool`	True / false	`true`, `false`
`String`	Unicode text	`"hello"`, `"α + β"`
`Char`	One Unicode character	`'a'`, `'\n'`, `'\u{2603}'`
`Unit`	The type with one value	`()`
`Nothing`	The empty type, no inhabitants	(no literal)

The first five are what you'd expect from any language. The last two deserve a word.

`Unit` is the type of the value `()`. It has a single inhabitant. It's what a function returns when it "doesn't return anything useful": the equivalent of C's `void`, but it's a real type with a real value, one you can pass as an argument or store in a variable. `println(...)` returns `Unit`. A block that ends without an explicit value returns `Unit`.

`Nothing` is the opposite. **Zero inhabitants.** No expression produces a `Nothing` the program can use. So what is it for? For describing the return type of things that **never finish normally**: a `panic(...)` that aborts the process, an infinite loop, a `forever` whose body cannot exit. A function `fn endless_loop() : Nothing` tells the type system that any code after calling it is unreachable, which is why an expression of type `Nothing` fits any context where another type is expected. You'll bump into `Nothing` rarely, but it helps to know the name.

3.2 Literals and string interpolation

Numeric literals accept underscores as visual separators:

```
let population = 19_000_000
let pi         = 3.141_592
let scale      = 1e6
```

`String` values use double quotes. The day-to-day shape is **interpolation**:

```
let name = "kaikai"
let age  = 1
println("Hello, #{name}. The language is #{age} year old.")
```

Output:

```
Hello, kaikai. The language is 1 year old.
```

Any expression fits inside `#{...}`, not just names. Its value is converted to `String` automatically:

```
let a = 7
let b = 2
println("The sum of #{a} and #{b} is #{a + b}.")
```

To concatenate without interpolation, use `++`:

```
let greeting = "Hello, " ++ name
```

The operator is `++`, not `+`, to avoid confusing it with addition. It's a small detail that prevents a classic bug.

For multi-line content, kaikai has **triple-quoted strings**:

```
let message = """
  This is a multi-line
  message that preserves indentation
  relative to the closing fence.
  """
```

Each line's indentation is measured against the closing `"""`, so you can format the string visually without dragging extra spaces into the output. Escape sequences (`\n`, `\t`, `\u{HHHH}`, etc.) work the same as in regular strings.

`Char` values use single quotes: `'a'`, `'\n'`, `'\u{2603}'`. A `Char` is not a `String` of length one — they are distinct types. That avoids a class of bugs typical of languages that conflate the two.

3.3 Arithmetic, logical and comparison operators

The arithmetic operators:

```
+   addition            (Int or Real)
-   subtraction / negate  (Int or Real)
*   product             (Int or Real)
/   division            (Int or Real)
%   remainder           (Int or Real)
```

The five are **overloaded by type**: they work with both `Int` and `Real`, and the result has the same type as the operands. What does **not** exist is implicit coercion between `Int` and `Real`: you cannot mix them in the same expression. If you need to, you convert explicitly with `int_to_real(...)` or `real_to_int(...)`.

The detail worth pinning down: `/` with two `Int`s already truncates the remainder. To get a quotient with a fractional part, both operands have to be `Real`.

```
let a : Int = 7
let b : Int = 2
println("a / b = #{a / b}")     # 3 — on Int, / truncates

let x : Real = 7.0
let y : Real = 2.0
println("x / y = #{x / y}")     # 3.5 — on Real, fractional
                                #       part is preserved
```

If you come from Python 3 there's a habit change. Over there, `/` always returns float; in kaikai the type rules: `Int / Int` is `Int`, and if you want the fractional part you convert explicitly or work with `Real` from the start.

Logical operators are words, not symbols:

```
and   or   not
```

```
if x > 0 and x < 10 { ... }
if not empty { ... }
```

Most languages you've used reach for `&&`, `||` and `!`. kaikai went with words because they read more cleanly, and because those symbols are reserved for other purposes (`||` is flat-map in pipes, chapter 6).

Comparison operators are the usual ones:

```
==  !=  <  >  <=  >=
```

They return `Bool`. They work over any type that implements the `Eq` protocol (for `==` / `!=`) and `Ord` (for the rest). We'll see those in chapter 9; for now, every primitive type implements them.

3.4 `let` and local type propagation

`let` binds a name to a value. The type is inferred from the right-hand side:

```
let x = 42              # x : Int
let y = 3.14            # y : Real
let name = "kaikai"     # name : String
```

If you want the type explicit, annotate with `:`:

```
let x : Int  = 42
let y : Real = 3.14
```

The annotation is more than decoration. It's there for two reasons: to document intent when the type isn't obvious, and to guide the inferer in the few cases where local inference isn't enough. The practical rule is **annotate the arguments and return type of public functions, and leave local `let`s unannotated**. Types travel through the signatures and the body stays clean.

Once a name is bound, you can't bind the same name again in the same block. The next line would error:

```
let x = 42
let x = 7       # error: name already defined in this scope
```

What you *can* do is bind the same name **in an inner scope**, which produces *shadowing* — the local name hides the outer one:

```
let x = 42
{
  let x = 7     # OK: shadowing inside the block
  println("#{x}")   # 7
}
println("#{x}")     # 42 — the outer one was untouched
```

This is orthogonal to mutation. You're not "changing `x`": you're introducing a different `x` in a nested scope, which ceases to exist when the scope closes. The outer one was never touched.

For the cases where you need a real mutable cell, kaikai gives you `var`, with `@name` and `name := v`. You saw this in §2.2 and we'll come back to it in chapter 11 when we talk about effects.

3.5 `if` as an expression

An `if` in kaikai produces a value:

```
let s = if x > 0 { "positive" } else { "non-positive" }
```

Both branches must produce values of the same type, and that's the type of the `if`. The syntax has no `then`, no parentheses around the condition, and each branch is a block.

If the condition has multiple branches, they chain with `else if`:

```
fn sign(n: Int) : String =
  if n < 0 { "negative" }
  else if n == 0 { "zero" }
  else { "positive" }
```

Each brace opens a block whose value is the result of the branch. The whole function is a single expression, bound to the signature with `=`. There's no `return`.

A variant worth pinning down: **an `if` without `else` always has type `Unit`**. The compiler does not synthesize a value for the missing branch; it makes the simplest possible call and says "the type of the `if` is `Unit`, and if the condition is false, the value is `()`".

So the usual shape of an `if` without `else` is a body that runs for its side effect:

```
if i <= n {
  println(label(classify(i)))
  loop(i + 1, n)
}
```

It's the standard "do something or stop". And if the `then` branch happens to produce a value of a different type, that value **is silently discarded** — the `if` is still `Unit`:

```
if x > 0 {
  x + 1    # an Int, but thrown away
}
```

This compiles, produces no warning, and is rarely what you intended. The error usually shows up a little later, when you try to use the result of the `if`:

```
let r = if x > 0 { 42 }
println(int_to_string(r))   # error: r is Unit, not Int
```

The practical rule is simple: if **you care about the value**, write a full `if/else`; if **you care about the side effect**, write a bare `if` and don't bind it to anything.

3.6 Blocks and the value of a block

A block `{ … }` is an **expression** whose value is the value of the last expression inside. The earlier lines run in order, and their values are discarded — except where you bind them with `let`.

```
fn square_plus_one(x: Int) : Int = {
  let square = x * x
  square + 1
}
```

`square` is a local binding. The last line, `square + 1`, is the return expression. There's no `return`; the block's value is the function's value.

This composes with everything else: an `if` branch can be a block, a `match` arm can be a block, a lambda body can be a block. The whole language reduces to expressions returning values.

3.7 The difference between `=` and `{ ... }` in a function body

A function body can take two shapes. You saw this in chapter 1; we pin it down here.

Short form, with `=` and a single expression:

```
fn double(x: Int) : Int = x * 2

fn sign(n: Int) : String =
  if n < 0 { "negative" }
  else if n == 0 { "zero" }
  else { "positive" }
```

Long form, with `{ ... }`:

```
fn square_plus_one(x: Int) : Int = {
  let square = x * x
  square + 1
}
```

The compiler accepts both. The difference is for the reader:

- **Short form** when the function is a direct expression with no intermediate steps. Reads like a mathematical definition.
- **Long form** when there are intermediate bindings or several steps that benefit from visual separation.

There's no "preferred" form: pick whichever communicates intent better for that particular function. The usual rule of thumb is that one-line functions go with `=`, and the ones that need to breathe go with `{ ... }`.

Exercises

3.1. Write a function `fn fahrenheit_to_celsius(f: Real) : Real` using only what's in this chapter. Verify that `fahrenheit_to_celsius(32.0)` gives `0.0` and that `fahrenheit_to_celsius(212.0)` gives `100.0`. Use `=` and a single expression.

3.2. Write `fn is_even(n: Int) : Bool` and `fn parity(n: Int) : String`, where `parity` returns `"even"` or `"odd"`. The second function should use the first, not duplicate the logic.

3.3. Given the function:

```
fn total_price(units: Int, unit_price: Int) : Int = {
  let subtotal = units * unit_price
  let tax = subtotal * 19 / 100
  subtotal + tax
}
```

What does `total_price(3, 100)` print? Modify the function so the tax is computed correctly as a real-valued percentage (with fractional part), not as an integer. Which types do you change?

3.4. Write `fn maximum(a: Int, b: Int, c: Int) : Int` that returns the largest of the three arguments, using `if` as an expression and without `match` or stdlib helpers. How many `else if`s do you need?

3.5. Read the documentation for `++` and figure out what happens if you write `"hello, " ++ 42`. Does it compile? If not, how do you fix it? Reason about it before testing.

Chapter 4 · Compound Types

The seven primitives from chapter 3 give you raw pieces. To build real programs, you glue them into structures: aggregates with named fields, lists, tuples, and the two stdlib gems you will see more than any other type, `Option` and `Result`.

This chapter covers all of that. **Sum types** (`type Tag = Foo | Bar(Int)`) you saw in the tour deserve their own chapter and we treat them in chapter 5.

4.1 Records

A **record** is an aggregate with named fields. You declare it with `type` and braces:

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

type Employee = {
  name: String,
  age: Int,
  salary: Int,
}
```

The trailing comma on the last field is optional but idiomatic: it means adding a new field doesn't touch the previous line, which keeps git diffs clean.

To build a record value, you write the type name followed by braces with the fields:

```
let origin = Point { x: 0, y: 0 }
let p      = Point { x: 3, y: 4 }
let ada    = Employee { name: "Ada", age: 30, salary: 1500 }
```

To read a field, you use `.`:

```
println("p is at (#{p.x}, #{p.y})")
println("#{ada.name} is #{ada.age}")
```

That's enough for the day-to-day. Three details worth pinning down:

- **Records are immutable.** `p.x = 7` does not exist. If you need a point with one different field, you build a new one with the **spread** sugar:

```
let p  = Point { x: 3, y: 4 }
let p2 = Point { ...p, x: 7 }    # Point { x: 7, y: 4 }
```

`...p` copies all fields of `p`; the named initializers after it (here `x: 7`) override what they repeat. It's the same idea as list spread (see §4.3) applied to records.

- **Records are nominal.** `Point { x: Int, y: Int }` and another `Position { x: Int, y: Int }` with the same fields are distinct types. The compiler doesn't conflate them even if they have the same shape. This is on purpose: if you want a position, say position.
- **Spread has rules.** Only one spread per literal, and it must come first — `Point { x: 7, ...p }` is a parse error. The initializers that follow must be named (`x: expr`), not punned or positional. These are deliberate: they make it obvious who wins on a duplicate.

4.1.1 Private fields

Record fields are public by default: any module that imports the type can read them and name them when constructing a literal. The `priv` keyword in front of a field name flips that default:

```
# module `safe`
pub type Account = {
  name:       String,   # public by default
  priv balance: Real,   # private to module `safe`
}

pub fn open(name: String) : Account =
  Account { name: name, balance: 0.0 }

pub fn deposit(a: Account, amount: Real) : Account =
  Account { ...a, balance: a.balance + amount }

pub fn balance_of(a: Account) : Real = a.balance
```

From inside the `safe` module, the `balance` field reads and writes like any other. From outside, it doesn't:

```
import safe

fn main() {
  let a = safe.open("savings")
  println("#{a.name}")          # OK, `name` is public
  println("#{safe.balance_of(a)}")  # OK, going through the getter

  # The next two lines don't compile:
  # println("#{a.balance}")        # ← field `balance` is private to module `safe`
  # let d = safe.Account {         # ← cannot construct `Account` from outside ...
  #   name: "x",
  #   balance: 1000.0,
  # }
}
```

The rule is strict: no external reads, no mention inside a construction literal. If module `safe` wants a consumer to create accounts, it exposes constructors (`open`) and operations (`deposit`) that preserve the invariants; the raw field stays hidden.

This turns a record into a lightweight abstract type: public shape, interior under the author's control. We'll use this in ch. 17 to hide the store actor's state, and in ch. 18 so ledger balances can't be constructed from outside the domain module.

4.2 Field access and destructuring

Accessing with `.` is fine for one or two fields. When you need several at once, **destructuring** is cleaner:

```
fn distance_squared(a: Point, b: Point) : Int = {
  let Point { x: ax, y: ay } = a
  let Point { x: bx, y: by } = b
  let dx = ax - bx
  let dy = ay - by
  dx * dx + dy * dy
}
```

When the field names work as variables, you can drop the `:` and just give the field name — binding the field to a variable of the same name:

```
fn describe(p: Point) : String = {
  let Point { x, y } = p
  "(" ++ int_to_string(x) ++ ", " ++ int_to_string(y) ++ ")"
}
```

Destructuring works in `match` too. That's its most useful form: deciding what to do **based on the specific values of the fields**:

```
fn classify(p: Point) : String =
  match p {
    Point { x: 0, y: 0 } -> "origin"
    Point { x: 0, y: _ } -> "Y axis"
    Point { x: _, y: 0 } -> "X axis"
    Point { x, y }       -> "point at (#{x}, #{y})"
  }
```

The compiler checks exhaustiveness, just like with sum types: remove the last arm and it stops compiling. Exhaustiveness over `Int` fields is covered by the final `Point { x, y }` arm, which matches any `Point`. Without that catch-all arm the `match` is not total.

4.3 Lists

A list is an immutable, linked sequence of values of the same type. The type is written with brackets around the element type: `[Int]`, `[String]`, `[Point]`, `[[String]]` (list of lists).

Building literals:

```
let primes = [2, 3, 5, 7, 11]
let empty : [Int] = []
```

kaikai also has **range literals**, sugar that produces a list:

```
let r1 = [1..10]       # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let r2 = [1..10..2]    # [1, 3, 5, 7, 9]
let r3 = [10..1..-1]   # [10, 9, 8, ..., 1]
```

To extend an existing list, use `...` (spread):

```
let xs = [1, 2, 3]
let ys = [0, ...xs, 99]     # [0, 1, 2, 3, 99]
```

To take them apart, the patterns of `match`:

```
fn sum(xs: [Int]) : Int =
  match xs {
    [] -> 0
    [h, ...t] -> h + sum(t)
  }
```

`[]` matches the empty list. `[h, ...t]` matches any non-empty list, binding `h` to the first element ("head") and `t` to the rest ("tail"). These two patterns cover every possible case, making the `match` exhaustive.

More specific patterns are also legal:

```
match xs {
  []                    -> "empty"
  [only]                -> "one: #{only}"
  [first, second, ...]  -> "at least two"
}
```

The compiler requires you to cover all cases. If you write just `[]` and `[h, ...t]`, it's enough for any list — but if you want to single out "exactly one element", you write `[only]` before the catch-all.

A language convention: if **the tail matters**, give it a name — `[h, ...t]` and then use `t`. If **it doesn't**, write `...` alone, no name — `[h, ...]`. It's the difference between "I take the head, the rest I keep for later" and "I take the head, the rest I discard". An invented name that goes unused is visual noise; the bare form communicates intent without forcing a name that adds nothing.

For indexed access, the stdlib exposes `list.nth`:

```
let first  = list.nth(xs, 0)   # Option[Int]
let third  = list.nth(xs, 2)   # Option[Int]
```

Notice the return type: `Option[a]`, not `a`. A linked list does not guarantee that an index exists — if you ask for element 99 of a list of three, there is no value to return. The type

forces you to consider it. This is consistent with `Option` and `Result` (see §4.5): kaikai prefers to enclose the possibility of failure in the type rather than abort at runtime.

Two more things about indexed access. One: it is **`O(n)`** — lists are linked, not indexed; walking to position `i` costs `i` steps. For fast random access kaikai has `Array[T]`, which we cover in chapter 13.

The other: the syntax `xs[i]` that some languages use for lists is reserved in kaikai for `Array[T]`. Writing it on a list is a type error. The reason is the same: `xs[i]` suggests cheap, guaranteed access, which would be lying about a linked list.

For most code, you don't want to index by hand anyway. Recursion over `[h, ...t]` or the higher-order functions of chapter 6 (`map`, `filter`, `reduce`) are the natural way to process lists.

Lists are **immutable**. There is no `xs[0] = 99`. If you need a modified list, you build a new one.

4.4 Strings, not lists of chars

Worth pausing on something many languages mix up: in kaikai, **a `String` is not a list of `Char`**. They are distinct types:

```
let s : String = "hi"
let cs : [Char] = ['h', 'i']
```

`s` and `cs` are not interchangeable. You can't write `s[0]` expecting a `Char`, and you can't pass a `String` where `[Char]` is expected.

Why? Because in Unicode there is no simple correspondence between "character" and "index". An emoji can occupy several code points; an accented letter may have one or two representations; a grapheme can skip bytes and code points arbitrarily. Treating a string as a list of chars forces a decision about what counts as "a character" — and every decision is wrong for some case.

kaikai makes the `String` **opaque**: the operations that make sense are exposed in the `string` module of the stdlib (`length`, `starts_with`, `ends_with`, `trim`, `repeat`, `join`) and we don't conflate text with lists. `string.length(s)` counts **bytes**, not characters or graphemes. For `"á"` it returns 2 (because "á" in UTF-8 takes two bytes); for `"👨"` it returns 3. It's a conscious choice: the internal representation of a string is UTF-8, and kaikai prefers a predictable and cheap answer over a philosophically correct but expensive one. If you need to count graphemes or logical characters, you use module functions that explicitly decode Unicode with its subtlety.

For concatenation, you saw it in chapter 3: use `++`:

```
let greeting = "Hello, " ++ name
```

For interpolation, `#{...}` inside a literal `"..."`.

4.5 `Option` and `Result`: the daily tools

You saw these two in chapter 2:

```
type Option[a] = None | Some(a)
type Result[e, a] = Err(e) | Ok(a)
```

Here we look at them in use. They are two generic sum types from the stdlib that you will use **constantly**. A reminder of the idea: `Option` represents "there might be no value"; `Result`, "there might be a value or an error".

A function that finds the first even element of a list:

```
fn first_even(xs: [Int]) : Option[Int] =
  match xs {
    [] -> None
    [h, ...t] -> if h % 2 == 0 { Some(h) } else { first_even(t) }
  }
```

And the caller has to consider both cases explicitly:

```
match first_even(xs) {
  Some(n) -> println("found: #{n}")
  None    -> println("no evens")
}
```

A function that parses an age from a string can fail in two distinct ways — and that's exactly what `Result` is for:

```
type AgeError = NotNumeric | OutOfRange

fn parse_age(s: String) : Result[AgeError, Int] =
  match string_to_int(s) {
    None -> Err(NotNumeric)
    Some(n) ->
      if n < 0 or n > 130 { Err(OutOfRange) }
      else { Ok(n) }
  }
```

`Result[AgeError, Int]` reads as "an `Int` or an error of type `AgeError`". The error is in the first parameter and the successful value in the second, the opposite of the convention some languages use — kaikai follows Haskell on this point.

Three patterns you'll see often:

- **Chain a failure with `!`.** If you have an expression that returns `Result[E, A]` and you want "if it fails, propagate the error; otherwise continue with the value", you write `expr!`. You saw this in §2.3.
- **Higher-order functions**: `option.map`, `option.and_then`, `result.map_err`. We cover them in chapter 6.
- **Convert between the two**: `option.ok_or(error)` takes an `Option[a]` and an error of type `e` and returns a `Result[e, a]`. Useful when the information lost by `None` isn't enough.

`Option` and `Result` are sum types like any other. We've separated them from chapter 5 because their role in day-to-day design is central — you'll use them before you start declaring your own sum types.

4.6 Tuples

A tuple is a **positional** aggregate: like a record, but without field names. The syntax is parentheses with elements:

```
let point2d = (3, 4)
let trio    = ("Ada", 30, true)
```

The type is written the same way: `(Int, Int)`, `(String, Int, Bool)`.

kaikai admits tuples of **arity 2 to 4**. There are no 1-element tuples — a single parenthesis, `(e)`, is grouping, not a tuple. And `(a, b, c, d, e)` is a parse error.

Why the bound? Because long tuples are unreadable. If you're hitting 5 elements, almost always what you wanted was a record with named fields.

To take a tuple apart, destructuring:

```
fn divmod(a: Int, b: Int) : (Int, Int) = (a / b, a % b)

fn main() {
  let (quotient, remainder) = divmod(17, 5)
  println("17/5 = #{quotient}, remainder #{remainder}")
}
```

Internally, tuples are sugar over three records from the stdlib: `Pair[A, B]` for arity 2, `Triple[A, B, C]` for 3, `Quad[A, B, C, D]` for 4. The declaration

```
let p = (1, 2)
```

is exactly equivalent to

```
let p = Pair { fst: 1, snd: 2 }
```

This is useful to know when you see a type `Pair[Int, String]` in a stdlib signature: now you know it's the same as `(Int, String)`, and you can destructure it with `let (a, b) = p`.

Tuple or record?

When in doubt, **use a record**. Tuples are convenient for cases where names don't add — the result of `divmod`, where "the first value is the quotient and the second the remainder" is information the reader already gets from the function name. Or when the aggregate lives one step:

```
xs |> map((e) => (e.name, e.age))
```

But as soon as the same aggregate appears more than once, or crosses a module boundary, or its shape is something the reader can't deduce, prefer a record. An `Employee` is much easier to read than a `(String, Int, Bool, String, Int)`.

Exercises

4.1. Define a record `Book` with fields `title`, `author` and `pages`. Write a function `fn short(b: Book) : Bool` that returns `true` if the book has fewer than 200 pages. Build two books and try the function.

4.2. Write `fn maximum(xs: [Int]) : Option[Int]` returning the largest element of the list, or `None` if the list is empty. Use recursion and `match`.

4.3. Rewrite `fn parse_age` from §4.5 so it distinguishes **three** errors: not numeric, negative age, age > 130. Use a sum type with three constructors and a `Result`.

4.4. Define a function `fn split(xs: [Int]) : ([Int], [Int])` returning a tuple with the evens and odds of `xs`, each in original order. Why a tuple and not two return values? And why a tuple and not a record?

4.5. Given a `[Point]` (with `Point = { x: Int, y: Int }`), write `fn center(ps: [Point]) : Option[Point]` returning the average of coordinates, or `None` if the list is empty. Hint: you'll need two passes or an accumulator, and `int_to_real` to average — but note the final result has to be a `Point` with `Int` fields, so you'll also need `real_to_int` (or settle for the integer part).

4.6. On paper, without writing code, draw what happens in memory when you run these three lines:

```
let xs = [1, 2, 3]
let ys = [0, ...xs]
let zs = [99, ...xs]
```

How many lists were created? How many elements were copied? Are there cells that `xs`, `ys`, `zs` share? The answer helps you understand why immutability isn't expensive when the structures are linked.

Chapter 5 · Sum types, unions, and `match`

This is the chapter where, if you come from an imperative language, you change how you model data. Sum types and the `match` that goes with them are not an exotic construction: they are the tool that replaces half of the class hierarchies, flag-based enums, `instanceof` chains and visitor patterns you've probably been writing. Once you have them in hand, you don't want to give them up.

The chapter goes from less to more. We start with basic sum types, pass through type recursion, see `match` with all its forms, reach unions of existing types — an idea uncommon in mainstream languages — and close with a complete arithmetic expression evaluator with typed errors.

5.1 Sum types with `|`

A **sum type** declares that a value can be one of several distinct constructors. The syntax is direct:

```
type Color = Red | Green | Blue
```

`Color` is a type. `Red`, `Green`, `Blue` are its three **constructors**. A value of type `Color` is exactly one of the three. There is no fourth hidden value, no `null`, no "unknown color". The type says it all.

If you come from a language with `enum`, this looks similar — with two differences: constructors **can carry data**, and the compiler **verifies that you cover them all** when you decide based on the constructor.

Start with the first. A constructor can take positional parameters, like a record with numbered fields:

```
type Shape
  = Circle(Real)
  | Rectangle(Real, Real)
  | Triangle(Real, Real, Real)
```

`Circle(2.0)` is a `Shape` value with a real inside. `Rectangle(3.0, 4.0)` is another `Shape` with two values. `Triangle(3.0, 4.0, 5.0)` is a third. All three fall under the same `Shape` type, but the compiler knows which is which and forces us to handle them separately when we consume them.

```
fn area(s: Shape) : Real =
  match s {
    Circle(r)       -> 3.14159 * r * r
    Rectangle(w, h)  -> w * h
    Triangle(a, b, c) -> {
      let s = (a + b + c) / 2.0
      s
    }
  }
```

`match` is an expression that decides based on the constructor. Each arm is a pattern followed by `->` and the expression that arm produces. The patterns unpack the data at the same time: in `Circle(r)`, `r` is the variable bound to the `Real` that came inside the constructor. There is no separate cast or indexed access; the pattern does the work.

Three details you'll use all the time:

- **The compiler checks exhaustiveness.** If you remove the `Triangle(...)` arm and the compiler knows `s : Shape` can be a triangle, it doesn't compile. This saves you from subtle bugs when you add a new constructor: every `match` that doesn't cover it becomes a compile error pointing exactly at the spot needing attention.
- **Constructors are names like any other.** When you declare `type Color = Red | Green | Blue`, kaikai creates four symbols: `Color` is the type, and `Red`, `Green`, `Blue` are simultaneously types (each with one inhabitant) and values. In chapter 9 we'll see how this lets you extend a sum type with protocols defined for individual constructors.
- **The `|` operator always means union of types.** We'll see more in §5.5: `Color = Red | Green | Blue`, `EvalError = ArithError | EnvError`, and `Result = Ok(Int) | Err(String)` are the same construction. The difference between "declaring new types" and "composing existing types" is decided by the compiler inspecting whether the names on the right are already declared.

5.2 Constructors with and without payload

You saw it above; let's pin it down. A constructor can:

- **Carry no data** (`Red`, `Green`, `Blue`). It is a unique value of the type, with no parameters.
- **Carry one or more positional values** (`Circle(Real)`, `Rectangle(Real, Real)`). The constructor is applied like a function to the data to produce the value.

There is no hard limit on how many values a constructor can carry — the grammar accepts them all — but for ergonomics, if you go past three or four positional values it pays to declare a record and put it in the payload. The difference between

```
type Event
  = Login(String, String, Int, Bool)
  | Logout(String, Int)
```

and

```
type LoginData = { user: String, ip: String, timestamp: Int, success: Bool }
type LogoutData = { user: String, timestamp: Int }
type Event = Login(LoginData) | Logout(LogoutData)
```

is that the second reads better wherever you build or destructure an event. The practical rule: if the data names itself by position — a point is `(Real, Real)` — keep it positional. If the data forces the reader to remember the order, use a record.

5.3 Recursion in types

This is where it gets powerful. A constructor of a sum type can mention the type itself in its fields. That gives you trees, lists, small graphs, and the representation of any nested structure.

A binary tree:

```
type Tree
  = Leaf
  | Node(Int, Tree, Tree)
```

`Tree` is `Leaf` (no data) or `Node` with an integer and two subtrees. The subtrees are `Tree` themselves, so they can recursively be `Leaf` or `Node`, to whatever depth you want.

A function over a tree is also recursive: the base case is `Leaf`, the recursive case descends into the children.

```
fn height(t: Tree) : Int {
  case Leaf          -> 0
  case Node(_, l, r)  -> 1 + max_int(height(l), height(r))
}
```

The pattern `Node(_, l, r)` ignores the node's data (we don't care about it for height) and binds the two subtrees to `l` and `r`. Then we recurse on each and take max plus one.

The same applies to an arithmetic expression AST:

```
type Expr
  = Lit(Int)
  | Add(Expr, Expr)
  | Mul(Expr, Expr)
  | Neg(Expr)

fn eval(e: Expr) : Int =
  match e {
```

```
  Lit(n)    -> n
  Add(a, b) -> eval(a) + eval(b)
  Mul(a, b) -> eval(a) * eval(b)
  Neg(x)    -> -eval(x)
}
```

Building an expression is transparent: `Mul(Neg(Add(Lit(2), Lit(3))), Lit(4))` is the literal representation of `-(2 + 3)

- `4, which evaluates to −20`. The tree and the code that walks it write themselves once you know how to look at it this way.

This replaces class hierarchies with visitor patterns in OO languages. The key difference: in kaikai, adding a new node to `Expr` is one line, and every `match` over `Expr` in the codebase becomes a compile error pointing at the spots that need attention. In a visitor pattern, you add a new method in each subclass and the compiler does not help you avoid forgetting any.

5.4 `match`: patterns, guards, exhaustiveness

You've seen `match` in action. Now we formalize.

A `match` takes a **scrutinee expression** (what's being inspected) and a series of **arms**, each with a **pattern** followed by `->` and the expression that arm produces. The arm whose pattern matches the value of the scrutinee is the one that runs; its value is the value of the `match`.

Patterns kaikai accepts:

- **Literals**: `0`, `"hi"`, `true`. Match the exact value.
- **Constructors**: `Some(x)`, `Lit(n)`, `Add(a, b)`. Match the constructor and bind the data to variables.
- **Records**: `Point { x, y }`, `Point { x: 0, y: _ }`.
- **Lists**: `[]`, `[h, ...t]`, `[only]`, `[first, second, ...]`.
- **Wildcard**: `_`. Matches anything, binds nothing.
- **Variable**: any undeclared identifier. Matches anything and binds the value to that variable.

Patterns nest: `Some(Point { x, y })` matches a `Some` that contains a `Point`, unpacking `x` and `y` in one pass.

Guards

A pattern can be followed by `if` and a condition — a **guard** — that's evaluated after the structural match. If the guard is false, the arm doesn't apply and the next arm is tried:

```
fn sign(n: Int) : String =
  match n {
    0          -> "zero"
    k if k > 0  -> "positive"
```

```
  _         -> "negative"
}
```

The pattern `k if k > 0` matches any integer, binds it to `k`, then evaluates `k > 0`. If true, runs the arm; otherwise moves on. The final `_` arm has no guard and matches everything else — integers that are neither zero nor positive.

Guards are convenient but don't participate in the exhaustiveness check: the compiler can't know that `k > 0` and `k < 0` complement each other, so it needs a final guardless pattern that covers "everything else". If you omit it, it doesn't compile.

Exhaustiveness

The compiler verifies that **every possible inhabitant** of the scrutinee's type is covered. If your `Expr` has four constructors and your `match` covers three, it doesn't compile, and the message doesn't stop at "something missing": it tells you what's missing, what's covered, and how to fix it:

```
error: non-exhaustive match on Expr: missing Neg
  --> eval.kai:12:3
   |
12 |   match e {
   |   ^
  = note: missing variant: `Neg`
  = note: covered: Lit, Suma, Mul
  = help: add an arm `Neg -> ...` or a wildcard `_ -> ...`
```

The wildcard `_` covers everything not covered by the preceding arms, so a `match` with a final `_` is trivially exhaustive. But using `_` as a final arm instead of enumerating the cases is a way of **silencing** the compiler: when you add a new constructor to `Expr`, the `match`es with a final `_` will absorb it without complaint, and you'll lose the warning.

Practical rule: use `_` only when you really don't care about distinguishing the rest. If there are three cases and all three matter, write the three.

5.5 Unions of existing types

So far every `|` we've seen had **new names** on the right — `Red`, `Green`, `Lit`, `Circle` — that kaikai auto-declares as constructors. But the operator doesn't require new names. If the names on the right are **already declared as types**, kaikai builds a **union** that can carry any value of those types.

This is the key tool for composing errors across layers:

```
type IdentityError = AccountNotFound | KycExpired | Frozen
type AuthError     = InsufficientBalance | OverDailyLimit

type QueryError = IdentityError | AuthError
```

`QueryError` is the union of the two prior types. A value of `QueryError` is **any** value of `IdentityError` or **any** value of `AuthError`. There is no new wrapper: `AccountNotFound` was already a valid value, and now it is also a valid value of `QueryError`.

That step is called **implicit upcast**: a variable typed `IdentityError` fits where `QueryError` is expected, no conversion:

```
let id_err : IdentityError = AccountNotFound
let qe : QueryError       = id_err     # OK, no ceremony
```

This is what other languages force you to do with wrapper constructors (`QEIdentity(IdentityError)`), `From` impls, or manual conversions with `map_err`. In kaikai, none of those three things exist: the compiler knows `IdentityError` is a component of `QueryError` and rewrites the upcast for you.

Pattern matching over unions

A `match` over a union value has two flavors that kaikai lets you **mix freely**.

The first, already familiar, enumerates every constructor individually:

```
fn describe(e: QueryError) : String =
  match e {
    AccountNotFound     -> "id: account not found"
    KycExpired          -> "id: KYC expired"
    Frozen              -> "id: frozen"
    InsufficientBalance -> "auth: insufficient balance"
    OverDailyLimit      -> "auth: over daily limit"
  }
```

This works, but is tedious for big unions. And worse, if a component grows — you add `RegulatoryHold` to `AuthError` — the `match` becomes a compile error in five places instead of one.

The second flavor is the **narrowing pattern** `bind : ComponentType`, which matches when the value belongs to the named component and binds it under that narrower type:

```
fn describe(e: QueryError) : String =
  match e {
    ie : IdentityError -> "id: " ++ id_str(ie)
    ae : AuthError     -> "auth: " ++ auth_str(ae)
  }
```

Each arm delegates to a function that knows that specific component. If `AuthError` grows, only `auth_str` changes. The `match` over `QueryError` doesn't notice.

The two forms also mix in a single `match` when you want to extract a specific case and delegate the rest:

```
match e {
  AccountNotFound    -> "id: specifically, account missing"
  ie : IdentityError -> "id: " ++ id_str(ie)      # KycExpired, Frozen
```

```
  ae : AuthError     -> "auth: " ++ auth_str(ae)
}
```

The compiler treats narrowing arms as if they covered every constructor of their component, so exhaustiveness closes correctly.

A deliberate limitation: upcast doesn't chain

There's a kaikai rule worth knowing. The implicit upcast **works only one step**:

```
type IdentityError = AccountNotFound | KycExpired
type AuthError     = OverLimit | DailyCap
type QueryError    = IdentityError | AuthError
type RoutingError  = NotFound | ServerDown
type AppError      = QueryError | RoutingError

fn handle_app(e: AppError) : String = "..."
```

And a value of the innermost type:

```
let id : IdentityError = AccountNotFound
handle_app(id)      # ERROR: IdentityError is not a direct
                    # component of AppError.
```

Although every `IdentityError` is logically an `AppError` (through `QueryError`), the compiler doesn't search that chain. To go from `IdentityError` to `AppError` you must write the intermediate step explicit:

```
let q : QueryError = id        # one step: IdentityError ⟶ QueryError
handle_app(q)                  # another step: QueryError ⟶ AppError
```

This is deliberate. Chained subtyping makes type inference brittle and error messages hard to phrase. The "one step" rule keeps kaikai with a predictable type system whose messages point to the right place.

5.6 Errors as unions, no wrappers

Unions are so useful for errors that the pattern is worth naming. When a function can fail in several ways and those ways break down into clear categories, the natural pattern is:

1. Each category is a small sum type.
2. The composite error is the union of the categories.
3. Each function returns `Result[CompositeError, T]`.
4. The `!` operator propagates the error of any layer up to the composite `Result`, via the implicit upcast.

```
fn check_identity(req: Req) : Result[IdentityError, Account] = ...
fn check_auth(c: Account) : Result[AuthError, Approved] = ...
```

```
fn query_balance(req: Req) : Result[QueryError, Balance] = {
  let acc = check_identity(req)!    # IdentityError <: QueryError
  let app = check_auth(acc)!        # AuthError <: QueryError
  Ok(load_balance(app))
}
```

Each `!` unpacks the `Ok` and propagates the `Err` with the correct upcast to the outer `Result[QueryError, _]`. Zero wrappers, zero `map_err`, zero `From`. The signature of `query_balance` documents exactly which errors it can emit, and the compiler ensures every one of them is covered when something consumes the result.

Compare it with the typical imperative version:

```
# Python: nothing in the return type tells you what may fail
def query_balance(req):
    acc = check_identity(req)   # may throw AccountNotFound, KycExpired...
    app = check_auth(acc)       # may throw InsufficientBalance...
    return load_balance(app)
```

The caller of `query_balance` in Python has to read the inner functions' code — or the docs, if any — to know which exceptions to expect. In kaikai the signature tells them.

5.7 Case study: evaluator with typed errors

We close the chapter with an integrative case. We'll build an evaluator of arithmetic expressions with three categories of error:

- **Arithmetic**: division by zero, square root of a negative.
- **Environment**: undefined variable.

The two form a union, `EvalError`, and the evaluator returns `Result[EvalError, Real]`. Full code in `examples/ch05/05_evaluator.kai`; here we walk through the interesting parts.

The AST

```
type Expr
  = Lit(Real)
  | Var(String)
  | Add(Expr, Expr)
  | Mul(Expr, Expr)
  | Div(Expr, Expr)
  | Sqrt(Expr)
```

Six constructors, two of them recursive. `Var(String)` introduces something new compared to the chapter 1 evaluator: expressions can now reference variables, so the evaluator needs an **environment** that associates names with values.

The errors

```
type ArithError = DivByZero | NegativeSqrt(Real)
type EnvError   = Undefined(String)
type EvalError  = ArithError | EnvError
```

Three types: two categories and the union. Four constructors in total, distributed over categories that make sense on their own. `NegativeSqrt(Real)` carries the value that was passed to sqrt — useful information for the final error message. `Undefined(String)` carries the name of the missing variable.

The environment

```
type Env = [(String, Real)]

fn lookup(env: Env, name: String) : Result[EvalError, Real] {
  case [], _                      -> Err(Undefined(name))
  case [(k, v), ...], n when k == n   -> Ok(v)
  case [_, ...rest], n              -> lookup(rest, n)
}
```

`Env` is an alias for a list of pairs — linear lookup, enough for a toy evaluator. `lookup` is written in the **multi-clause form** of chapter 6: each `case` lists one pattern per argument separated by comma, with an optional `when` for guards. Three cases:

- Empty list: variable wasn't there; return an error.
- Head with the key we're looking for: success.
- Any other head: continue with the tail.

Notice the `Err` is of type `EvalError`, not `EnvError` — but `Undefined(name)` constructs as `EnvError` and the implicit upcast promotes it to `EvalError` at the return site, no explicit conversion.

`eval` (next) uses the `match`-with-wrapper form because it dispatches on a sum type with many constructors and the form `match e { ... }` reads better when the discriminator is a single well-marked argument. The two forms coexist in the same file without tension.

The evaluator

```
fn eval(env: Env, e: Expr) : Result[EvalError, Real] =
  match e {
    Lit(n)    -> Ok(n)
    Var(name) -> lookup(env, name)
    Add(a, b) -> {
      let va = eval(env, a)!
      let vb = eval(env, b)!
      Ok(va + vb)
    }
    Mul(a, b) -> {
      let va = eval(env, a)!
      let vb = eval(env, b)!
      Ok(va * vb)
    }
```

```
  Div(a, b) -> {
    let va = eval(env, a)!
    let vb = eval(env, b)!
    if vb == 0.0 { Err(DivByZero) } else { Ok(va / vb) }
  }
  Sqrt(x) -> {
    let v = eval(env, x)!
    if v < 0.0 { Err(NegativeSqrt(v)) } else { Ok(sqrt(v)) }
  }
}
```

Each `match` case corresponds to a constructor of `Expr`. Recursive cases use `!` to evaluate subtrees and propagate any error that appears; cases that can fail locally (division by zero, negative sqrt) construct an `Err` of the appropriate type and let the upcast promote it.

The `!` operator appears several times and is worth reading carefully. `eval(env, a)!` means: if `eval(env, a)` returns `Ok(v)`, bind `va` to `v`; if it returns `Err(e)`, **exit immediately** from the current function returning that `Err`.

Lo que `!` does internally

`!` is a **return** in disguise. The line

```
let acc = check_identity(req)!
```

is exactly equivalent to:

```
let acc = match check_identity(req) {
  Ok(x)  -> x              # unpack and continue
  Err(e) -> return Err(e)     # exit query_balance with that error
}
```

For `Option[A]` the desugar is analogous: `Some(x)` unpacks, `None` triggers `return None`.

Three things to pin down:

- **It's not a `panic` or an exception.** The program does not abort; the current function ends normally returning the `Err` or `None` to its caller. Control flows out **one level only**, no further.
- **It only works if the current function returns a compatible type.** If your function returns `Int`, you can't use `!` on a `Result` inside — the `return Err(e)` has nowhere to land. The compiler tells you so clearly.
- **The upcast happens at the `return`.** When `check_identity(req)` returns `Result[IdentityError, _]` but `query_balance` declares `Result[QueryError, _]`, the `return Err(e)` applies the upcast `IdentityError <: QueryError` on the way out. That's why `!` and union errors compose so well: each level of the cascade absorbs the error of the layer below without writing conversions.

If you come from Rust, this is exactly the `?` operator. If you come from Swift, it's what `try` with `throws` does. If you come from Haskell, it's the `do`-notation over `Either` collapsed into a single symbol.

Printing the result

```
fn describe(e: EvalError) : String =
  match e {
    DivByZero       -> "division by zero"
    NegativeSqrt(v) -> "sqrt of negative (" ++ real_to_string(v) ++ ")"
    Undefined(name) -> "undefined variable: " ++ name
  }

fn print_result(r: Result[EvalError, Real]) : Unit =
  match r {
    Ok(v)  -> println("ok: " ++ real_to_string(v))
    Err(e) -> println("error: " ++ describe(e))
  }
```

`describe` consumes an `EvalError` enumerating its three constructors. Here `match` enumerates because we want specific messages per constructor; in other cases — when the logic differs per category — we'd use narrowing.

The main program assembles the environment, evaluates several expressions, and prints them:

```
fn main() {
  let env : Env = [("x", 9.0), ("y", 4.0)]

  print_result(eval(env, Add(Var("x"), Var("y"))))    # ok: 13
  print_result(eval(env, Sqrt(Var("x"))))             # ok: 3
  print_result(eval(env, Div(Lit(10.0), Lit(0.0))))   # division by zero
  print_result(eval(env, Sqrt(Lit(0.0 - 1.0))))       # negative sqrt
  print_result(eval(env, Var("z")))                   # undefined variable
}
```

The pretty thing about this case: every error is documented in the type of the evaluator. If later you want to add an **TypeError** (say, attempting to add two non-numbers), you declare the new category, include it in `EvalError`, and the compiler walks you to every place that needs to adapt.

Exercises

5.1. Define `type Maybe[a] = Just(a) | Nothing` (another form of `Option`) and write `fn or_else[a](m: Maybe[a], default: a) : a` returning the value if `Just`, the default if `Nothing`. Note: `Nothing` is also a kaikai primitive type (the bottom type from chapter 3); use `Empty` or another name if shadowing bothers you.

5.2. Extend the §5.7 evaluator to support subtraction: add `Sub(Expr, Expr)` to the AST, add the matching arm in `eval`, and verify that `2 - 3` evaluates to `-1`. How many lines did you have to change? How many places did the compiler walk you to?

5.3. Define a binary decision tree:

```
type Decision[a]
  = Leaf(a)
  | Question(String, Decision[a], Decision[a])
```

where a `Question` has text, a "yes" branch and a "no" branch. Write `fn apply[a](d: Decision[a], answers: [(String, Bool)]) : Option[a]` walking the tree following the answer to each question, returning `Some` with the final decision if it reaches a `Leaf`, or `None` if some question has no answer.

5.4. Take the union example with `IdentityError` and `AuthError`. Add a third component `RoutingError` with two constructors. Rewrite the `match` with narrowing so the code stays equally short. Then add a new constructor to `AuthError` (say `Blocked`) — what happens to your code? How many places does the compiler intervene?

5.5. Create a sum type for a chat message: `type Message = Text(String) | Image(String, Int, Int) | Audio(String, Int)` (path, dimensions or duration). Write a function `fn description(m: Message) : String` returning a human description, and a function `fn heavy(m: Message) : Bool` returning `true` if the message is image or audio. The second function should use narrowing in the `match`, not enumeration.

5.6. On paper, draw the evaluation tree the §5.7 evaluator builds when processing the expression `Mul(Add(Var("x"), Lit(2.0)), Var("y"))` with environment `[("x", 9.0), ("y", 4.0)]`. How many calls to `eval`? How many to `lookup`? In what order? This helps you understand why the `!` operator cuts execution as soon as something fails: three cascaded calls, one line per level.

Chapter 6 · Functions and pipelines

So far you've been seeing functions as you needed them: the first ones in the tour, those of chapter 3 with their two body forms, the recursive ones of chapter 5 that decompose sum types. This chapter puts everything together and adds what was missing: how to declare functions carefully, how to write lambdas, what higher-order functions are, and the four pipe operators kaikai uses to chain transformations.

We will also look in detail at something kaikai promises and very few languages guarantee seriously: **tail-call elimination**. That guarantee is what lets you write loops without `for` or `while` and sleep at night.

6.1 Declaration

A function is declared with `fn`, typed parameters, return type, and a body separated by `=`:

```
fn double(x: Int) : Int = x * 2
```

The body can take **three forms**. The first is a single expression, like the one above.

The second is a block `{ … }` with intermediate `let`s and the implicit final expression:

```
fn square_plus_one(x: Int) : Int = {
  let square = x * x
  square + 1
}
```

The third is **arms with patterns**, where the function decides what to do based on the shape of its arguments. Each arm starts with `case`, followed by a pattern, an arrow `->`, and the expression that case produces:

```
fn sign(n: Int) : String {
  case 0          -> "zero"
  case k when k > 0 -> "positive"
  case _          -> "negative"
}
```

This is exactly equivalent to:

```
fn sign(n: Int) : String =
  match n {
    0         -> "zero"
    k if k > 0 -> "positive"
    _         -> "negative"
  }
```

just without the `match` wrapper. When the function has **multiple parameters**, the patterns are listed comma-separated, one per argument:

```
fn divide(a: Real, b: Real) : Result[Error, Real] {
  case _, 0.0 -> Err(DivByZero)
  case a, b   -> Ok(a / b)
}
```

A language rule: inside a function's `{…}` block, **either all arms are `case` arms or all are statements**. Mixing is a parse error. If you need setup before discriminating, wrap a `match` in the short form or extract a helper.

The practical rule for choosing among the three:

- **Short body with `=`** when the function is a direct expression, no intermediate steps.
- **Block with `{…}`** when there are intermediate `let`s or several visually-separated steps.
- **Multi-clause with `case`** when the function dispatches primarily on the shape of its arguments. The natural form for many recursive functions and for dispatchers over sum types.

Three notes worth pinning down:

- **Parameter annotations: mandatory.** kaikai infers types on local bindings (`let x = 5`), but not on function signatures. Each parameter must say its type.
- **Return type: required on public functions, recommended always.** The compiler can infer it on local functions, but the signature documents the contract and makes type errors reported at the right place. Annotate.
- **Functions with effects: the effect goes after `/`.** A function that writes to stdout is `: Unit / Stdout`, and chapter 12 covers it carefully. For now, knowing that if your function calls `println`, its signature says so.

`main` is the only function where the return type is optional. If you omit it, kaikai assumes `Unit`.

6.2 Lambdas

A **lambda** is an anonymous function, an expression that evaluates to a function value. kaikai gives you three forms to write them:

```
# Form 1 — single-argument arrow
let square = (x) => x * x
```

```
# Form 2 — multi-argument arrow
let sum = (a, b) => a + b

# Form 3 — placeholder, implicit unary lambda
xs |> list.filter(. > 0)
```

The first two are interchangeable; choice is style. The third is **sugar** that only applies when the context **expects a function** — the second argument of `list.filter`, for example, is `(Int) -> Bool`, and the compiler converts `. > 0` into `(n) => n > 0`.

The placeholder rules:

- `.` only works where a function is expected. Outside that, it's a compile error.
- Unary functions only. For two or more arguments, explicit arrow.
- Multiple `.` occurrences in the same expression refer to the **same value**. For example, `xs |> list.map(. * .)` squares.
- Field access works: `people |> list.map(.name)` projects the `name` field of each element.

When to use which? My suggestion:

- **Placeholder** when the lambda is trivial and inside a pipe. `xs |> list.filter(. > 0)` reads at a glance.
- **Arrow** when the lambda has several steps or when the argument appears in non-obvious places.
- **Named function** when the lambda is used more than once, or when the name documents the intent.

Lambdas are **first-class values**: you bind them with `let`, pass them as arguments, return them from functions, store them in records. That's what makes higher-order functions natural.

6.3 Higher-order functions

A **higher-order function** is one that takes or returns another function. That's the whole definition. What's interesting isn't the name: it's what it lets you do.

The simplest case is a function that applies another twice:

```
fn twice[a](f: (a) -> a, x: a) : a = f(f(x))
```

`f` is the first parameter, of type `(a) -> a` — any function from `a` to `a`. `twice(plus_one, 5)` computes `plus_one(plus_one(5))` = `7`. The function is **generic** over `a`: it works with `Int`, `String`, any type as long as `f` has the same input type as output. The annotation `[a]` after the name declares the type parameter.

A more interesting case is a function that **returns** another — a *closure*:

```
fn add_n(n: Int) : (Int) -> Int = (x) => x + n
```

`add_n(10)` returns a new function that adds 10 to its argument. The trick is that the lambda **captures** `n` from the context where it was created. Once returned, that function has a copy of `n` inside:

```
let plus_ten = add_n(10)
plus_ten(7)      # 17
plus_ten(100)    # 110
```

And the classic composition:

```
fn compose[a, b, c](f: (b) -> c, g: (a) -> b) : (a) -> c =
  (x) => f(g(x))
```

`compose(f, g)` is the function that first applies `g`, then `f`. Three type parameters (`a`, `b`, `c`) because chained functions touch three distinct types in general.

Higher-order functions are the main tool to abstract **over the what-to-do**. Instead of writing `for each element, do X` and `for each element, do Y`, you write `for each element, do F`, where `F` is a parameter. That's how `list.map`, `list.filter`, `list.fold` are born — all the functional workhorses.

6.4 Pipes: `|>`, `|`, `||`, `|?`

kaikai has four pipe operators. All four are distinct and each communicates a specific intent.

`|>` — apply

`xs |> f` is exactly `f(xs)`. The operator takes the left-hand side and puts it as the **first argument** of the right-hand call. If `f` has more arguments, they go in the parens:

```
xs |> list.sum                 # ≡ list.sum(xs)
xs |> list.filter(is_even)     # ≡ list.filter(xs, is_even)
xs |> list.map((n) => n * 2)   # ≡ list.map(xs, (n) => n * 2)
```

But there's a useful detail: sometimes the piped value **doesn't want** to go as the first argument. For example, a division function where what you're piping is the divisor, not the dividend. kaikai lets you indicate the exact position with an underscore `_`:

```
fn divide(a: Int, b: Int) : Int = a / b

100 |> divide(_, 4)      # ≡ divide(100, 4) = 25
100 |> divide(1000, _)   # ≡ divide(1000, 100) = 10
```

The `_` is the **slot** where the LHS lands. Without `_`, the LHS goes to the first argument — the short form of `f(_, a, b)`. With `_`, wherever you put it. This lets you write natural pipelines even when stdlib functions weren't designed with the "main argument" in first position.

`|` — map

`xs | f` is exactly `list.map(xs, f)`. A specific operator for mapping over lists:

```
let doubles = xs | (n) => n * 2     # = list.map(xs, (n) => n * 2)
```

Why have `|` if `|>` already covers the case? Because `xs | f` reads as "xs processed with f", which is exactly what a map is. The shorter form makes list-transformation pipelines read like declarative sequences.

`||` — flat-map

`xs || f` desugars directly to `list.flat_map(xs, f)`, just like `|` desugars to `list.map(xs, f)`. The three following forms are equivalent:

```
let extended = [10, 20, 30] || neighbors              # sugar
let extended = list.flat_map([10, 20, 30], neighbors)     # direct call
let extended = [10, 20, 30] | neighbors |> list.concat    # map + concat
```

The first reads "expand each element". The second is the named operation. The third shows how flat-map is defined: map and concat. Use `||` to make the operation evident in a pipeline; use the direct call when you're not in a pipeline.

`|?` — filter

`xs |? p` is exactly `list.filter(xs, p)`. Filters the list keeping the elements where the predicate is true:

```
fn is_even(n: Int) : Bool = n % 2 == 0

[1, 2, 3, 4, 5, 6] |? is_even     # = [2, 4, 6]
```

The predicate is any expression the context admits as `(a) -> Bool`: a function name, an arrow, a block lambda:

```
xs |? is_even                 # name
xs |? (n) => n > 3              # arrow
xs |? { n -> n % 3 == 0 }       # block
```

`|?` closes the family of list-specific pipes: `|` is map, `||` is flat-map, `|?` is filter. The three canonical list-transformation operations have their own operator.

Together

The four operators mix freely:

```
let total =
  orders
  |? is_pending
  | apply_discount
```

```
| amount_of
|> list.sum
```

Each step does one thing. The result reads left to right. No intermediate variables, no nested parens. This is the main reason kaikai has four operators and not one.

6.5 Trailing lambdas and other sugars

kaikai has several syntactic sugars you'll see in real code, mostly when you use higher-order functions. The main ones:

Trailing lambdas

When a function takes a lambda as its **last** argument, you can pull it out of the parens and put it in braces, with the syntax `{ param -> body }`:

```
list.map(xs) { n -> n * 2 }
list.filter(xs) { n -> n > 0 }
```

Equivalent to `list.map(xs, (n) => n * 2)`. Both forms are accepted; trailing is friendlier when the lambda's body is long.

When you're inside a pipeline with `|`, the block-lambda is even more compact. `|` already expects a function as its second argument, so you can write the body directly:

```
xs | { n -> n * 2 }          # equivalent to xs | ((n) => n * 2)
xs | { n -> n * n + 1 }
```

This reads almost like prose. It works for `|` and `||`, and mixes with the rest of the pipeline without noise.

Double trailing lambda

If the **two** last arguments are lambdas, both go in braces:

```
fn while_loop(cond: () -> Bool, body: () -> Unit) : Unit / e = ...

while_loop { @i < 10 } { i := @i + 1 }
```

This gives kaikai user-defined control flow. `while_loop` is an ordinary stdlib function; it just looks like a keyword.

None of these sugars introduce new semantics. They are alternative ways to write lambdas that the compiler desugars before type inference. Use them when the reading improves; ignore them when it doesn't.

6.6 Recursion and mandatory TCO

An important promise from kaikai: **every recursive call in tail position compiles to a loop**. No stack consumption. No risk of stack overflow from a long recursion.

Let's see what "tail position" means. A call is in **tail position** if it's the last thing the function does before returning. Compare these two versions of `sum`:

```
# NOT in tail position: the call leaves `h + ...` pending.
fn sum_naive(xs: [Int]) : Int {
  case []       -> 0
  case [h, ...t]  -> h + sum_naive(t)
}
```

Here, after `sum_naive(t)` returns, you still have to add `h`. The call **is not** the last thing; an operation remains. Each call consumes a stack frame.

```
# In tail position: the last thing in each recursive arm is
# **only** the call, no pending operation.
fn sum_tco_loop(xs: [Int], acc: Int) : Int {
  case [], a       -> a
  case [h, ...t], a  -> sum_tco_loop(t, a + h)
}

fn sum(xs: [Int]) : Int = sum_tco_loop(xs, 0)
```

Here, in each recursive arm, `sum_tco_loop(t, a + h)` is **the last thing**. The sum `a + h` is evaluated first, passed as an argument, and then the call happens. When the call returns, the current function returns immediately too: nothing remains to do. The compiler detects this pattern and compiles to a loop, no new frame.

```
# This works without blowing the stack:
let many = [1..100_000]
sum(many)       # 5_000_050_000
```

The **accumulator** technique you see in `sum_tco_loop` is the standard way to convert a naive recursion into a tail recursion: you add an extra parameter that carries the partial result, and at the end you return it.

Why does this matter, really?

- **Without mandatory TCO, you couldn't replace `for` and `while`.** With limited stack, an iteration over a million elements would run out. Only with guaranteed TCO can you take the leap to programming with recursion.
- **It's a language guarantee, not an opportunistic optimization.** Some languages optimize TCO when they remember; kaikai promises it. If a recursive call is in tail, the compiler converts it. Period.
- **The compiler warns you if you think you wrote TCO but didn't.** There's a flag for that, so you don't find out by surprise when your program dies in production.

In practice, most recursive functions you write to process lists or trees will have the shape `match xs { [] -> base; [h, ...t] -> recursion }`. The naive version (with pending operation) is the first you write; the accumulator version is the one you keep. For more complex operations, the higher-order functions (`list.fold`, `list.reduce`) are already TCO-written and are almost always what you wanted.

6.7 Case study: transformation pipeline

Let's close with an integrative case. You have a list of orders from a store, with an id, an amount, and a status. You want the total to bill **considering only pending orders** and **applying a 10% discount on high amounts**:

```
type Status = Pending | Paid | Cancelled

type Order = {
  id: Int,
  amount: Int,
  status: Status,
}

fn is_pending(o: Order) : Bool =
  match o.status {
    Pending   -> true
    Paid      -> false
    Cancelled -> false
  }

fn apply_discount(o: Order) : Order =
  if o.amount >= 1000 {
    Order { ...o, amount: o.amount - o.amount / 10 }
  } else {
    o
  }

fn amount_of(o: Order) : Int = o.amount
```

Each small function does **one thing**. One decides if an order is pending, another applies a discount if applicable, another extracts the amount. None knows about the pipeline; all are useful on their own.

And the pipeline composes them:

```
let total =
  orders
  |? is_pending
  | apply_discount
  | amount_of
  |> list.sum
```

Five lines, four steps. Filter the pending, apply discount to each, extract the amount, sum everything. If later the client asks for "let's add a 100 fixed fee for orders > 5000", you add a small function and put it in the pipeline:

```
fn fee_if_large(o: Order) : Order =
  if o.amount > 5000 {
    Order { ...o, amount: o.amount + 100 }
  } else {
    o
```

```
}

let total =
  orders
  |? is_pending
  | apply_discount
  | fee_if_large
  | amount_of
  |> list.sum
```

One more line, zero coupling. That's the point of functional style: **small functions composed in linear-reading pipelines**. The complexity lives in each individual function; composition is trivial.

Compare with the typical imperative version:

```
total = 0
for o in orders:
    if o.status != "pending":
        continue
    amt = o.amount
    if amt >= 1000:
        amt -= amt // 10
    if amt > 5000:
        amt += 100
    total += amt
```

Works, but the calculation logic is tangled with the loop mechanics. Reordering transformations, adding a step, removing a step — operations that in the pipeline are one line, in the imperative version are a refactor.

If you come from Java, JavaScript, C# or Kotlin, you've seen something similar with their respective stream/iterator APIs. The difference is that in kaikai the pipeline is a syntactic construction of the language, not an API on top of the language. You don't need to wrap the list in a special object, no `.collect()` methods or terminators; pipes are as ordinary as `+` or `*`.

Exercises

6.1. Write `fn apply_n(f: (Int) -> Int, x: Int, n: Int) : Int` that applies `f` to `x` exactly `n` times. For example, `apply_n(plus_one, 5, 3)` should be `8`. Use tail recursion.

6.2. Define `fn compose3[a, b, c, d](f: (c) -> d, g: (b) -> c, h: (a) -> b) : (a) -> d` that composes three functions. Test with `compose3(square, plus_one, double)(2)`.

6.3. Rewrite the §6.7 pipeline using only `|>` (no `|`, `|?` or `||`). How many more characters? How does readability change?

6.4. Define a sum type `type Operation = Add(Int) | Multiply(Int) | Negate`. Write `fn apply_op(op: Operation, x: Int) : Int` that runs the operation on `x`, and `fn run_all(ops: [Operation], x: Int) : Int` that runs a sequence of operations starting with `x`. Hint: `list.fold` with `apply_op` flipped is a good path.

6.5. You have a list of strings with numbers: `["10", "abc", "20", "", "30"]`. You want the sum of the **valid numbers**, ignoring the rest. Build a pipeline using `list.flat_map` (or `||`) and `string_to_int : String -> Option[Int]`. Hint: a function `Option[a] -> [a]` helps — if it's `Some(x)` return `[x]`, if `None` return `[]`. That step is the natural flat-map.

Chapter 7 · Tests, properties, and benchmarks

So far you've been writing functions and looking at the output. That's a reasonable cycle while your program fits in your head, but it doesn't scale. As soon as your code grows past a few dozen lines — as soon as there are more than three functions calling each other — you stop being able to verify by eye that each change preserves behavior.

That's what tests are for. kaikai brings **three top-level constructs** dedicated to verification: `test`, `check`, and `bench`. All three live in the same file as the code they exercise, all three run via the `kai` driver, and all three are stripped out when you build a binary for production.

This chapter walks through the three, explains when to use which, and closes with a case study: a mini-evaluator with contractual tests, verified properties, and benchmarks measuring per-operation cost.

7.1 `test "..." { ... }` and `assert`

The simplest form is a test with a name and a body:

```
fn square(n: Int) : Int = n * n

test "square of zero" {
  assert square(0) == 0
}

test "small cases" {
  assert square(3) == 9
  assert square(7) == 49
}
```

`test` is a **top-level block** — it lives alongside `fn` in the same file, not nested in another function. Its name is a string literal the runner prints verbatim. Inside, you write assertions with `assert`: a `Bool` expression that must be `true`. If all the block's assertions pass, the test passes. If **one** fails, the test fails and the runner moves on to the next.

The test name should say **what is being tested**, not how. "square of zero" is good; "test 1" isn't.

`assert` also accepts an optional message with a comma:

```
test "valid ranges" {
  let n = classify(42)
  assert n > 0, "expected positive, got #{n}"
}
```

The message appears when the assertion fails. Useful when the expression you tested doesn't, by itself, tell you what was unexpected.

What the runner prints

```
$ kai test examples/ch07/01_basic_test.kai
  ok   base case
  ok   small cases
  ok   significant case

3/3 tests passed
```

If a test fails, the output changes to show it:

```
$ kai test examples/ch07/02_assert_fails.kai
  ok   double preserves positives
  FAIL broken test: assert will fail : assertion failed
  ok   this test still runs

2/3 tests passed
```

Three details worth recalling:

- **Tests run in declaration order.** Your file lists them top-to-bottom and the runner runs them in that order. No parallelism within a single file.
- **A failing test doesn't stop the rest.** The runner moves on and reports the final count.
- **`test` blocks don't end up in the production binary.** `kai run` and `kai build` discard them. They're only compiled and executed under `kai test`.

7.2 `kai test` and the short feedback loop

The command is straightforward:

```
$ kai test my_file.kai
```

Compiles the file in `--test` mode (which activates `test` blocks), produces a binary, runs it, and reports. The edit→run cycle takes a second or two on small files.

If you call `kai test` without a file name, there's no auto-discovery like `pytest` or `cargo test`. That's deliberate — kaikai doesn't have a standard project layout yet — but the flow is worth knowing: you point at the file, the runner runs everything that file and its imports declare with `test`.

Three practical recommendations you'll internalize within weeks:

- **Tests next to the code, in the same file.** Don't separate them into `tests/` or parallel files. When you touch a function, its tests are right there.
- **One test per aspect, not per line.** If your function has a base case, small cases, and an edge case, write three `test`s. If "small cases" has three examples, add them as three `assert`s in the same block.
- **Descriptive names.** The name appears in the runner's output every time you run the tests. `"validate email rejects spaces"` reads; `"test_3"` doesn't.

7.3 `check "..."` — properties

The tests you've seen so far check **fixed cases**: "for this input, I expect this output". This is what other languages call "example-based testing". It's the most common, but it has an obvious limit: it only tests what you write.

Properties flip the thing around. Instead of "for `square(7)` I expect `49`", you write "for every integer `n`, `square(n)` must be `>= 0`". The runner generates random values of `n` and verifies the property over each. If it finds a counterexample, it shows you; if it passes a hundred iterations without failure, it considers the property proved.

```
fn double(n: Int) : Int = n * 2

check "double is even" with n: Int {
  double(n) % 2 == 0
}

check "addition is commutative" with a: Int, b: Int {
  a + b == b + a
}

check "reverse of reverse" with xs: [Int] {
  list.reverse(list.reverse(xs)) == xs
}
```

`check "..." with name: Type { body }` declares a property. The `with` clause lists the parameters the runner generates randomly; the `body` is a `Bool` expression that must be `true`.

```
$ kai check examples/ch07/03_check_properties.kai
  double is even: 100 iter, OK
  addition is commutative: 100 iter, OK
  addition is associative: 100 iter, OK
  reverse of reverse: 100 iter, OK

4/4 checks passed
```

A hundred iterations per property is the default; each iteration generates fresh values. For `Int`, the default range is `[-50, 50]`. For `[Int]`, small lists. For records and sum types, the generator structures the components recursively.

When a property fails

```
check "all Ints are positive" with n: Int {
  n > 0
}
```

```
$ kai check false_property.kai
  all Ints are positive: counterexample at iter 1: n=-32

0/1 checks passed
```

The runner gives you the **exact counterexample** (`n = -32`) at the first failing iteration. That tells you three things:

- The property is false for some value of `n`.
- The concrete value.
- Which iteration failed (useful for reproducing with the same seed if you want to debug).

Unlike a `test` with a fixed case, where the test name is what diagnoses the failure, a `check` gives you the test case **along with** the report. You don't have to write it: you have it.

When to write a `check`

Properties are useful when you can **state a universal truth** about your code. Some common examples:

- **Inverses**: `decode(encode(x)) == x`, `parse(format(x)) == x`.
- **Idempotence**: `normalize(normalize(s)) == normalize(s)`.
- **Algebraic invariants**: commutativity, associativity, identity.
- **Conservation**: the `length` of the output equals that of the input, the sum is preserved, etc.
- **Monotonicity**: if `a < b`, then `f(a) < f(b)`.

If what you want to verify is "for input 7, output is 14", that's a `test`. If it's "for any input, what comes out is twice the value", that's a `check`.

7.4 `bench "..." { ... }` — measure, don't guess

The third construct is for **performance**. `bench` takes a block and measures how long it takes to execute, repeated many times for the average:

```
fn fib(n: Int) : Int =
  if n < 2 { n } else { fib(n - 1) + fib(n - 2) }

bench "arithmetic: 2 + 3 * 4" {
  2 + 3 * 4
}

bench "fib(10)" {
  fib(10)
```

```
}

bench "fib(15)" {
  fib(15)
}
```

```
$ kai bench examples/ch07/04_basic_bench.kai
  arithmetic: 2 + 3 * 4: 1000 iter / 7 ns/iter
  fib(10): recursion without memo: 1000 iter / 92 ns/iter
  fib(15): cost grows exponentially: 1000 iter / 1063 ns/iter
  list.sum [1..100]: 1000 iter / 4305 ns/iter

4 benches
```

Each bench runs 1000 iterations (configurable with `KAI_BENCH_ITERS`) and reports nanoseconds per iteration.

What matters about benchmarks isn't the absolute number — it depends on the machine and on what else is running — but the **comparison**. When you refactor a critical function, you run the bench before and after. When your pipeline starts to feel slow, you compare candidate function versions. The rule:

> ***Optimizing without measuring is guessing.***

If your code isn't slow, don't bench it. If it is slow, don't optimize without measuring first. `bench` is the tool that closes that cycle.

Three practical pieces of advice:

- **The body of the `bench` is what's measured.** If your setup is expensive and you don't want to include it, do it outside the block and leave only the operation to measure inside.
- **kaikai doesn't discard "pure" calls without observable effect.** `bench "fib(10)" { fib(10) }` actually computes `fib(10)` each iteration. In other languages with more aggressive optimizations you have to use tricks to avoid dead-code elimination; here you don't.
- **The absolute number is suggestive, not authoritative.** To compare, always measure on the same machine, in the same session, without other heavy loads running in parallel.

7.5 When to use which

You have the three tools. The decision boils down to a simple question:

Question	Tool
For this concrete input, does the output match my expectation?	`test`
For **every** input, does this invariant hold?	`check`
How much does this operation cost?	`bench`

The three complement each other. A serious project will have all three in the same file: tests for the contractual cases (the customer's, the edges, the ones that historically failed), checks for the algebraic invariants the code preserves, and benchmarks for the few critical functions where performance matters.

A note on writing order. The natural sequence is:

1. **Start with a `test`** — the concrete case for the feature you're working on. It's the easiest to write and the easiest to look at when something fails.
2. **Add `test`s** for edge cases as they show up.
3. **Move to `check`s** when you see a pattern in the tests: "all these cases are checking the same invariant, just with different data". That's a sign that a property should capture the general rule.
4. **Add a `bench`** when you start to notice slowness, or before an optimization refactor to have a baseline.

Not the other way around. Starting with a `check` when you don't yet know what properties you'll preserve leads to vague properties that pass by accident. Starting with a `bench` before performance matters is premature optimization. Tests first.

7.6 Case study: tests for a mini-evaluator

Closing with an integrative example: a small arithmetic expression evaluator with error handling, tested with the three tools. The complete code is in `examples/ch07/05_evaluator_tests.kai`; here we walk through the parts.

The AST and the evaluator

```
type Expr
  = Lit(Int)
  | Add(Expr, Expr)
  | Mul(Expr, Expr)
  | Div(Expr, Expr)

type EvalError = DivByZero(Int)

fn eval(e: Expr) : Result[EvalError, Int] = match e {
  Lit(n)    -> Ok(n)
  Add(a, b) -> { ... }
  Mul(a, b) -> { ... }
  Div(a, b) -> {
    let va = eval(a)!
    let vb = eval(b)!
    if vb == 0 { Err(DivByZero(va)) } else { Ok(va / vb) }
```

```
  }
}
```

It's a smaller cousin of the chapter 5 evaluator: four constructors, one error category (division by zero). Enough to show the flow.

Tests for contractual cases

```
test "literal" {
  assert must_yield(Lit(42), 42)
}

test "combined: 2 + 3 * 4 = 14" {
  assert must_yield(Add(Lit(2), Mul(Lit(3), Lit(4))), 14)
}

test "division by zero is an error" {
  assert must_fail(Div(Lit(10), Lit(0)))
}
```

Three tests documenting three behaviors. `must_yield` and `must_fail` are helpers wrapping the `match` over `Result` and returning `Bool`, so the `assert` stays readable.

```
4/4 tests passed
```

Checks for invariants

```
check "Lit(n) evaluates to n" with n: Int {
  must_yield(Lit(n), n)
}

check "Add(a, b) == Add(b, a)" with a: Int, b: Int {
  must_yield(Add(Lit(a), Lit(b)), a + b) and must_yield(Add(Lit(b), Lit(a)), a + b)
}
```

Two properties. The first says a literal evaluates to itself — a trivial invariant but important: if it failed, something is very wrong in the evaluator. The second checks that addition is commutative **through the evaluator**, not just at the integer-arithmetic level.

```
2/2 checks passed
```

A hundred iterations per property with randomly generated values. None failed. If in the future someone breaks commutativity — say, by adding a side effect to evaluating `Add` that depends on order — the `check`s detect the counterexample immediately.

Benchmarks for performance decisions

```
bench "literal" {
  eval(Lit(42))
```

```
}

bench "deep expression (3 levels)" {
  eval(Add(Mul(Lit(2), Lit(3)), Add(Lit(4), Mul(Lit(5), Lit(6)))))
}
```

Two benches: the cheap case (a literal) and a more complex case (three nesting levels). On my machine:

```
literal:                    1000 iter / 15  ns/iter
deep expression (3 levels):    1000 iter / 134 ns/iter
```

The second is ~9x more expensive than the first. That's the information you need if you later decide the evaluator is a bottleneck: you know what baseline you're measuring against.

What the file doesn't show

The mechanism is direct. The file declares functions, declares tests, declares checks, declares benches, all in the same place. Three commands process it:

```
$ kai test  examples/ch07/05_evaluator_tests.kai
$ kai check examples/ch07/05_evaluator_tests.kai
$ kai bench examples/ch07/05_evaluator_tests.kai
```

And `kai run` and `kai build` ignore the three constructs. The deployed binary loads neither tests nor checks nor benches — only production code.

That unification is what makes the model comfortable. There is no separate test project, no frameworks to import, no decisions about where to put each thing. The question "where are the tests for this function?" has only one possible answer: next to the function.

Exercises

7.1. Take a simple function you've already written — from earlier chapters or new — and write three tests for it: one with a typical case, one with an edge case, and one with an invalid input (if the type allows). Run `kai test` and verify all three pass.

7.2. Write `fn is_sorted(xs: [Int]) : Bool` returning `true` if the list is sorted ascending. Then write a `check` that verifies `is_sorted(list.sort(xs))` for any `xs : [Int]`. What if your `is_sorted` has a bug — say, accepts lists with one "skipped" element? The runner should give you a counterexample.

7.3. Go back to the §7.6 evaluator. Add a new constructor `Sub(Expr, Expr)` to `Expr` and the corresponding branch in `eval`. What tests break? Which tests should you add to cover the new case? Is there any **new property** worth writing as a `check` (e.g., `Sub(Lit(a), Lit(0)) == Lit(a)`)?

7.4. For an operation of your choice, write two implementations: a "naive" one and an "optimized" one. Write a `bench` for each. How many times faster is the optimized one? Does the difference justify the added complexity?

7.5. Look closely at this innocent-looking `check`:

```
check "concatenating lists preserves length" with xs: [Int], ys: [Int] {
  list.length(list.concat([xs, ys])) == list.length(xs) + list.length(ys)
}
```

What property does it express? Why is it trivial but useful? What should happen if someone (you, in six months, in a hurry) "optimizes" `list.concat` and breaks the property? Write the check, run it, then mentally try to break `list.concat` — at which iteration do you think the counterexample would appear?

Chapter 8 · Modules, imports, organizing code

Up to this point all your code has lived in a single file. That's the right way to start, but as soon as a program grows, a single file becomes a problem: readers get lost, changes clash, searches conflate logical modules that are physically tangled. The code has to be split into named pieces.

kaikai handles this with a deliberately simple system. **One file is one module.** There is no `module Foo` declaration. There are no special files that reopen a module from elsewhere. The module name is derived from the file path, and that's it. Above modules sits a second scale: a **project**, described by a `kai.toml`, that groups its modules and external dependencies. And above the project, a **package manager** resolves dependencies across projects.

This chapter walks through the three scales, from smallest to largest.

8.1 One file, one module

Create a file `arithmetic.kai` with a handful of functions:

```
pub fn double(x: Int) : Int = x * 2

fn helper(x: Int) : Int = x + 1

pub fn double_plus_one(x: Int) : Int = helper(double(x))
```

Three design decisions appear in these three lines:

- **`pub` marks what the module exports.** By default, a declaration is private to the file. Only the things marked `pub` are visible from another module that imports this one. This is the opposite of Java or C++, where public is the default and you have to remember to write `private`.
- **Module names need no declaration.** The file `arithmetic.kai` is the module `arithmetic`. Put it under a subdirectory like `util/arithmetic.kai` and it becomes the module `util.arithmetic`. The name is derived from the path relative to the project root.
- **Any `fn`, `type`, `effect`, or `let` can carry `pub`.** There is no visibility distinction between types and functions: a construct is visible outside the module iff it is marked `pub`.

Private functions are useful for breaking up a public one without polluting the consumer's namespace. In `arithmetic`, `helper` exists only inside the file; whoever imports the module never sees it.

8.2 `import` and qualified names

To use `arithmetic` from another file:

```
import arithmetic

fn main() {
  println("double_plus_one(5) = #{arithmetic.double_plus_one(5)}")
}
```

`import arithmetic` makes the `pub` items of that module accessible under the prefix `arithmetic.`. The function `double_plus_one` is called as `arithmetic.double_plus_one`.

The prefix is deliberate. When a project grows to fifteen or twenty modules, seeing `arithmetic.double_plus_one` in an expression tells you instantly where it comes from. The alternative — pulling all names loose into the consumer's namespace — saves keystrokes but loses the trail.

The same prefix is used for types the module exports and for constructing them:

```
import geometry

fn main() {
  let a : geometry.Point = geometry.Point { x: 0.0, y: 0.0 }
  let b : geometry.Point = geometry.Point { x: 3.0, y: 4.0 }
  println("distance = #{geometry.distance(a, b)}")
}
```

Three uses of the prefix, all consistent: in the type annotation (`: geometry.Point`), in the record construction (`geometry.Point { ... }`), and in the function call (`geometry.distance(a, b)`). One mental pattern.

If a module lives under a subdirectory, the import uses the same dotted name as the module:

```
import util.math

fn main() {
  println("3^4 = #{math.pow(3, 4)}")
}
```

Two details: the file lives at `util/math.kai`, the module is called `util.math`, but when you use it the prefix is only `math` (the last segment). That keeps prefixes short without losing the structural origin.

Three import forms

kaikai admits three forms that cover the space:

```
import math.vector              # use qualified: vector.dot(a, b)
import math.vector as V           # alias: V.dot(a, b)
import math.vector.{dot, cross}     # specific names into scope
```

The first is the most common and should be your default: the prefix `vector.` makes clear where each name comes from. The second helps when the module name is long or appears many times and a short alias buys you readability without losing the trail. The third is **selective import**: an explicit list of names brought into the local namespace. It pays off only when those names are the protagonists of the file and the prefix becomes noise (a `Point` that appears forty times, for instance). The cost is that the reader no longer knows where `Point` came from without looking up the file.

There is no wildcard import. That is, no `import math.vector.*` or equivalent that pulls "everything the module exports" into the local namespace. This is deliberate: wildcards look convenient at write-time but make future readers unable to trace where a name came from.

8.3 Visibility: the module's contract

`pub` is the contract your module offers to the world. Everything not marked is private, and marking something `pub` declares "I will sustain this name, this signature, this type".

A rule the language doesn't enforce but is worth applying: **keep the public surface narrow.** Every `pub` name is a commitment you'll have to maintain. If you export a useful helper today that's specific to your implementation, tomorrow during a refactor you'll have to choose between breaking callers and carrying around code you no longer want. Public is measured in what you promise; private, in what you can change without notice.

In practice:

- **Exported types:** yes, usually part of the contract.
- **Helper functions:** rarely. If you need to export them, ask whether they belong in this module at all.
- **Constants:** yes, if they're part of the API. Otherwise, no.

Most languages with public-by-default end up with modules whose "real API" is mixed with everything else. kaikai inverts that: public is what you named explicitly.

Private fields inside a public type

There's a useful asymmetry. When you declare `pub type T = { ... }`, the record's fields are **public by default**: the type is part of the contract, so are its fields. That's what you want most of the time.

But sometimes the type itself belongs in the contract while a particular field is an implementation detail. The `priv` keyword before a field name marks it as invisible to other modules:

```
pub type Account = {
  name:        String,
  priv balance: Real,
}
```

Outside the module that declares `Account`, `a.balance` doesn't compile and a literal `Account { name: "x", balance: 100.0 }` doesn't either. Only the declaring module can read the field and name it when constructing. §4.1.1 covered the pattern in detail; what matters for ch. 8 is that `priv` works at **field granularity**, complements the `pub` that controls visibility at the declaration level, and together they cover a very common case: "export the type, hide its interior."

The `ch08/06_priv/` example is a two-file project that shows exactly this rejection from the other side of the module boundary.

8.4 The stdlib you get for free

There is one special module you **don't need to import**: the core stdlib. Functions like `println`, `assert`, `string_concat`, `real_sqrt` live in the global namespace and are available in every `.kai` file without ceremony. This is what other languages call the **prelude**.

What lives in core is deliberately small: primitive types, arithmetic operators, `Option` and `Result`, a handful of list functions, basic IO via `println`. The rest of the stdlib (encoding, networking, files, cryptography) lives in separate modules imported like any other:

```
import list             # non-prelude list operations
import string           # string operations
import encoding.json    # JSON parsing
```

The bar for core is high: only what almost every program needs gets in.

8.5 Projects: `kai.toml`

So far we've talked about **modules** inside a single file tree. When those modules become a unit you want to version, distribute, or that depends on other similar units, you move to the second scale: the **project**.

A kaikai project is described by a `kai.toml` file at its root. The minimal form:

```
name = "myapp"
version = "0.1.0"

[dependencies]
```

`name` is the project's name. It must match a simple grammar: lowercase, digits, underscore and hyphen, not starting with a digit. This is the same shape Cargo, Go modules, and Hex.pm use, and it dodges path traversal, flag collisions, and other nasties.

`version` is the project's version. Before 1.0, `0.MINOR.PATCH` with breaking changes bumping MINOR (cz convention). After 1.0, standard semver.

`[dependencies]` is the table where you declare which other projects yours needs. Empty if your project has no external deps beyond the stdlib.

edition

There's one more optional field worth pinning the moment a project stops being a sketch:

```
name = "myapp"
version = "0.1.0"
edition = "hanga-roa"

[dependencies]
```

`edition` binds the source to a version of the **language contract**: syntax, type-system semantics, stdlib signatures, the `kai` CLI surface. While an edition is active, kaikai guarantees your project keeps compiling as the compiler moves forward, even when internals change. When an edition closes and a new one opens, kaikai keeps accepting the old one — you just have to declare which one you use.

If you omit the field, the compiler assumes the installation's default edition. That's fine for sketches; for any project that will live, pinning the edition is what stops a silent upgrade from moving the ground under you. Ch. 16 covers the rest: how editions are chosen, how to migrate between them, and what `#[unstable]` means for APIs in flux.

Starting a fresh project

```
$ mkdir myapp && cd myapp
$ kai init myapp
kai-pkg: wrote kai.toml for package 'myapp'
```

`kai init` writes the skeleton `kai.toml`. From there you add `.kai` files and import them as in §8.2.

8.6 Dependencies: git, path, lock

When you declare a dependency, three forms are accepted:

```
[dependencies]
manutara = { source = "github.com/kaikailang-org/manutara", ref = "v0.1.0" }
kohau = "github.com/kaikailang-org/kohau@v0.2.0"
local = { path = "../another-lib" }
```

The first form (inline table with `source` and `ref`) is the canonical shape for git dependencies. `source` is the repo URL. `ref` is whatever git understands as a ref: a tag (`v0.1.0`), a branch (`main`), or a commit SHA (`abc1234`). Tags are the convention; branches and SHAs are escape hatches.

The second form (string `"<source>@<ref>"`) is the same thing shortened. The `@` splits source from ref.

The third form (`{ path = "..." }`) is the **local dependency**. It points at another project on disk, typically because you're developing it in parallel. Edit the dependency, re-run your app, changes show up instantly without republishing.

Adding a dependency

```
$ kai add github.com/kaikailang-org/manutara@v0.1.0
```

`kai add` does two things atomically: clones the dependency and writes the entry in `kai.toml`. If the clone fails (bad URL, missing ref, network error), neither manifest nor lockfile changes. This matters: the working tree never drifts into an inconsistent state.

The lockfile

The first time dependencies are resolved, `kai` clones the repos, captures the exact commit (SHA), and writes a `kai.lock`:

```
# kai.lock — generated by kai-pkg. Do not edit by hand.

[[package]]
name = "manutara"
source = "github.com/kaikailang-org/manutara"
ref = "v0.1.0"
sha = "abc123def456..."
```

The lockfile closes the contract. If two developers run `kai install` with the same `kai.toml` and the same `kai.lock`, they **download exactly the same SHA**, exactly the same file tree, exactly the same binary at the end. Reproducibility is the lockfile's core promise.

That's why `kai.lock` is committed alongside the code: it's part of the project's contract. `kai.toml` declares what you want; `kai.lock` declares what you got.

When the lock is updated

- `kai install` creates it if absent; respects it if present.
- `kai update` regenerates it with the latest version of each dependency that satisfies the declared ref.
- `kai add` refreshes it when you add a new dependency.
- `kai run` and `kai build` refresh it automatically when they detect deps in `kai.toml` not in the lock (first run after a `git clone`, say).

8.7 Version selection: MVS

When a project depends transitively on the same other project through two paths with different declared versions, which version wins?

kaikai handles this with **minimum-version selection (MVS)**, the algorithm used by Go modules. The rule is blunt but clear: from the set of versions declared by the transitive chain, the **maximum** wins.

If your app declares `manutara@v0.1.0` and one of its dependencies declares `manutara@v0.2.0`, the whole project uses `v0.2.0`. The underlying assumption is that versions are backwards-compatible within a major: if everyone respects semver, bumping to the higher number should not break anyone.

MVS contrasts with Cargo's or npm's algorithm, which solves complex constraints and sometimes downgrades to satisfy someone. MVS is **predictable**: given the same tree, the result is the same. There is no "resolver failed", no diamond-dependency hell.

The price is that the responsibility for compatibility rests on library authors. If you bump to a version that breaks, everyone depending transitively on you breaks. That forces you to take versioning seriously.

8.8 Cache and `kai install`

When you download a dependency, it doesn't land in your project: it lands in a **cache shared across projects**, by default at `~/Library/Caches/kai/pkg` (macOS) or `~/.cache/kai/pkg` (Linux).

The cache layout:

```
~/Library/Caches/kai/pkg/
  github.com/kaikailang-org/manutara/
    abc123def456.../          # content pinned to that SHA
    789abc012def.../          # another SHA of the same repo
```

Each entry is identified by its SHA, not by the user-facing ref. So three different projects asking for `manutara@v0.1.0` share the same on-disk tree. And if at some point `v0.1.0` is updated upstream (tag movement, which shouldn't happen but does), the cache pinned to the original SHA stays intact.

`kai install` can be run explicitly, but **`kai run` and `kai build` invoke it automatically** when they detect unresolved deps. In practice, after `git clone`ing a project, `kai run main.kai` is enough: the driver downloads what's missing and runs.

8.9 Case study: refactoring a monolith

Imagine a project that started as a single `main.kai` of 800 lines. Inside, three responsibilities are tangled: parsing a config format, business logic, and file IO. The change rate has dropped: any modification forces reading too much.

The refactor proceeds in four steps.

Step 1: split into modules of the same project

Identify the three responsibilities and break them into files:

```
myapp/
├── kai.toml
├── main.kai          # only the main flow
```

```
├── config.kai        # config parsing
└── domain.kai        # business logic
```

In `main.kai`:

```
import config
import domain

fn main() {
  let cfg = config.load("./config.toml")
  let result = domain.process(cfg)
  println("result: #{result}")
}
```

`config.kai` and `domain.kai` export only the functions `main` needs. Everything else (helpers, internal types) stays private.

Step 2: review `pub`

Walk through every `pub`. For each one, ask: is this really part of the module's contract, or did I leave it exported by inertia? Every extra `pub` is a future commitment.

Step 3: extract a local dependency

`config.kai` turns out to be useful beyond this project. You extract it into its own repo:

```
config-lib/
├── kai.toml          # name = "config_lib"
└── config.kai

myapp/
├── kai.toml          # now depends on config_lib
├── main.kai
└── domain.kai
```

In `myapp/kai.toml`:

```
name = "myapp"
version = "0.1.0"

[dependencies]
config_lib = { path = "../config-lib" }
```

While developing, `config_lib` is a local dependency. `main.kai` updates its import to `import config_lib.config` (or an alias if the long name annoys you).

Step 4: publish

When `config_lib` stabilizes, tag it:

```
$ cd ../config-lib
$ git tag v0.1.0
$ git push origin v0.1.0
```

And in `myapp` switch the path to git:

```
[dependencies]
config_lib = { source = "github.com/youruser/config-lib", ref = "v0.1.0" }
```

`kai install` downloads the pinned version and the lockfile nails it down. The code in `main.kai` and `domain.kai` doesn't change: the imports remain the same.

That's the full trajectory: from one file to modules, from modules to a project, from a local project to a distributed dependency. Every step is reversible. Every step is optional.

8.10 Philosophy: simple and predictable

The decisions in this chapter aren't the most expressive available. Other languages have more powerful module systems: automatic re-exports, dynamic aliases, modules parameterized by values. kaikai picks an austere variant and stops there.

The reason is the same as elsewhere in the language: what eliminates head-scratching matters more than what adds power. A module system where nothing is imported without appearing in a visible list, where a module name is derived from its path, and where dependencies are pinned to an exact SHA is a system you can understand by reading, without having to mentally execute it.

It's the same principle as exhaustive pattern matching: if the compiler can take you to the exact place where you need to act, you don't need a more sophisticated tool.

Exercises

8.1. Take a program you wrote in earlier chapters and split it into two files. Identify which declarations need to be `pub` and which can stay private. Did you feel tempted to mark something `pub` "just in case"?

8.2. Create a project with `kai init`. Add an auxiliary file under a subdirectory (say, `util/strings.kai`). Import a function from `main.kai`. What is the module's name from the import side?

8.3. Write two sibling projects on disk: a library `my_lib` with one public function, and an app `my_app` that uses it with `{ path = "../my_lib" }`. Edit `my_lib`, re-run `my_app`. How long is the edit-run cycle?

8.4. Open a kaikai project on GitHub (say, `github.com/kaikailang-org/kaikai`) and read its `kai.toml` if any. What dependencies does it declare? Do you recognize the versioning pattern?

8.5. If `my_app` declares `manutara@v0.1.0` and a transitive dependency declares `manutara@v0.2.0`, which version does the project use? Why does this design avoid the "diamond dependency hell"? Construct a scenario where MVS may surprise you.

8.6. A library of yours had a `pub fn parse_legacy(s: String)` that nobody uses anymore. Under what conditions could you remove it without releasing a `1.0`? What kind of communication with users do you need first?

Chapter 9 · Protocols

So far you've seen how to group data (records, sum types) and how to define functions over it. What's missing is a concrete question: **how do you add operations to a type from outside its declaration?**

For example: `Point` is a record with two fields. You want it to print as `(3, 4)` when it appears in interpolation. Do you modify the type? Pass a function to `println`? Define a `println` variant just for `Point`? None of these scale.

kaikai's answer is **protocols**: a named contract with a small set of operations, that any type can satisfy. Conceptually, protocols do what **interfaces** do in Go, **traits** in Rust, **protocols** in Clojure and Elixir, and the easy part of **typeclasses** in Haskell. But kaikai picks a precise point in the design space: **single-dispatch, explicit, no constraint propagation, no higher-kinded types**. That removes complexity from the type system at the cost of some things you can't express — and this chapter covers both sides.

9.1 Why protocols exist

Three concrete pains protocols solve.

Printing your types without writing it each time. When you declare a record, wanting to print it in logs, responses, errors should not cost you a `user_to_string` function per type. With `Show` implemented once, interpolation `"#{user}"` uses it automatically.

Structural equality without effort. Any new type needs "this equals that" at some point. Without protocols, you write `eq_point`, `eq_account`, `eq_invoice`, and spend the rest of the project remembering which name you used in which file. With `Eq`, the operation is called `eq` for all of them.

Comparison, hashing, serialization. Same argument as the previous, multiplied by the three standard operations almost every type needs eventually.

Without protocols, each of the three is solved with naming conventions case by case, or with huge `match`es enumerating each possible type. With one mechanism, the three (and the ones to come) are solved in one line per type.

9.2 Declaring a `protocol` and `impl`

The syntax is direct. A protocol declares a name and one or more operations:

```
protocol Show {
  show(x: Self) : String
}
```

`Self` is a reserved name referring to the type that later implements the protocol. Every operation mentions `Self` at least once — that's why it's called **single-dispatch**: the operation is decided from a single type, that of `Self`.

To implement it, you write `impl ... for ...`:

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

impl Show for Point {
  fn show(p: Point) : String =
    "(" ++ int_to_string(p.x) ++ ", " ++ int_to_string(p.y) ++ ")"
}
```

And from then on, `show(p)` calls your implementation when `p : Point`:

```
fn main() {
  let p = Point { x: 3, y: 4 }
  println(show(p))            # prints "(3, 4)"
  println("p is at #{p}")     # interpolation: uses Show
}
```

Three details worth pinning down:

- **The body of the `impl` lists the protocol's functions one by one**, with the standard `fn` syntax. Each `fn` in the block must match the signature declared in the `protocol`, with `Self` substituted by the concrete type.
- **One implementation per (protocol, type) pair.** If two `impl Show for Point` appear in the same compilation, the compiler rejects with "duplicate impl". No overriding, no contextual resolution.
- **The orphan rule**: you can only implement a protocol `P` for a type `T` if `P` is declared in your module **or** `T` is declared in your module. This prevents two external packages from defining conflicting impls for types they both import. It's a practical limitation, not a type-system one.

9.3 The five stdlib protocols

kaikai ships five protocols in `stdlib/protocols.kai` you'll use all the time:

Protocol	Operations	What for
`Show`	`show(x: Self) : String`	Convert to string for printing
`Eq`	`eq(a: Self, b: Self) : Bool`	Equality
`Ord`	`cmp(a: Self, b: Self) : Int` , `min` , `max`	Total ordering
`Hash`	`hash(x: Self) : Int`	For hash tables and sets
`Serialize`	`to_string` , `from_string`	Text ↔ value conversion

The primitive types (`Int` , `Real` , `Bool` , `String` , `Char`) already have implementations for the five protocols. When you declare a new type, you choose which protocols are worth implementing for it.

`Ord` is worth a note: it has **three operations**, not one. `cmp(a, b) : Int` returns a negative integer if `a < b` , zero if equal, positive if `a > b` . The other two — `min(a, b)` and `max(a, b)` — return one of the two arguments according to ordering. The three are implemented together in the same block:

```
impl Ord for Account {
  fn cmp(a: Account, b: Account) : Int =
    if a.balance < b.balance { 0 - 1 }
    else if a.balance > b.balance { 1 }
    else { 0 }

  fn min(a: Account, b: Account) : Account =
    if a.balance < b.balance { a } else { b }

  fn max(a: Account, b: Account) : Account =
    if a.balance > b.balance { a } else { b }
}
```

If implementing three operations of `Ord` feels like too much work, hold on for §9.4: in many cases, the compiler derives them for you.

9.4 `#[derive(...)]` and when to use it

For records where the obvious "delegate to each field" implementation is enough, kaikai gives you a shortcut: the `#[derive(...)]` directive before the type declaration.

```
#[derive(Show)]
type Person = {
  name: String,
  age: Int,
}

#[derive(Show, Eq)]
type Point = {
  x: Int,
  y: Int,
}
```

`#[derive(Show)]` tells the compiler "generate me a `Show` for this record, walking the fields and delegating to each field's `Show`". The result for `Person`:

```
$ kai run example.kai
Person { name: Ada, age: 30 }
```

The canonical format — `TypeName { field: value, ... }` — is what `#[derive(Show)]` produces, and is reasonable for debugging and logs. If you want a different format (say, the classic `(3, 4)` for a point), you write the `impl` by hand, like §9.1.

`#derive` works for the five stdlib protocols, as long as **every field of the record also implements** the protocol you're deriving. If your record has a field whose type doesn't have `Show`, `#[derive(Show)]` fails compilation with a message pointing at the offending field.

The practical rule:

- **Start with `#derive`** — the fastest way and almost always correct for records.
- **Switch to `impl` by hand** when the derived implementation isn't useful: different format, equality by some fields only, comparison by a specific field (not the natural ordering).

You already saw `#[derive(Show)]` over `Point` in the tour (§1.6); here we show the manual implementation too and when each form pays off.

9.5 Custom protocols

The five from the stdlib are the most common, but nothing prevents you from declaring your own. It's exactly the same syntax the stdlib uses, but in your code:

```
protocol Drawable {
  draw(x: Self) : String
}

type Circle = { radius: Int }
type Square = { side: Int }
type Triangle = { base: Int, height: Int }

impl Drawable for Circle {
  fn draw(c: Circle) : String =
    "Circle of radius " ++ int_to_string(c.radius)
}

impl Drawable for Square {
  fn draw(s: Square) : String =
    "Square of side " ++ int_to_string(s.side)
}

impl Drawable for Triangle {
  fn draw(t: Triangle) : String =
    "Triangle " ++ int_to_string(t.base) ++ "x" ++ int_to_string(t.height)
}
```

From then on, three different types share the operation `draw`. The polymorphic function `draw` resolves statically: the compiler knows in each call which `impl` to use from the type of the argument.

`Drawable` is a toy example; real cases you'll see in kaikai code include `Encodable` for various formats, `Loggable` so the logging system knows how to represent your type, `Validable` for validation rules, etc. Any "behavior several types share" is a candidate.

9.6 Why no Haskell-style typeclasses

kaikai's protocols may resemble Haskell typeclasses — and draw inspiration from them — but are **deliberately simpler**. Three things kaikai doesn't do that Haskell does:

No constraints in function signatures. This is the most visible difference. In Haskell, a function that sorts a list declares that the element type must have `Ord`:

```
sort :: Ord a => [a] -> [a]
```

The `Ord a =>` is the **constraint**. When you call `sort xs`, the compiler looks up the `Ord` for the type of `xs` on its own and "injects" it into the function without you writing anything. The constraint travels hidden.

In kaikai that doesn't exist. You can't write:

```
fn sort[T : Ord](xs: [T]) : [T] = ...    # ERROR: kaikai admits no constraints
```

How do you sort a list, then? The function takes the **comparator as an explicit argument**:

```
fn sort_by[T](xs: [T], cmp: (T, T) -> Int) : [T] = ...
```

And the call site **names** the comparator. If `Transaction` implements `Ord`, its `cmp` is available as an ordinary function, and you pass it:

```
list.sort_by(transactions, cmp)   # cmp comes from impl Ord for Transaction
```

The difference is small to write but big conceptually: in Haskell the `Ord` is implicit, in kaikai it's explicit. The function `sort_by` doesn't "demand" `T` to have `Ord` — it demands that **someone pass it a comparison function**. That this function comes from an `impl Ord for T` is the caller's decision, not the signature's.

No higher-kinded types (HKT). `protocol Functor[F[_]]` doesn't parse. Type parameters of protocols are first-order. This rules out a family of abstractions (Functor, Monad, Applicative, etc.) that in Haskell are central — and that in kaikai are addressed with algebraic effects (chapter 12) and explicit combinators.

No constraint propagation. A polymorphic function does not "carry" `Ord` to the functions it calls. If you call something that requires `Ord`, you pass the comparator.

What do you gain with these restrictions? Three things:

- **Fast compilation**. Type inference stays Hindley-Milner extended with effects, no Haskell-style constraint solver overhead.
- **Clear errors**. If a function needs `Ord` and doesn't receive it, the error points at the site missing the comparator. There are no chains of "no instance for `Ord (Maybe a)` because of `Ord a`".
- **The call site says what it does**. When you see `list.sort_by(xs, cmp)`, you know it's sorted with `cmp`. When you see `sort xs` in Haskell, you have to look at `sort`'s signature to know which `Ord` is used.

What do you lose? Some abstractions that are elegant in Haskell — particularly everything that lives over Functor and friends. That trade-off is deliberate: the abstractions kaikai prioritizes live in the effect system (chapter 12), not in the type system.

9.7 Operators: `+`, `==`, `<` as protocols

A practical note: standard operators **are** protocols. `==` is `Eq.eq`, `<` is comparison based on `Ord.cmp`, `+` is `Add.add`. When you declare `impl Eq for Account`, automatically `a1 == a2` (with `a1, a2 : Account`) calls your implementation.

```
#[derive(Eq)]
type Point = { x: Int, y: Int }

fn main() {
  let p = Point { x: 3, y: 4 }
  let q = Point { x: 3, y: 4 }
  if p == q { println("equal") }   # uses derived Eq.eq
}
```

This unifies syntax: `+` for `Int`, `Real`, vectors, matrices, money — all those that have `impl Add for ...`. `==` for everything that has `Eq`. The uniformity isn't accidental; it's the first benefit of having a single mechanism for "operations dispatched by type".

Operators kaikai treats as protocols:

Operator	Protocol	Op
`==`, `!=`	`Eq`	`eq`
`<`, `<=`, `>`, `>=`	`Ord`	`cmp`
`+`, `-`, `*`, `/`	`Add`, `Sub`, `Mul`, `Div`	`add`, `sub`, `mul`, `div`
`++`	`Concat`	`concat`

Primitive types implement them all. Your types implement them when you declare them. And when an operator doesn't make sense for a type, you simply don't implement it — and the compiler rejects that expression.

Exercises

9.1. Define `type Distance = { meters: Int }` and give it an `impl Show` so `show(d)` produces `"42 m"`. Then test `println("the distance is #{d}")`. Note: interpolation works, but if you call another protocol op inside the `#{...}` with multiple args, remember the binding-first workaround.

9.2. Define `type Card = { suit: String, value: Int }` and give it `impl Ord` ordering by `value`. Verify with three cards that `cmp(c1, c2)` returns the expected numbers.

9.3. Take any record from earlier chapters and add `#[derive(Show, Eq)]` above the declaration. Verify that `show` and `eq` work without you writing anything else.

9.4. Declare your own protocol `Validable` with an operation `validate(x: Self) : Result[String, Self]` returning `Ok(x)` if the value is valid or `Err("reason")` otherwise. Implement `Validable` for `type Age = { years: Int }` so it rejects negative ages or those above 130.

9.5. Read the stdlib code in `stdlib/protocols.kai`. How many operations does `Hash` have? How would it be used to implement a hash table of `Account`? Design the signature of the function you'd write to "look up an account by id" in a hypothetical hash table — what would need to be implemented in `Account` for it to work?

Chapter 10 · Units of measure and branded types

In 1999, NASA lost the Mars Climate Orbiter — a 327 million-dollar project — because two software modules exchanged numerical values without agreement on units. One produced thrust in pound-force per second; another read it as newtons per second. Nobody had written units on the interface. The probe disintegrated on Mars.

It's an extreme example, but the family is huge: two components sharing the same numeric type but not the same interpretation. Passing seconds where milliseconds are expected. Adding balances in different currencies. Mixing a `UserId` with an `OrderId` because both are integers. In every case, the language's type system sees two numbers or two strings and has no way to distinguish them.

kaikai solves this whole family with a single construct: **units of measure**. The idea, inherited from F#, is that you can annotate a number with a unit (`Real<USD>`, `Int<Seconds>`) and the compiler rejects any operation that mixes incompatible units. The unit **lives in the type**, **is erased at runtime**, and is documented in every signature it touches.

This chapter covers classical units (physics, finance, time) and their less obvious but more frequent use in day-to-day code: **branded types**, where the unit is just a label distinguishing types that the language, without it, would treat as the same.

10.1 `unit` and annotated literals

A unit is declared with the keyword `unit`:

```
unit USD
unit EUR
unit m
unit sec
unit kg
```

That's all. `unit USD` introduces a symbol `USD` the type system can use as an annotation. There's no implementation, no runtime value — `unit` only declares the existence of the symbol.

To annotate a number with a unit, use angle brackets in the literal:

```
let price : Real<USD> = 1.50<USD>
let speed : Real<m / sec> = 9.81<m / sec>
let timeout : Int<sec> = 30<sec>
```

Three things to pin down:

- **Units are written with `<...>`**, not `[...]`. Brackets are for type parameters (`List[Int]`, `Option[String]`); angle brackets are for units.
- **Both the literal and the type are annotated.** `1.50<USD>` is a `Real` literal with unit `USD`. `Real<USD>` is the type. They match, but each plays a role.
- **The compiler accepts any identifier as a unit.** By convention, SI units go in lowercase (`m`, `s`, `kg`), person-named in titlecase (`Newton`, `Pascal`), and currencies in uppercase per ISO 4217 (`USD`, `EUR`, `CLP`). The compiler enforces none of these conventions — it's community style.

10.2 Arithmetic with units

Arithmetic between values of the same unit works as you'd expect. Adding two `Real<USD>` gives another `Real<USD>`:

```
let price : Real<USD> = 1.50<USD>
let tip : Real<USD> = 0.30<USD>
let total : Real<USD> = price + tip    # 1.80 USD
```

But **mixing incompatible units is a compile error**. If you try:

```
let mix = price + 1.20<EUR>    # type error
```

the compiler rejects with an error saying "expected `USD`, got `EUR`". The program never runs with mixed values — the bug is detected before existing.

The broader rule:

- `+`, `-` between two values of the same unit: legal, preserves the unit.
- `*`, `/` between values with units: legal, **composes the units**. See §10.3.
- `+`, `-` between distinct units: type error.
- **Comparisons (`<`, `==`)** between distinct units: type error.

Arithmetic with a value without a unit and one with a unit is asymmetric. Multiplying by a scalar (`2.0 * price`) is legal — the scalar is treated as dimensionless. Adding a scalar (`price + 1.0`) is an error: we don't know what unit the `1.0` is in.

10.3 Unit algebra: product, quotient, power

Multiplying two units produces a composite unit. The classic example is force:

```
let mass : Real<kg> = 70.0<kg>
let acceleration : Real<m / sec^2> = 9.81<m / sec^2>
let force : Real<kg * m / sec^2> = mass * acceleration
```

`kg * m / sec^2` is the composite unit of force. The compiler deduces it automatically from the operands: `kg * (m / sec^2)` simplifies to `kg * m / sec^2`. If you then divide force by area to get pressure, the final unit is `kg / (m * sec^2)`, all derived mechanically:

```
let area : Real<m^2> = 4.0<m^2>
let pressure : Real<kg / (m * sec^2)> = force / area
```

The type system does **unit algebra**. `m * m` simplifies to `m^2`, `sec / sec` cancels to no-unit, `(m / sec) * (sec)` cancels to `m`. It's abelian algebra over unit symbols — and any expression legal in elementary math over units is legal in kaikai.

Aliases for derived units

When a composition appears often, you can name it with a `unit` with a body:

```
unit Newton = kg * m / sec^2
unit Pascal = Newton / m^2
unit Hertz  = 1 / sec
```

From there, `Real<Newton>` is exactly the same as `Real<kg * m / sec^2>` — the compiler accepts them interchangeably. The difference is for the reader: `Real<Newton>` communicates intent; `Real<kg * m / sec^2>` communicates derivation.

10.4 Generic units

So far, each function operates on a concrete unit. But many operations are **unit-agnostic** — averaging, summing, sorting, finding the maximum. Those functions are written **generic over the unit**, just as a function can be generic over the element type of a list.

```
fn average[u: Measure](a: Real<u>, b: Real<u>) : Real<u> =
  (a + b) / 2.0
```

`u : Measure` declares `u` as a type parameter in the **kind** `Measure`. The only thing `u` admits is being a unit. The function `average` takes two `Real<u>` and returns a `Real<u>` — the "for any u" lets you use it with `USD`, `kg`, `m/sec`, anything, **as long as the two arguments have the same unit**.

```
let pp : Real<USD> = average(10.0<USD>, 20.0<USD>)   # 15 USD
let pm : Real<kg>  = average(70.0<kg>, 80.0<kg>)     # 75 kg
```

But this is a type error:

```
let mix = average(10.0<USD>, 70.0<kg>)   # u can't be USD and kg
```

The compiler instances `u = USD` for the first argument, and demands the second use `u = USD` too. `kg` doesn't fit, and the program doesn't compile.

Generic units are what makes the system **scalable**. Stdlib functions (`list.sum`, `list.max`, `list.min`) are polymorphic over the unit: pass a `[Real<USD>]`, you get `Real<USD>`. Pass `[Real<kg>]`, you get `Real<kg>`. The unit is preserved without you naming it.

10.5 Explicit conversions

What if you **do** want to mix two distinct units? For example, adding a USD balance with an EUR one. The compiler doesn't allow it accidentally, but it does allow it with an **explicit conversion**.

The technique is a conversion factor carrying the quotient unit `dest/source`. Multiplying by it cancels the source unit and leaves the destination one:

```
let eur_amount : Real<EUR> = 80.0<EUR>
let rate : Real<USD / EUR> = 1.10<USD / EUR>
let usd_amount : Real<USD> = eur_amount * rate    # 88 USD
```

The arithmetic follows: `EUR * (USD / EUR)` cancels `EUR` and leaves `USD`. The compiler verifies the cancellation is correct — if you put the rate the wrong way around (`Real<EUR / USD>`), the multiplication produces `Real<EUR^2 / USD>`, which doesn't fit the `Real<USD>` expected by the `let`.

Mental rule: **conversion is multiplication by a factor whose units cancel at the output**. It's exactly what physicists do by hand when they write "10 km × 1000 m/km = 10000 m". kaikai forces writing that multiplication explicitly.

This is deliberate and at the same time freeing. Deliberate because forcing a visible conversion makes the programmer think about what rate they're using, when it was applied, and where it came from. Freeing because once the conversion is written, the compiler guarantees you didn't forget it anywhere.

10.6 Branded types

Units are useful when the value "has a physical dimension": meters, kilograms, dollars. But the same machinery applies to a more everyday and more frequent case: **distinguishing values that have the same underlying type but mean different things**.

The classic example are identifiers. A `UserId` is an integer. An `OrderId` too. Without units, the compiler cannot tell them apart:

```
fn cancel_order(id: Int) : Unit / Stdout = ...
fn send_email(uid: Int) : Unit / Stdout = ...

let user_id = 42
let order_id = 99
cancel_order(user_id)   # bug: we passed a user id to a
                        # function expecting an order id,
                        # but it compiles anyway.
```

With units, both identifiers are distinct types:

```
unit UserId
unit OrderId

fn cancel_order(id: Int<OrderId>) : Unit / Stdout = ...
fn send_email(uid: Int<UserId>) : Unit / Stdout = ...

let user_id : Int<UserId>  = 42<UserId>
let order_id : Int<OrderId> = 99<OrderId>

cancel_order(user_id)   # type ERROR: UserId ≠ OrderId
```

The compiler detects the bug before it exists. The technique of using a unit as a **label** over a numeric type is called **branded type**, and it's one of the uses that pays off most in everyday code. Typical cases:

- `Int<UserId>` vs `Int<OrderId>` — identifiers sharing underlying type.
- `Int<Cents>` vs `Int<Quantity>` — money in cents vs a count of units.
- `Int<Seconds>` vs `Int<Milliseconds>` — wrongly expressed timeouts are responsible for a non-trivial percentage of intermittent bugs.
- `String<Email>` vs `String<Username>` — strings that passed through different validations.
- `String<RawHtml>` vs `String<Sanitized>` — input not escaped vs ready for safe injection.

The first two cases over `Int` work in any current version of kaikai. The last two over `String` are partially implemented; full handling of branded types over `String` and arbitrary records is an extension the language doc lists as a near milestone.

Zero cost at runtime

Units are erased after type checking. The binary that `kai build` produces operates with plain `Int`, plain `Real`, plain `String`. The unit **doesn't exist** at runtime: no tag, no dynamic verification, no overhead. The promise is the same as algebraic effects and contracts: information in the type, zero cost at runtime.

10.7 Case study: multi-currency wallet

We close with an integrative example. A wallet contains balances in different currencies. Adding balances of the same currency is trivial; combining them into a total requires conversion.

```
unit USD
unit EUR
unit CLP

fn add[c: Measure](a: Real<c>, b: Real<c>) : Real<c> = a + b

fn convert[source: Measure, dest: Measure](amount: Real<source>, rate: Real<dest / source>) :
Real<dest> = amount * rate
```

Three declarations. `add` is generic over the currency and preserves the unit of the arguments — same technique as §10.4. `convert` takes an amount and a rate, and returns the amount in the destination unit thanks to unit cancellation.

And the main calculation:

```
fn main() {
  let usd_balance_1 : Real<USD> = 100.0<USD>
  let usd_balance_2 : Real<USD> = 50.0<USD>
  let total_usd : Real<USD> = add(usd_balance_1, usd_balance_2)    # 150

  let eur_balance : Real<EUR> = 80.0<EUR>
  let eur_to_usd_rate : Real<USD / EUR> = 1.10<USD / EUR>
  let eur_in_usd : Real<USD> = convert(eur_balance, eur_to_usd_rate)
  let global_total : Real<USD> = add(total_usd, eur_in_usd)        # 238

  let clp_balance : Real<CLP> = 100000.0<CLP>
  let clp_to_usd_rate : Real<USD / CLP> = 0.0011<USD / CLP>
  let clp_in_usd : Real<USD> = convert(clp_balance, clp_to_usd_rate)
  let final_total : Real<USD> = add(global_total, clp_in_usd)      # 348
}
```

Three conversions, three uses of `add`. Each verified by the compiler: had you tried to add `Real<EUR>` with `Real<USD>` without converting somewhere, it wouldn't compile. If the rate were inverted (`<EUR / USD>` instead of `<USD / EUR>`), neither would. The full wallet is **a mini-type-system over money** that the compiler upholds.

What happens the day a colleague arrives and adds a new currency? They declare a `unit JPY`, add its rate, and the rest of the code keeps compiling or not as appropriate. No old calculation breaks — types are orthogonal — and calculations that need to consider JPY have to say so explicitly. It's the same principle chapter 5 showed with union errors: adding a new component is safe because the compiler walks you to the exact place needing change.

Compare with the no-units version:

```
fn add(a: Real, b: Real) : Real = a + b

let total = add(usd_balance, eur_balance)   # compiles, sums values
                                            # in different currencies
                                            # without converting.
```

It works, returns a number, and produces a total that **means nothing**. In kaikai with units, the same program doesn't compile. The bug that in other languages is discovered in production — or never, depending on luck — doesn't exist here.

Exercises

10.1. Define `unit Celsius` and `unit Fahrenheit`. Write `fn celsius_to_fahrenheit(c: Real<Celsius>) : Real<Fahrenheit>` applying the right formula. What's the conversion factor? Does it have a unit?

10.2. Define `unit Cents` (cents) and write `fn pay(amount: Int<Cents>) : Unit / Stdout`. Build an example where the language detects a bug of "passing an amount in dollars where cents were expected".

10.3. Take the §10.7 case study and add a new currency `JPY` with its rate. How many lines do you have to change? Is there any line of existing code that breaks just because you added the unit?

10.4. Write a generic function `fn range[u: Measure](xs: [Real<u>]) : Option[Real<u>]` that returns the difference between max and min of a list, or `None` if empty. What unit does the result have?

10.5. In your work or a personal project, identify **two** places where two distinct integers mean different things (identifiers, indices, counters, timeouts). Write in pseudo-kaikai how the affected function signatures would look using branded types. How many historic bugs could have been avoided?

Chapter 11 · Programming by contract and refinement types

This chapter closes Part II and completes the thread "information in the type, zero cost at runtime" started by algebraic effects (chapter 12), continued by units of measure (chapter 10), and finished here with two mechanisms that come from Eiffel (1986) and Ada 2012: **contracts** — preconditions and postconditions that live in a function's signature — and **refinement types** — restrictions on the values a type accepts.

The two mechanisms share the same idea: **state restrictions in the type, have the compiler verify them where it can, and defer them to runtime where it can't**. Where they differ is in scope: contracts speak about **operations** (what a function expects and guarantees); refinements speak about **values** (which numbers or strings are valid). The two cover distinct areas and complement each other.

11.1 Why contracts and refinements go together

Imagine you want to model a bank account. The natural rule is "the balance can never be negative". There are two ways to express it:

- **In the value**: declare `type ValidBalance = Int where self >= 0`. Any value of type `ValidBalance` satisfies the rule by construction. The compiler rejects any attempt to put a negative.
- **In the operation**: declare `fn withdraw(c: Account, amount: Int) : Account requires c.balance >= amount ensures result.balance >= 0`. The function demands a condition on the arguments and promises one on the result.

Both forms say the same and one without the other is incomplete. Refinements describe **which values are legal**; contracts describe **what operations do with those values**. A robust bank account uses both: balance refined to non-negative, operations with preconditions ensuring the rule holds.

kaikai treats both as a single project. The language doc (`refinements-and-contracts.md`) says:

> *Together they form a single coherent mechanism — types describe what values are valid, contracts describe what operations guarantee — and the two share most of the implementation machinery.*

That's why they live in the same chapter.

11.2 `requires` and `ensures` in a signature

A contract is written as annotations on the signature of a function, before the `=` opening the body:

```
fn divide(a: Int, b: Int) : Int
  requires b != 0
  ensures  result * b + (a % b) == a
= a / b
```

Three components:

- `requires <expr>` — a **precondition**. The expression must be `true` when **entering** the function. The caller is responsible. If violated, the bug is the caller's.
- `ensures <expr>` — a **postcondition**. The expression must be `true` when **exiting** the function. Your body is responsible. If violated, the bug is internal.
- `result` — a reserved name inside `ensures` that refers to the return value.

A function can have multiple `requires` and multiple `ensures`. The compiler accumulates them:

```
fn withdraw(c: Account, amount: Int) : Account
  requires amount > 0
  requires c.balance >= amount
  ensures  result.balance == c.balance - amount
=
  Account { ...c, balance: c.balance - amount }
```

Two preconditions (positive amount, sufficient balance), one postcondition (the resulting account has exactly `c.balance - amount`). Preconditions are verified in order on entry; the postcondition on exit.

Three details worth pinning down:

- **`requires` and `ensures` are not comments**. They are code the compiler emits as actual verifications. If a precondition is violated at runtime, the program aborts with a clear message:

  ```
  panic: requires violated in `divide`
  required: b != 0
  declared at line 9, col 14
  ```

- **The compiler proves statically what it can**. If you call `divide(10, 0)` with literals, the compiler sees `b == 0` and rejects the call at compile time — no waiting for runtime. If you call with dynamic values, it inserts the assert.
- **Contracts don't run in release builds**, depending on a compiler flag. In that mode, `requires` and `ensures` vanish from the binary; cost is zero. For development and tests, contracts are evaluated.

11.3 `result` and names in scope inside `ensures`

Inside a postcondition you have access to:

- `result` — the return value.
- **The names of the parameters** — their input values.
- **Any pure function** the module provides.

This lets you write relations between input and output:

```
fn duplicate(n: Int) : Int
  ensures result == n * 2
= n + n

fn sort(xs: [Int]) : [Int]
  ensures list.length(result) == list.length(xs)
= ...
```

The first says "the output is twice the input". The second says "the resulting list has the same length as the input" — a reasonable invariant of any sort. Note we're **not** saying that `result` is sorted — only that it preserves length. Postconditions document what's worth documenting; they don't have to be exhaustive.

Unlike Eiffel, **there is no `old`**. In Eiffel you need `ensures balance = old balance + amount` because records are mutable and the `balance` you see in `ensures` has already changed. In kaikai records are immutable: the `c` that comes in and the `result` that comes out are **distinct values**, both in scope, no need for a "value before the call" mechanism.

11.4 Refinement types

While contracts speak about operations, **refinement types** speak about values. A refinement type is a base type plus a predicate:

```
type NonNeg = Int where self >= 0
type Probability = Real where self >= 0.0 and self <= 1.0
type Age = Int where self >= 0 and self <= 130
type Port = Int where self >= 1 and self <= 65535
```

The predicate refers to `self` — the value being restricted. Any value of type `NonNeg` satisfies `self >= 0` by construction; any `Probability` falls in the unit interval; any `Port` is within TCP range.

Building a value of the refined type requires the predicate to hold. If you satisfy it with a literal, the compiler verifies at compile time:

```
let x : NonNeg = 16      # OK: 16 >= 0
let y : NonNeg = 0 - 5   # ERROR: -5 doesn't satisfy self >= 0
```

If you satisfy it with a dynamic value, the compiler inserts a runtime check — same as a `requires` whose argument can't be deduced statically.

Functions that accept refined types **benefit from the guarantee without checking it**. If your signature says `n: NonNeg`, inside you can assume `n >= 0` without writing an `if`. That's exactly what a contract `requires n >= 0` does, but encoded in the type instead of the signature.

When do you use one and when the other? The simple rule:

- **Refinement type** when the restriction **defines what is a valid value** of the domain. `Age`, `Probability`, `Port` are classic examples: the type makes no sense outside the restriction.
- **Contract** when the restriction is **about the operation**, not the value. "Don't divide by zero" is about the divisor; "the amount to withdraw must be positive" is about the withdraw operation; "the resulting list preserves length" is about the function's behavior.

Sometimes both apply and you choose for readability. The bank account in §11.7 uses contracts because "the balance can't be left negative" is a property of **the operations' behavior**, not of the balance's value.

11.5 When proved statically, when at runtime

kaikai treats contracts and refinements as a **continuum between static and dynamic**, decided by what the compiler can demonstrate. Three levels:

Compile time, fully proved. When arguments are literals or the compiler knows their ranges:

```
divide(10, 0)            # compile ERROR: 0 != 0 is false
let x : NonNeg = 0 - 5   # compile ERROR: -5 < 0
```

The program doesn't even produce a binary. The strongest guarantee.

Compile time, partially proved. When ranges can be inferred via bounded analysis, kaikai proves the prudent without invoking a heavy SMT solver. The scope is limited: literal-comparators over `Int`, `Bool`. If the predicate crosses complex arithmetic or functions the compiler can't inspect, it's deferred.

Runtime. When the above isn't decidable, the compiler emits an assert in the generated code. The program compiles; the check happens when executing the function. If it fails, abort with `panic: requires violated`.

What kaikai **doesn't do**, deliberately, is **invoke an SMT solver like Z3 or CVC5** to prove arbitrarily complex contracts. That's the boundary with SPARK (the verifiable subset of Ada). kaikai prefers a small interval evaluator, decidable, linear — and deferring the rest to runtime — over having unpredictable compilation and heavy external dependencies.

Mental rule: **what the compiler can prove cheaply, it proves; the rest is checked at runtime**. The binary carries both cases without you having to distinguish which applied.

11.6 Three forms of guarantee

We have three mechanisms for "guaranteeing the code does the right thing" spread across three chapters:

Mechanism	Ch.	What it guarantees	When verified
`test`	7	For a specific input, the output is this	Under `kai test`
`check`	7	For **every** input, this invariant holds	Under `kai check`, with generated values
Sum types + `match`	5	Cover every possible case of the type	Compile time
Contracts + refinements	11	Restrictions on input/output and valid values	Compile time when possible, runtime when not

The three are distinct tools with overlapping areas. A well-written function probably uses all four:

- **Types** as narrow as possible for the domain (sum types, refinements, units of measure).
- **Contracts** documenting what the function demands and guarantees, in relational terms.
- **Tests** covering the punctual contractual cases the client asks for.
- **Checks** verifying algebraic invariants ("inverting twice is identity", "sorting preserves length").

They aren't redundant: each catches a distinct class of bugs and the four together form a much denser safety net than any of them individually. And all four have zero runtime cost when they don't fail: a passing `test` doesn't run in production; a contract that proves statically generates no assert; a legal refinement generates no dynamic check.

11.7 The Design by Contract family

Contracts aren't a kaikai invention. They have a history, and it's worth placing kaikai in that history.

Eiffel (Bertrand Meyer, 1986) introduced the term "Design by Contract" and most of the ideas: `require` / `ensure` over methods, class invariants, inheritance with weakening preconditions and strengthening postconditions. The canonical syntax lives inside the body:

```
deposit (amount: REAL)
  require
    positive_amount: amount > 0
  do
    balance := balance + amount
  ensure
    balance_increased: balance = old balance + amount
end
```

kaikai takes the idea but puts contracts in the **signature**, not in the body, and removes `old` because records are immutable.

Ada 2012 added "aspect specifications" to Ada with the syntax `with Pre =>`, `with Post =>`. This is the closest to kaikai stylistically:

```
function Divide (A, B : Integer) return Integer
  with Pre  => B /= 0,
       Post => Divide'Result * B = A;
```

Ada's `with Pre => / Post =>` and kaikai's `requires / ensures` are the same annotational idea. Ada also introduced **subtypes with predicates** (`subtype Positive is Integer range 1 .. Integer'Last`), which are direct ancestors of kaikai's refinement types.

SPARK is the verifiable subset of Ada, which uses an SMT solver to prove arbitrary contracts statically. kaikai **doesn't adopt this on purpose**: SPARK requires installing Z3 or similar, and compile times become unpredictable. The language doc (`refinements-and-contracts.md`) says it explicitly: the interval evaluator is a few hundred lines max, decidable, linear; what doesn't fit there is deferred to runtime without remorse.

D has contracts similar to kaikai with the syntax `in { ... } / out (result) { ... }`. **Cobra**, **Kotlin** (with `require/check`), **Clojure** (with `:pre/:post`) are other lighter-weight versions of the same idea.

What **distinguishes** kaikai in this family:

1. **No SMT**. The SPARK line is out of bounds. The consequence is that complex contracts are checked at runtime; the benefit is fast compilation and no external dependencies.
2. **Purity by default**. Eiffel and Ada deal with rampant mutability; contracts have to handle `old` and aliasing. kaikai starts from immutability: the postcondition speaks about the input and the output as two values that coexist, no extra machinery.
3. **Continuity with the rest of the type system**. Contracts and refinements are the **third leg** of "information in the type, zero cost at runtime", together with algebraic effects and units of measure. All three use the same pattern: declare in the signature or the type, check statically when possible, runtime when needed. Eiffel and Ada don't have algebraic effects or UoM, so their contracts live more alone.

11.8 What kaikai doesn't do, and why

It's worth enumerating what kaikai **deliberately** doesn't support:

- **No SMT solving**. If your contract is `ensures result.entries == sort(c.entries)`, kaikai won't prove statically that your body in fact sorts the list. It will check it at runtime.
- **No arbitrary refinements over complex structures**. `Real where 0.0 <= self <= 1.0` is legal. `[Int] where list.length(self) > 0` isn't implemented in the initial version. The restriction can be expressed with a sum type (`type NonEmptyList = ...`) or a wrapper, but not with a direct refinement.
- **No contract inheritance** in Eiffel style. kaikai has no classes or inheritance; contracts live in each individual function's signature.

Why these restrictions? The argument is the same as the whole language: **simplicity and predictability**. A type system that invokes a solver is opaque — the programmer doesn't know why their program compiles or doesn't, and error messages become unintelligible. A bounded system, that proves the obvious and defers the rest, gives weaker guarantees but **understandable ones**, and leaves the program's runtime as a healthy safety net.

11.9 Case study: bank account

We close with a minimal bank account where contracts document and ensure behavior.

```
type Account = {
  holder: String,
  balance: Int,
}

fn open(holder: String, initial_deposit: Int) : Account
  requires initial_deposit >= 0
  ensures  result.balance == initial_deposit
=
  Account { holder: holder, balance: initial_deposit }

fn deposit(c: Account, amount: Int) : Account
  requires amount > 0
  ensures  result.balance == c.balance + amount
=
  Account { ...c, balance: c.balance + amount }

fn withdraw(c: Account, amount: Int) : Account
  requires amount > 0
  requires c.balance >= amount
  ensures  result.balance == c.balance - amount
=
  Account { ...c, balance: c.balance - amount }
```

Three operations, five preconditions, three postconditions. Human reading:

- `open` opens an account with non-negative initial balance. The postcondition confirms the new account's balance is exactly what was deposited.
- `deposit` demands the amount be positive (no zero-or-negative deposits) and promises the final balance is the initial plus the amount.
- `withdraw` demands positive amount and sufficient balance, and promises the final balance is initial minus amount.

What if someone — you, in six months, in a hurry — writes `withdraw(account, 0 - 50)` (passing a negative)? The contract `requires amount > 0` is violated and the program aborts with a message pointing at the exact `requires` line. No silence, no weird behavior, no inconsistent balance: immediate abort and diagnostic.

And if `withdraw`'s body had a bug — someone changes `c.balance - amount` to `c.balance + amount` accidentally — the `ensures result.balance == c.balance - amount` violates on exit and aborts too. The postcondition is your insurance against internal bugs, just as `requires` is your insurance against caller misuse.

With four lines of contracts, this minimal bank account documents its rules, verifies them when running, and leaves a clear diagnostic when they break. Compare with the contractless version:

```
fn withdraw(c: Account, amount: Int) : Account =
  Account { ...c, balance: c.balance - amount }
```

Works in the happy case. But a caller passing a negative ends up with an account whose balance grew (because `c.balance - (-50) == c.balance + 50`), and nobody finds out. A caller passing more amount than balance ends up with a negative-balance account, and nobody finds out. The bugs that in kaikai with contracts are immediate aborts, in kaikai without contracts are silently incorrect balances in production.

Exercises

11.1. Define `type Age = Int where self >= 0 and self <= 130`. Write `fn average_ages(a: Age, b: Age) : Age`. On which line does the verification appear when you build an `Age` from a dynamic value?

11.2. Take the `divide` function from §11.2. Add a postcondition ensuring that when `b > 0` and `a > 0`, the result is non-negative. How does the postcondition look? What happens if your body had a bug returning a negative in some case?

11.3. Rewrite the bank account from §11.9 using a **refinement type** for the balance (`type ValidBalance = Int where self >= 0`), and `Account = { holder: String, balance: ValidBalance }`. Which preconditions and postconditions become redundant with this change? What information does the signature lose?

11.4. Imagine a function `fn percentile(p: Probability, xs: [Real]) : Real`. What preconditions would you add over `xs`? What postconditions document correct behavior? Which of the two forms (refinement type or contract) would you use for each restriction?

11.5. Read `docs/refinements-and-contracts.md` from the kaikai repo. Identify a restriction the doc mentions as "post-MVP". Discuss with a colleague or with an AI agent what consequences implementing it would have — what new class of errors it would catch, what class of programs would become more loaded with checks.

Chapter 12 · Algebraic effects

This is the chapter where kaikai earns its novelty. Up to now we've seen types, pattern matching, protocols, units of measure, contracts. All elegant pieces, none unique to kaikai: you'd find counterparts in Haskell, Rust, F#. **Algebraic effects** are what set kaikai apart from nearly every production-ready language today.

The idea is easy to state and strange at first: **a function declares in its signature which effects it uses, but not how they happen**. Printing to a screen, reading a file, failing with an error, suspending execution, generating a random number — these are all effects. The function says "I need this capability"; code further up decides what "this capability" means in the current context. The separation between **what** and **how** is the heart of the system.

If that sounds like *dependency injection*, hold the thought. If it sounds like exceptions, hold that too. If it sounds like generators, also yes. The reason one mechanism resembles so many things is that, in the theory underneath, all those things are the same. Algebraic effects are the generalisation.

We'll move slowly. This chapter rewards patience more than speed: the first reading is for each idea to register; solid intuition lands when you come back a month later and it suddenly feels obvious.

12.1 The friction effects resolve

Before showing the syntax, let's look at the concrete problems effects solve. If you recognize the pattern from your work, you'll know why a new tool is worth learning.

Invisible exceptions

In Java or Python, any function call can throw an exception and nothing in the signature tells you. You read the code and you don't know what can fail. You find out in production.

```
def load_user(id):
    return db.get(id)   # can this fail? with what exceptions?
```

Languages with checked exceptions (Java) tried to fix it by forcing you to declare `throws`. The result was that people wrote `throws Exception` to silence the compiler, and we were back where we started. Modern functional languages (Rust, OCaml, kaikai's chapter 5) push you toward `Result` or `Option`: the cost of failure shows up in the type, the caller decides

what to do. Good for local failures, heavy when many functions fail: signatures fill up with wrapping and unwrapping.

`async` / `await` infection

In JavaScript, Python, C#, Rust, marking a function `async` forces every function that calls it to be `async` as well. A seemingly local change spreads through the whole call tree.

```
async function read(path) { ... }
async function process(path) {
  const data = await read(path);   // process had to become async too
  return transform(data);
}
```

Bob Nystrom named this in 2015 with his famous essay **"What Color is Your Function?"**. The idea: `async` splits functions into two colors. Red ones (`async`) and blue ones (non-async). A red function can call a blue one, but a blue one can't call a red one. If you have a blue function and you need to use a red one inside, you have to repaint the blue. And the one that called it. All the way up. It's an infection that won't stay local.

The essay's point isn't that `async` is bad. It's that this kind of marker in the signature, when it's specific to one type of effect, creates two parallel systems that don't compose. `async` doesn't compose with generators: you need `async function*`. It doesn't compose cleanly with exceptions (exceptions in async functions become rejected promises). Each new combination needs its own syntax.

Algebraic effects solve the problem at the root: **there are no special colors**, only one dimension — the effect row. `async` doesn't need to be a syntactic property of the function; it's simply one effect among others. And the row extends without new syntax: `/ Async + Fail` is no stranger than `/ Async`.

Dependency injection

To make a function testable, you pass the "external things" it uses as parameters: the clock, the logger, the database client. You pass mocks in tests, the real things in production.

```
class Processor {
  public Processor(Clock clock, Logger logger, DbClient db) { ... }
  public void run() { ... }
}
```

It works, but it pollutes signatures and forces manual wiring. And every time a new function needs one of the services, you have to thread it through the constructors up the hierarchy.

The common pattern

The three frustrations share the same shape:

- A function needs a **capability** (to fail, to suspend, to log).
- The capability must be **explicit** (visible in the type) so it doesn't surprise.

- The capability must be **provided by context** (caller, test, framework) without the function knowing.
- Multiple capabilities must **compose cleanly**.

Algebraic effects are **one construct** that covers all four. What exceptions, `async` / `await`, and dependency injection do separately and often poorly, effects do with a single mechanism.

12.2 Declaring an `effect`

An effect is an **interface**. It declares what operations exist, with what signatures, but not how they're implemented.

```
effect Log {
  log(msg: String) : Unit
}
```

This introduces an effect named `Log` with one operation, `log`, that takes a string and returns nothing useful. Any function that calls `Log.log(...)` is using the `Log` effect.

What it does **not** introduce: implementation. There's no body here, no `if`, no print. Just the signature. That's the fundamental difference with a protocol or a traditional interface: the effect decides nothing, it only declares what can be requested.

Operations in an effect are declared without the `fn` keyword and without a body — just name, parameters, return type. The same effect can have several operations:

```
effect Io {
  print(s: String)  : Unit
  read_line()       : String
}
```

12.3 Calling an operation: the signature changes

To use an operation, call the effect as if it were a namespace and the operation a method:

```
fn greet(name: String) : Unit / Log {
  Log.log("hello, " ++ name)
}
```

Two new things:

- **`Log.log(...)`** is the invocation syntax. The effect is the namespace; the operation is the method.
- **`: Unit / Log`** in the signature. The slash introduces the **effect row**. It's the list of effects the function needs to run. Without `/ Log`, the function doesn't compile: it's using a capability it didn't declare.

The type system guarantees that **every function using an effect declares it**. If you call `Log.log(...)` from a function without `/ Log`, the compiler rejects it with a clear message. There's no escape: effects are visible in the signature, always.

And this is recursive. If `greet` uses `Log` and `main` calls `greet`, then `main` also needs `Log` in its signature, unless it **handles** the effect first (that's §12.4).

```
fn main() : Unit / Log {      # propagates the effect
  greet("kaikai")
}
```

This is the same contagion principle we saw with `async` / `await`, but without the color problem: there's no special syntax to "pay for the effect" at the call site. `Log.log(...)` is a call like any other. The signature is where the discipline lives, and adding a new effect doesn't introduce a new incompatible color: it only extends the row.

Several effects: the row

If a function uses more than one effect, list them with `+`:

```
fn process() : Int / Log + Fail {
  Log.log("start")
  if bad_condition() {
    Fail.fail("can't")
  }
  42
}
```

`Log + Fail` is the **row** of effects. Order doesn't matter: the row is a set, not a sequence. `Log + Fail` and `Fail + Log` are the same row to the compiler. The `+` operator is just syntax for building it.

12.4 Handling an effect with `handle ... with`

The interesting part: **deciding what an effect means** at a specific point in the program. That's what `handle ... with` does.

```
fn main() {
  handle {
    greet("kaikai")
    greet("Ada")
  } with Log {
    log(msg, resume) -> {
      println("[INFO] " ++ msg)
      resume(())
    }
  }
}
```

Read literally: "run this block, and whenever someone inside invokes `Log.log`, do this". The handler intercepts each call to `log`, decides what happens, and resumes with `resume(...)`.

Three details worth pinning down:

- `handle { body } with Effect { clauses }` is a control construct, in the same family as `if` and `match` — not a function call. `handle` and `with` are reserved keywords.
- **`body` is where `Effect` is handled.** Inside, calling `Log.log(...)` is legal even if the surrounding function doesn't have `Log` in its row, because the handler provides it. Outside the `with`, the type system demands the effect again.
- **`resume(())` continues the body** from the point where `log` was called, with the value you passed as argument.

The remarkable thing: **`main` doesn't need to declare `Log` in its signature**. The `handle` "consumes" it: the inside uses it, the `with` provides it, and `main`'s effect row comes out without `Log`. The type system stays strict, but the handler is the door through which an effect leaves the scope.

The same body with two handlers

The power shows when you notice that `greet` doesn't change between runs. One handler gives you verbose logs; another gives you silence:

```
# Verbose handler
handle {
  greet("verbose mode")
} with Log {
  log(msg, resume) -> {
    println("[INFO] " ++ msg)
    resume(())
  }
}

# Silent handler
handle {
  greet("silent mode")
} with Log {
  log(msg, resume) -> resume(())   # drops the message
}
```

`greet` doesn't notice the difference. The same is true for a handler that writes to a file, one that accumulates messages into a list, one that ships them over the network. The function `greet` is **handler-agnostic**.

This is what replaces dependency injection. No constructor to pass around, no service to mock: the handler IS the implementation.

12.5 `resume`: the handler decides what happens next

`resume` is the piece that confuses people most at first, and the one that pays off most once you get it. It's the **continuation** of the body from the point of the operation.

When the body invokes `Log.log("hi")`, control jumps to the handler. The handler receives two things:

1. The argument passed to the operation: `"hi"`.
2. A `resume` function that, when called, continues the body from where it left off.

```
log(msg, resume) -> {
  println("[INFO] " ++ msg)   # decide what to do with the effect
  resume(())                  # hand control back to the body
}
```

`resume(())` passes `()` as the return value of the operation `log` (which returns `Unit`). The body continues after the `Log.log(...)` call with that value.

Operations that return values

`Log.log` returns nothing useful, but other operations do. An effect can **supply** a value:

```
effect Ask {
  name() : String
}

fn greeting() : String / Ask {
  "hello, " ++ Ask.name()
}

fn main() : Unit / Stdout {
  let message = handle {
    greeting()
  } with Ask {
    name(resume) -> resume("world")
  }
  println(message)   # prints "hello, world"
}
```

`Ask.name()` suspends the body. The handler receives `resume` and decides what `String` to give the caller: here, `"world"`. `resume("world")` continues the body with that value, and the `++` concatenates it.

This replaces many uses of dependency injection: instead of threading `name` as a parameter through the whole call tree, you "ask" for it via an effect, and the handler in `main` decides the answer. In tests, a different handler answers something else.

Handlers that DON'T call `resume`

If the handler **doesn't** call `resume`, the body is discarded and the handler's value becomes the value of the entire `handle`. This is how exceptions are built:

```
effect Fail {
  fail(reason: String) : Nothing
}

fn divide(a: Int, b: Int) : Int / Fail {
  if b == 0 { Fail.fail("division by zero") }
  else      { a / b }
}

fn main() : Unit / Stdout {
```

```
let r = handle {
  let x = divide(10, 2)
  let y = divide(20, 0)    # Fail.fail fires here
  x + y                    # never reached
} with Fail {
  fail(reason, resume) -> {
    println("failed: " ++ reason)
    0                      # replacement value
  }
}
println("result: #{r}")    # prints: result: 0
}
```

The signature `fail(reason: String) : Nothing` says the operation **never returns**. `Nothing` is kaikai's empty type (chapter 3's bottom type): no inhabitants, no value of `Nothing` can be constructed. By construction, there's nothing to pass to `resume`, so the type system guarantees you can't continue the body after `Fail.fail`. The program doesn't break if you try; the compiler doesn't let you write the code.

That's the key: kaikai's "exceptions" are a special case of the general mechanism. There's no special syntax for `try/catch`; there's `handle` and an effect whose operation returns `Nothing`.

12.6 Handlers with state: the `State` pattern

So far our handlers have been stateless: they decide what to do and resume. But a handler can carry **its own state** without the body noticing. This replaces global mutation.

```
effect State[T] {
  get() : T
  set(v: T) : Unit
}

fn sum(xs: [Int]) : Int {
  handle {
    list.foreach(xs, (x) => State.set(State.get() + x))
    State.get()
  } with State[Int](0) {
    get(resume)    -> resume(state)        # return state
    set(v, resume) -> resume((), v)        # update state
    return(x)      -> x                    # drop state at the end
  }
}
```

Three new things:

- **`State[T]` is parametric.** The type `T` is what state holds. Here it'll be `Int`.
- **`with State[Int](0)`** installs the handler with initial state `0`. The argument in parentheses is the initial value.
- **`state`** is a special identifier available inside the handler's clauses. It refers to the current value of the state.

- `resume(value, new_state)` has two arguments when the handler carries state: the operation's return value and the new state. Plain `resume(value)` leaves the state unchanged.
- `return(x) -> x` runs when the body completes normally. `x` is the body's result. Here we drop the final state and return only `x`; if you wanted both, you'd write `return(x) -> (x, state)`.

From the body's view, there's no mutation: only invocations to pure-looking operations. Mutation lives entirely inside the handler. From outside the `handle`, nothing shows.

This pattern is generic: with the same shape you build `Reader` (read-only environment), `Writer` (output accumulation), counters, caches, sessions. All without touching global variables, all without threading parameters.

12.7 `var`, `Ref[T]` and `Array[T]`: two distinct mechanisms

`State[T]` is the general tool for carrying a changing value, but writing `handle ... with State[Int](0)` every time you want a local counter would be tedious. So kaikai ships syntactic sugar and, separately, a stdlib effect for cases where memory outlives a block. They're two distinct constructs and worth keeping apart.

`var`: sugar over `State[T]`

The short form for a local cell:

```
fn count_evens(xs: [Int]) : Int {
  var n = 0
  list.foreach(xs, (x) => {
    if x % 2 == 0 {
      n := @n + 1
    }
  })
  @n
}
```

Three new forms:

- `var n = 0` declares the cell with its initial value.
- `@n` reads the current value.
- `n := v` writes `v`.

How does it work? **`var` is syntactic sugar over `State[T]`.** The compiler rewrites

```
var n = 0
... rest of the block ...
```

into

```
handle {
  ... rest of the block ...
} with State[Int](0) as n {
```

```
  get(resume)    -> resume(state)
  set(v, resume) -> resume((), v)
  return(x)      -> x
}
```

Because the inserted `handle` lives **inside the same block** where the `var` appears, the `State[Int]` effect closes right there and never escapes the function's signature. From the caller's side, `count_evens` is just `: Int`. No effects.

The effect isn't being masked: the `handle` is literally next to the `var`. The row closes at the exact spot where the cell is declared.

And it isn't slow: the compiler detects the pattern "local cell with one-shot resume" and specializes it down to a stack frame slot, equivalent to a C mutable variable. Zero cost compared to the imperative code you'd otherwise write.

`Mutable`: the effect behind `Ref[T]` and `Array[T]`

`var` covers local cells. But there are cases where memory needs to **outlive a block**: an array you'll return, a cell you'll pass between functions, a structure shared by several routines. For those cases kaikai ships two stdlib types, `Ref[T]` and `Array[T]`, and both live under the `Mutable` effect.

```
fn fill(n: Int) : Array[Int] {
  let a = Mutable.array_make(n, 0)
  var i = 0
  list.foreach([0..n], (_) => {
    a[i] := @i * 2
    i := @i + 1
  })
  a
}
```

- `Mutable.array_make(n, init)` creates an array of size `n` with initial value `init`.
- `a[i]` reads index `i`. Sugar for `Mutable.array_get(a, i)`.
- `a[i] := v` writes index `i`. Sugar for `Mutable.array_set(a, i, v)`.

`Ref[T]` is the single-cell version: `Mutable.ref_make(v)`, `Mutable.ref_get(r)`, `Mutable.ref_set(r, v)`. No indexing sugar, but everything else is parallel to `Array[T]`.

Look at the signature of `fill`: it says `: Array[Int]`, **without** `Mutable`. Why, given the function obviously mutates?

Because `Mutable` follows the discipline of **observable effects**: the effect shows up in the signature only when the mutation is **visible to the caller**. And here it isn't. The array is created inside, filled inside, and returned only when it's complete. Whoever receives it gets a settled value; they don't observe any mutation.

When `Mutable` becomes visible

If the mutation is **observable**, the effect appears:

```
fn fill_in_place(a: Array[Int]) : Unit / Mutable {
  let n = Mutable.array_length(a)
  var i = 0
  list.foreach([0..n], (_) => {
    a[i] := @i * 2
    i := @i + 1
  })
}
```

Here `a` comes from outside. The caller holds a reference to the same array we're modifying. The mutation is visible to the caller, and the signature must declare it.

The rule from §12.3 still applies, just for any other effect: the type system guarantees that every function producing an observable effect declares it. Hidden assignments don't exist.

`Mutable` versus `State[T]`

Both represent mutable state. When to use which?

- `var` **(which is** `State[T]`**)**: the cell lives inside the block. No need to survive the function, no need to be passed to other routines — just a local accumulator or counter. Signature stays clean.
- `Mutable` **with** `Ref[T]` **or** `Array[T]`: memory outlives the block or is shared across functions. Appears in the signature whenever the mutation is observable to the caller.

If you pass a `Ref[T]` or `Array[T]` as argument, or return one after mutating it, you're in `Mutable` territory. If you just need a local counter, it's `var`.

12.8 Composing effects: nested handlers

Real functions use more than one effect. One that logs and accumulates can declare `/ Log + State[Int]`, and in `main` you handle it with two nested `handle`s:

```
fn accumulate(xs: [Int]) : Int / Log + State[Int] {
  list.foreach(xs, (x) => {
    Log.log("adding #{x}")
    State.set(State.get() + x)
  })
  State.get()
}

fn main() : Unit / Stdout {
  let total = handle {
    handle {
      accumulate([10, 20, 30])
    } with State[Int](0) {
      get(resume)    -> resume(state)
      set(v, resume) -> resume((), v)
      return(x)      -> x
    }
  } with Log {
```

```
    log(msg, resume) -> {
      println("[LOG] " ++ msg)
      resume(())
    }
  }
  println("total = #{total}")
}
```

Nesting order matters **when the handlers interact**. Here they don't: `State` only reads and writes its state, `Log` only prints. Either order works. But with `Fail` inside `State`, the order decides whether state survives a failure (outer `Fail`) or is discarded (inner `Fail`). Each combination has an explicit, checkable semantics.

This is a deep difference with `try/catch + global variables`: there, the order is implicit and runtime-dependent. Here it's explicit and you decide it in the signature of the `handle`s.

12.9 Effect row aliases

When a combination appears many times, name it with `type`:

```
effect Log   { log(msg: String) : Unit }
effect Audit { audit(user: String, action: String) : Unit }

type Tracing = Log + Audit
```

From there on, writing `: Unit / Tracing` is the same as writing `: Unit / Log + Audit`. The alias is **transparent**: it introduces no new effect, only shortens the row.

```
fn make_purchase(user: String, amount: Int) : Unit / Tracing {
  Log.log("start purchase")
  Audit.audit(user, "purchase($#{amount})")
  Log.log("end purchase")
}
```

One restriction: aliases must be **closed**. You can't write `type WithIo[e] = Io + e` (with a row variable). The restriction sidesteps unification complications the compiler doesn't need to pay for.

12.10 Default handlers: the effect carries its own

Every `handle` we've seen so far was written by hand. But there are effects where one of the implementations is so obvious that making the user write it every time is pure ceremony: if your effect is `Log` and the "reasonable" implementation writes to stderr with a timestamp, you'd want that implementation to ship with the effect.

Kaikai lets you declare a **`default { }` block** inside the effect declaration. It's exactly like a `handle ... with`, but it lives next to the operations and the compiler installs it around `main` when nobody handles the effect by hand.

```
effect Log {
  info(msg: String) : Unit
  warn(msg: String) : Unit

  default {
    info(msg, resume) -> $extern_handler("kai_default_log_info")
    warn(msg, resume) -> $extern_handler("kai_default_log_warn")
  }
}
```

The clauses inside the `default` block have the **same shape** as the ones in a `handle`: op name, parameters, `resume`, arrow, body. What changes is where they live and who fires them. If `main()` declares `: Unit / Log` and no `handle ... with Log` wraps it, the compiler emits code equivalent to:

```
handle {
  original_main()
} with Log {
  info(msg, resume) -> ...   # the default block's clauses
  warn(msg, resume) -> ...
}
```

The user doesn't write that wrapping. The compiler derives it from the `default` block and emits it at program entry.

`$extern_handler`: the sigil and the C bridge

The body of each clause above is `$extern_handler("kai_default_log_info")`. That needs an explanation.

`$` is a **sigil**: a character that marks a special syntactic form. In kaikai it introduces a **compiler intrinsic**, a construct the compiler resolves directly instead of looking up a function defined in kaikai code. The general form is `$name(args)`. Today there's just one: `$extern_handler`. There may be more tomorrow; the sigil is reserved for that family.

`$extern_handler("kai_default_log_info")` means: "the body of this clause is a call to the C symbol `kai_default_log_info`". When the effect fires, the compiler doesn't look for a kaikai function by that name; it emits a direct call to the C runtime linked into the program.

This is the bridge between high-level algebraic effects and the concrete world — files, sockets, syscalls. OS primitives live in C; effects live in kaikai; `$extern_handler` joins them.

When the default fires and when it doesn't

The lookup rule is the one from §12.4 with one extra step at the bottom:

1. The nearest `handle ... with Eff` that covers the op wins.
2. If no enclosing handle exists and the op is in `main`'s row, the compiler installs the effect's default.
3. If even the default doesn't cover the op, the compiler rejects the program when typechecking `main`.

Note the detail: the default **only** fires when the op would escape to `main`. If your function is inside a `handle`, the handle wins, not the default. There's no ambiguity, no surprising precedence: nearest wins, always.

Looking at the original example again

This explains why `println` compiles without every signature carrying `/ Stdout`:

```
fn hi() {
  println("hi")
}

fn main() {
  hi()
}
```

`Stdout` ships from stdlib with a `default` block whose clauses call `$extern_handler("kai_default_stdout_print")` and friends. The C symbol writes to the process's real `stdout`. Since `main` doesn't handle `Stdout`, the compiler installs that default and the program prints.

When you want to control the output (silence in tests, redirect to a file, capture it for assertions), you write your own `handle ... with Stdout` and inside the block the runtime default doesn't participate. **Nearest wins**: the `handle` you wrote is closer than the implicit wrapping at `main`'s entry.

Your own default: the complete example

If you declare an effect and equip it with a default, programs that just call `main` look just as simple:

```
effect MyLog {
  info(msg: String) : Unit
  default {
    info(msg, resume) -> $extern_handler("kai_default_log_info")
  }
}

fn greet(name: String) : Unit / MyLog {
  MyLog.info("hi, " ++ name)
}

fn main() : Unit / Stdout {
  let r = handle {
    MyLog.info("hello from extern_handler")
    7
  } with MyLog {
    info(msg, resume) -> resume(())   # silence inside the handle
  }
  print("result: #{int_to_string(r)}")
}
```

Inside the `handle ... with MyLog`, the explicit clauses win: the `info` is silenced. If another `main` skipped that `handle`, the default would fire and print via the C runtime. `greet` doesn't know the difference: for it, `MyLog` is whatever the context decided.

When there's no default — the effect without a net

Not every effect carries a `default`. `Fail` is the clear counter-example: if an op can abort the program, you don't want "forgetting to handle it" to be legal. The public declaration of `Fail` (appendix D) has no `default` block, and so a function producing `/ Fail` must be handled somewhere before `main`, or the compiler rejects with a clear message.

The same applies to `State[T]`, `Reader[T]`, `Writer[W]`: generic effects where **no** reasonable implementation exists without context, so making the user write it isn't ceremony, it's discipline.

The mental rule: an effect carries a `default` when there's **one obvious** implementation (write to `stdout`, read the system clock, generate pseudo-random numbers). If "reasonable" depends on the program, there's no default and the user provides it.

Wrapper function: the alternative when there's no default

When an effect doesn't ship a `default`, or when the default exists but your program always wants a different one, the idiomatic pattern is a **wrapper function**:

```
fn with_test_log[A](body: () -> A / MyLog) : A {
  handle {
    body()
  } with MyLog {
    info(msg, resume) -> resume(())   # silence in tests
  }
}

fn main() {
  with_test_log { ->
    greet("kaikai")
    greet("ada")
  }
}
```

This is what stdlib uses for constructs like `try { body }` and `with_state(0) { body }`. The difference from a default is that the wrapping is **explicit in the code**: whoever reads `main` sees the line, opens the function, knows what it does. A default lives in the effect declaration.

If your effect has one reasonable default for production and a different one for tests, expose both as wrapper functions (`with_test_log`, `with_quiet_log`) and let `main` use the default. Whoever writes tests calls the wrapper.

12.11 The stdlib handlers are kaikai code

When a program runs `println("hi")` and it just works, it's easy to imagine the compiler ships a special case for `Stdout`. It doesn't. The handlers the runtime installs around `main` for `Stdout`, `Stdin`, `Random`, `Clock`, `File`, `Env`, `NetTcp`, and the rest are written in **plain stdlib kaikai**: each

one is an `effect … { ops; default { … } }` using the same `$extern_handler` sigil you'd use to connect your effect to C.

```
# stdlib/io/console.kai (schematic form)
effect Stdout {
  print(s: String) : Unit
  default {
    print(s, resume) -> $extern_handler("kai_default_stdout_print")
  }
}
```

The compiler doesn't know `Stdout` by name. It knows **`default` blocks** and **`$extern_handler`**. For `Stdout`, it installs the default the same way it does for your `MyLog`: by walking the AST of the effect declaration, not by reading a hardcoded table.

This wasn't always the case. Until version 0.55, the compiler shipped internal tables with names like `default_stdout_setup`, `default_random_shims`, one entry per stdlib effect. The #533 trilogy (PRs #551, #559, #561 in `kaikai-org/kaikai`) migrated all seventeen builtin effects to `default {}` blocks declared in stdlib and deleted the tables. The motive isn't aesthetic: it's that the AST becomes the single source of truth, and user effects get exactly the same guarantees as stdlib ones. If your effect declares `default {}` with `$extern_handler`, the compiler installs it like a builtin.

The practical consequence: **you can read how `Stdout` is implemented**. It's in `stdlib/io/console.kai` (or the analogous file in the version you're running). It's kaikai code like yours. If a question about default semantics comes up — "what happens if the pipe is closed?", "who catches `EPIPE`?" — the answer lives in the clause `print(s, resume) -> …` or in the C symbol it bridges. There's no secret runtime behavior separate from code you can read.

Worth repeating the rule, to nail down the model: the compiler resolves an effect by checking, in order, (1) the nearest `handle … with`, (2) the effect's `default {}` block if the op escapes to `main`, (3) compile error. The stdlib handlers aren't a fourth category; they're instances of (2).

Why the sigil has an odd name

`$extern_handler` may sound long. The reason is that the sigil is a system, not a single operation. The #533 trilogy introduced `$` as the prefix for a **family** of intrinsics; `$extern_handler` is the first. If kaikai later needs to expose other runtime bridges — asking the current `errno`, calling a platform-specific symbol — they'll live under the same sigil with descriptive names: `$os_name`, `$panic_with_trace`, whatever. Reserving `$<ident>(args)` leaves the door open without reopening the syntactic debate every time.

For your daily work: if you never bridge an effect to C, you'll never write `$extern_handler`. But when you see it in stdlib you know what it is: a clause that hands its body off to a runtime symbol, declared with the same syntax as any other handler.

12.12 Case study: configuration processor

We close with an example that mixes the three patterns we saw: logging, state, failure. The program processes a list of lines with format `key=value`, parses them, logs each step, counts how many succeeded, and aborts if any line has the wrong format.

```
effect Log {
  log(msg: String) : Unit
}

effect State[T] {
  get() : T
  set(v: T) : Unit
}

effect Fail {
  fail(reason: String) : Nothing
}

type Entry = { key: String, value: String }

fn parse(line: String) : Entry / Fail {
  match string.split(line, "=") {
    [k, v] -> Entry { key: k, value: v }
    _      -> Fail.fail("invalid line: '#{line}'")
  }
}

fn process(lines: [String]) : Int / Log + State[Int] + Fail {
  list.foreach(lines, (l) => {
    let e = parse(l)
    Log.log("#{e.key} = #{e.value}")
    State.set(State.get() + 1)
  })
  State.get()
}
```

`process` declares the three effects in its signature and uses them freely: it parses (can fail), logs, accumulates. But it decides nothing about the context it runs in.

In `main`, the three nested handlers decide:

```
fn main() : Unit / Stdout {
  let n = handle {
    handle {
      handle {
        process(["name=ada", "age=42", "role=admin"])
      } with State[Int](0) {
        get(resume)    -> resume(state)
        set(v, resume) -> resume((), v)
        return(x)      -> x
      }
    } with Log {
```

```
    log(msg, resume) -> {
      println("[LOG] " ++ msg)
      resume(())
    }
  }
} with Fail {
  fail(reason, resume) -> {
    println("error: " ++ reason)
    0 - 1
  }
}
println("entries processed: #{n}")
}
```

Output:

```
$ kai run examples/ch12/08_config_parser.kai
[LOG] name = ada
[LOG] age = 42
[LOG] role = admin
entries processed: 3
```

And if a line is invalid, the outer `Fail` catches it, prints the reason, and `n` ends up `-1`. The inner `Log` and `State` already emitted whatever they caught before the failure.

Why is this a good closing example? Because it shows the three patterns cooperating, each contributing something different, and because `process` is **directly testable**: no files, no IO, no mocks. In a test, the three handlers have different implementations: `Log` accumulates messages into a list instead of printing, `Fail` propagates inside a `Result`, `State` starts from whatever value the test wants.

12.13 Philosophy: three ideas worth remembering

If this feels like a lot of pieces, three ideas underpin everything else:

1. **Effects are visible in the type.** If a function can fail, suspend, mutate, or do IO, its signature says so. No invisible exceptions, no infectious `async`, no hidden dependencies.
2. **The handler decides what happens.** A function's body declares it needs a capability. The handler in context decides how to materialize it. That decoupling replaces dependency injection, mocking, global configuration.
3. **Zero cost when unused.** The compiler resolves handlers at compile time when it can (which is most cases), and the generated code is as fast as if you'd written a direct `if`. There's no structural overhead from having effects in the type. Same promise as units of measure and contracts: rich information in the type, efficient code underneath.

Algebraic effects come from academia (Pretnar, Plotkin, Power) and appeared in languages like Koka, Eff and Effekt before kaikai. What kaikai contributes is readable syntax (the `/ Eff` notation in the signature), integration with the rest of the language (rows instead

of lists, aliases), and a default-handler model with no special cases: the stdlib handlers are declared in kaikai with the same shape yours use. But the underlying idea is old and solid.

If after this chapter you're still not comfortable, don't worry. Effects are the piece that takes the longest to land. You'll read this chapter several times. Each reading peels back one more layer.

Exercises

12.1. Write an effect `Clock` with one operation `now() : Int` (milliseconds since start). Write a function `measure` that runs a block and returns how long it took using `Clock`. Then write two handlers: a real one (queries the system clock) and a simulated one (advances a counter). What is the second one for?

12.2. Modify `sum` from §12.6 to return both the total and the number of elements summed, without adding parameters. Hint: change `return(x) -> x`.

12.3. The case study in §12.12 prints `[LOG]` for each entry. Change the `Log` handler so that instead of printing, it accumulates the messages into a list and returns them as part of the final result, along with `n`. Hint: you'll need another `State`.

12.4. Write `fn count_evens(xs: [Int]) : Int` two ways: one with `var` and `list.foreach`, the other without `var`, using `list.filter` and `list.length`. Which feels clearer? Why does the `var` version add no effect to the signature?

12.5. Build an effect `Choice` with one operation `choose(options: [Int]) : Int` that supplies "some" of the options. Write a handler that always picks the first, and another that picks the last. How would the implementation change if you wanted a handler that explored **all** options (backtracking)? Hint: it would need to call `resume` more than once. That's *multi-shot* and lives under `resume_multishot`.

12.6. Take a program you have in another language where you use dependency injection to mock services in tests. Write down in pseudocode which effects you'd declare and what the test vs production handlers would look like. How much of the original code survives unchanged?

12.7. Read the difference between `resume` (one-shot) and `resume_multishot` in the language documentation. Why does kaikai make the common case cheap and force you to mark the expensive case explicitly?

12.8. Declare an effect `MyLog` with operation `info(msg: String) : Unit` and a `default {}` block that bridges to a fictional C symbol `my_log_info_to_stderr` via `$extern_handler`. Then write a wrapper function `with_quiet_log` that silences the messages. A `main` without the wrapper triggers the default; a `main` wrapped in `with_quiet_log` doesn't. Compare with how you'd do the same in a language with classical dependency injection.

12.9. Why doesn't `Fail` carry a `default {}` block? Pick three hypothetical effects (yours or stdlib ones you imagine) and for each decide whether it would have a default. Argue in one line why or why not.

12.10. Read the declaration of `Stdout` in `stdlib/io/console.kai` in the language repo. What does the default's clause do when the pipe is closed (`EPIPE`)? Where does that logic live: in kaikai or in the C symbol it bridges to?

12.11. Chapter 13 covers fibers: concurrent tasks modeled as effects. Note before reading it: what operations would an effect `Spawn` need? What decision would you make as a handler when a child fiber aborts?

Chapter 13 · Concurrency and memory

Concurrency is where most languages accumulate debt. Threads with shared memory lead to races that show up once a month and get fixed three times; `async`/`await` introduces function colors; actors fix the previous problems but historically come with a GC and a heavy runtime.

kaikai bets on an unusual combination: **cooperative fibers + per-fiber memory + Perceus**. The structure has three consequences worth naming before the syntax:

- **No shared memory between fibers.** Each fiber has its own heap. What one passes to another is copied or moved. Goodbye data races by construction.
- **No GC, no borrow checker.** Perceus frees memory when the last use of each value ends, without an asynchronous collector and without asking the programmer to annotate lifetimes. The compiler figures out where to put the `free`s by analysing the program.
- **Concurrency is an effect.** `spawn` is an operation of a `Spawn` effect, not a keyword. Creating and awaiting fibers composes with the rest of the system (`State`, `Fail`, `Cancel`) using the chapter-12 machinery.

Let's take it piece by piece.

13.1 The model: isolated fibers

A **fiber** is a unit of execution similar to a thread, but much lighter: on the order of hundreds of bytes instead of megabytes. A kaikai application can have thousands or hundreds of thousands of live fibers without breaking a sweat.

Fibers are **cooperative**. Each one runs until it hits a **yield point**: a call that voluntarily hands control back to the scheduler. Yield points are explicit:

- `spawn.yield()`: "I've run for a while, try another".
- `spawn.await(f)`: "wait until fiber `f` finishes".
- IO operations that the scheduler intercepts (network reads, sleep, etc.).

Without yields, a fiber runs to completion. That's **local determinism**: inside a block without yields, you know exactly what's happening. Compared to preemptive threads, it

takes away a whole class of bugs: there's no race over data you touch between two yields because nobody can interrupt you.

The trade-off is that a fiber that never yields blocks every other one. It's the programmer's responsibility to add yields where they make sense. In practice, IO calls already include them, and the only case where you have to think about manual yields is a tight CPU-heavy loop.

Per-fiber memory

Each fiber has its **own heap**. When a fiber creates a record, a list, a closure, the space comes from that heap. Another fiber can't touch it: can't read it, can't write it. The type system guarantees this.

How do two fibers communicate, then? By passing values. When a fiber sends a message to another (via an actor mailbox or the result of an `await`), the value is copied to the receiver's heap. For small types that's trivial; for big structures, kaikai uses Perceus to move instead of copy when the sender no longer uses the value.

The guarantee that matters: **there's no way for two fibers to hold a pointer to the same object**. Data races, memory visibility problems, cache coherence bugs — everything that in traditional threads needs `Atomic` reads/writes or locks simply doesn't exist here. Concurrency is by message, not by shared memory.

13.2 Perceus in one page

How is memory freed? Without GC and without a borrow checker, there's a third approach: **strict reference counting driven by Perceus** (Lorenz, Leijen, Reinking, 2021).

The idea is that the compiler analyzes each function to find the exact point where each value is last used. At that point, it inserts an instruction that decrements the value's reference count: if it hits zero, the value is freed; if not, it stays for another use.

```
fn example(xs: [Int]) : Int {
  let n = list.length(xs)   # first use of xs
  let s = list.sum(xs)      # last use of xs: consumed here
  s + n                     # xs no longer exists; n and s do
}
```

Compared to a GC:

- **No pause.** Freeing is synchronous, predictable, part of the generated code.
- **No asynchronous overhead.** The compiler knows the exact lifetime of each value.
- **No separate thread.** The scheduler doesn't compete with a collector.

Compared to a borrow checker:

- **No lifetime annotations.** No `'a`, no `&`, no `mut`.
- **No restrictions on use patterns.** If you need two references to the same value, the compiler inserts the necessary increments and decrements.

The cost? When a value is used many times, counters move. For heavily shared values that adds overhead, and Perceus includes aggressive optimizations to minimize it (reuse in place: if a value is about to be freed and a value of the same shape is needed immediately after, the same memory is reused without touching the counter). In practice the cost is low and predictable.

Why this matters for concurrency: Perceus works per fiber. Each fiber has its own counters, its own frees. There's no synchronization between fibers for any counter: no two fibers ever share pointers to a value with a shared count. That's why the "isolated fibers" model fits so cleanly with Perceus: the same invariant that rules out data races also keeps the reference counting lock-free.

13.3 Creating and awaiting fibers: the basic operations

The simplest way to use fibers is with `spawn.spawn` and `spawn.await`:

```
import spawn

fn worker(tag: String, n: Int) : Unit / Stdout + Spawn {
  if n > 0 {
    println(tag)
    spawn.yield()
    worker(tag, n - 1)
  }
}

fn main() {
  let f = spawn.spawn(() => worker("B", 3))
  worker("A", 3)
  spawn.await(f)
}
```

Output:

```
$ kai run examples/ch13/01_two_fibers.kai
A
B
A
B
A
B
```

Reading literally:

- `import spawn` brings in the fiber operations.
- `spawn.spawn(() => worker("B", 3))` creates a new fiber that will run the lambda when the scheduler picks it.
- `worker("A", 3)` runs in the current fiber (`main`'s).
- `spawn.yield()` inside `worker` hands control back. On each yield, the other fiber gets its turn.
- `spawn.await(f)` waits for `f` to finish before `main` returns.

`Spawn` appears in `worker`'s signature because the function calls `spawn.yield()`, which is a `Spawn` operation. The row propagates upward like any other effect from chapter 12.

Why yields are explicit

In languages with preemptive threads (Java, Go, Rust with `std::thread`), the scheduler can interrupt a thread at any instruction. That forces you to program as if any line could be interrupted by another fiber modifying shared data.

In kaikai, **a fiber keeps running until it hits a yield point**. Between yields, you have local determinism: if you modify a local value, nobody else will touch it until you give up control. This drops a lot of cognitive load.

In exchange, you have to **remember to yield**. The mental rule: if your function has a long pure-CPU loop, add a `spawn.yield()` every so many iterations. IO functions already yield internally.

13.4 Nurseries: structured concurrency

`spawn.spawn` + `spawn.await` works, but it has a problem: if you forget the `await`, the fiber outlives the scope that created it. And if that fiber fails, you find out late or not at all.

Nurseries tie fibers to a lexical scope. A fiber can only live inside a nursery, and the nursery waits for all its children before exiting.

```
import spawn

fn worker(tag: String, n: Int) : Unit / Stdout + Spawn {
  if n > 0 {
    println(tag)
    spawn.yield()
    worker(tag, n - 1)
  }
}

fn main() : Unit / Stdout + Spawn + Cancel {
  nursery { n ->
    let a = n.spawn(() => worker("A", 3))
    let b = n.spawn(() => worker("B", 3))
    n.await(a)
    n.await(b)
  }
}
```

`nursery { n -> ... }` opens a scope. Inside, `n` is the capability to create and await fibers:

- `n.spawn(f)` creates a child fiber. Returns a `Fiber[T]` where `T` is what `f` returns.
- `n.await(f)` waits for that fiber and returns its value.
- `n.select([a, b, ...])` waits for any one to finish and cancels the others.
- `n.cancel(f)` cancels a specific fiber.
- `n.cancel_all()` cancels all children.

What the nursery guarantees:

- **By block exit, all children have finished.** No leaks: a fiber doesn't outlive the `nursery` that created it.
- **If a child fails with an unhandled effect, the others are canceled.** The nursery collects the cause and re-raises it.
- **If the nursery is canceled from outside, the cancellation propagates to all children.**

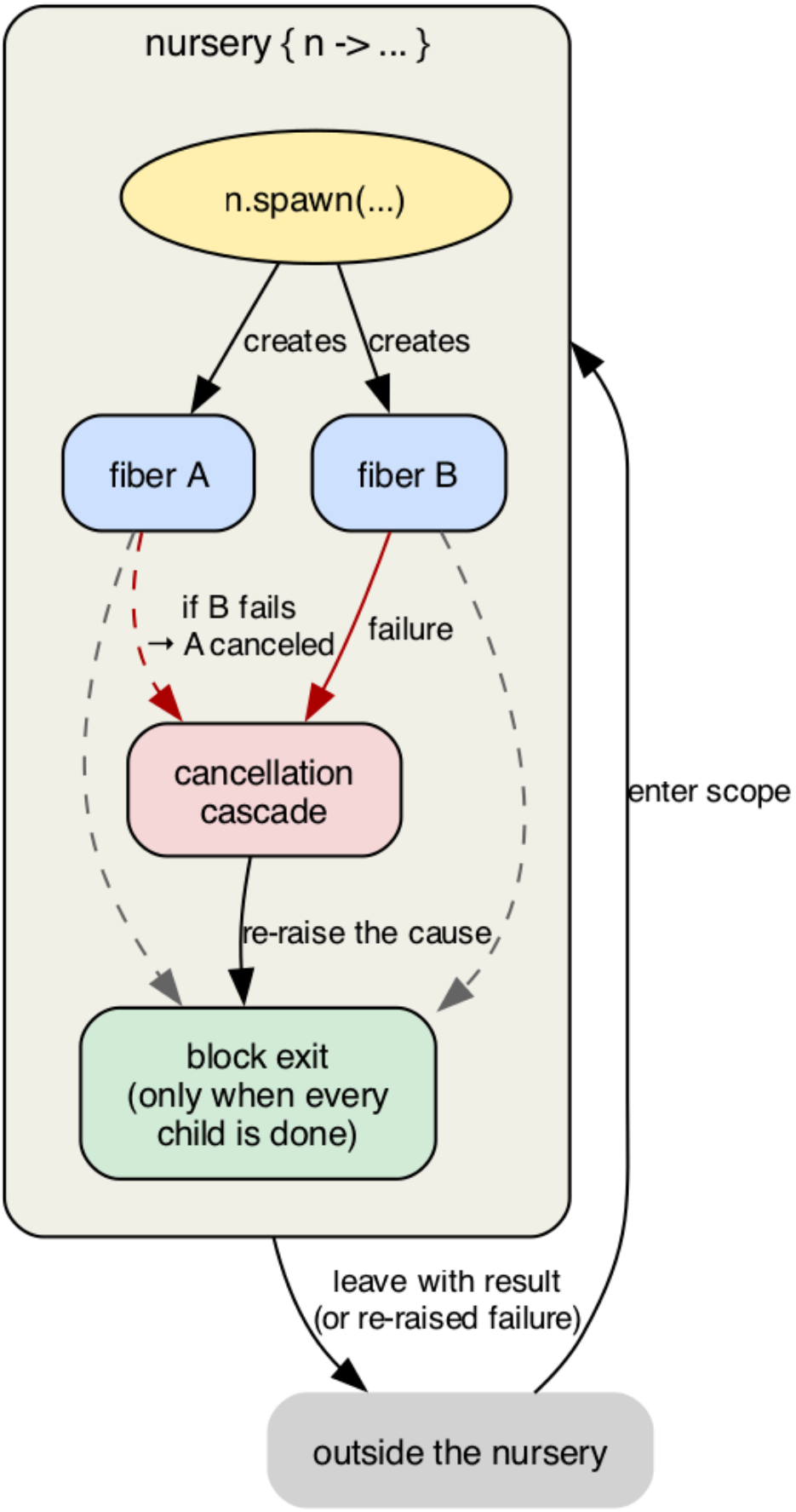
nursery { n -> ... }
n.spawn(...)
creates
creates
fiber A
fiber B
if B fails
→ A canceled
failure
cancellation
cascade
re-raise the cause
block exit
(only when every
child is done)
enter scope
leave with result
(or re-raised failure)
outside the nursery

Figure 13.1 · *Structured concurrency in one picture. The nursery is a lexical scope; child fibers live inside; nothing escapes. If one child fails, the nursery cancels the rest before re-raising; if the parent is canceled from outside, the cascade flows down.*

This is called **structured concurrency**. The idea is Nathaniel Smith's, in his essay *"Notes on structured concurrency, or: Go statement considered harmful"* (2018), and appears also in Trio, Kotlin coroutines, Swift, and OCaml 5 Eio. kaikai's version integrates the pattern into the effect system: the `Spawn` capability is only available inside a nursery, and that's what the type system enforces.

Why `Cancel` appears in `main`'s signature

Notice that `main` declares `/ Stdout + Spawn + Cancel`. Why `Cancel`? Because each `spawn`, `await`, and `select` is a yield point, and every yield point can receive a `Cancel.raise()` from the scheduler (if someone cancels the nursery from outside, or if a sibling fiber fails). Every function using `Spawn` implicitly carries `Cancel`.

`nursery` is sugar over `handle`

`nursery { n -> ... }` looks like a language keyword but it isn't. Fibers are **an effect** called `Spawn`, declared in the stdlib:

```
effect Spawn {
  spawn[T, e](f: () -> T / e) : Fiber[T]
  await[T](f: Fiber[T])       : T
  select[T](fs: [Fiber[T]])   : T
  yield()                     : Unit
  cancel[T](f: Fiber[T])      : Unit
}
```

It's an ordinary effect, declared the same way `Log` or `State[T]` were in chapter 12. And `nursery { n -> body }` is rewritten at compile time to `handle { body } with Spawn as n { ... }`, with an internal handler that manages the child tree, waits for pending children on exit, and propagates failures.

That means the language core **has no concurrency primitives**: it has effects. Fibers, nurseries, and cancellation are a library built on two stdlib effects (`Spawn` and `Cancel`). Chapter 14 will do the same for actors (effect `Actor[Msg]`), and the pattern repeats: what's distinctive about kaikai isn't the list of constructs, but that **all of them are the same construct** (algebraic effects) under different names.

13.5 Cooperative cancellation

Cancellation in kaikai is **cooperative**: the scheduler doesn't kill a fiber outright. It delivers a `Cancel.raise()` at the next yield point. The fiber can:

- **Unwind cleanly.** If it doesn't handle `Cancel`, the unwinding pulls it out of any `handle`, `nursery`, etc., and the cancellation handlers above take control.

- **Handle `Cancel` to clean up.** The fiber installs a `Cancel` handler that runs the cleanup (close files, release connections) and does NOT call `resume`, letting the unwinding continue.

```
fn long_worker(tag: String) : Unit / Stdout + Spawn + Cancel {
  handle {
    count(tag, 0)
  } with Cancel {
    raise(resume) -> {
      println("#{tag}: canceled, doing cleanup")
      # not calling resume: the fiber unwinds
    }
  }
}

fn count(tag: String, n: Int) : Unit / Stdout + Spawn + Cancel {
  println("#{tag}: #{n}")
  spawn.yield()
  count(tag, n + 1)
}
```

`long_worker` counts indefinitely. If the nursery cancels it, the `Cancel` handler prints its message and the fiber exits. It doesn't hang, it doesn't kill the process.

The conceptual key: `Cancel.raise()` is the **operation**, and `handle ... with Cancel { ... }` is the handler. Same pattern as chapter 12: cancellation is another effect, with a handler written by the user or installed by the runtime.

13.6 Per-fiber mutable memory

Chapter 12 §12.7 covered `var` (local cells, sugar over `State[T]`) and the `Mutable` effect (which rules `Ref[T]` and `Array[T]` when the mutation is observable). All that machinery works the same in sequential code, with one addition that comes from the fiber model: **mutable memory lives in the heap of the fiber that created it**.

When a fiber creates an `Array[T]` or a `Ref[T]`, the space comes from its own heap. Another fiber has no way to reach that memory: no shared pointers, no reference passing between fibers. If a fiber wants to give a mutable value to another, it sends it through a mailbox (chapter 14) and the runtime moves the contents to the receiver's heap.

This is the piece that ensures mutation doesn't introduce data races in kaikai. In a language with threads and shared memory, a mutable `Array[T]` needs locks or atomics to be touched from several threads. In kaikai, the type system guarantees that no `Array[T]` is being modified by two fibers at the same time, because no `Array[T]` is reachable from two fibers at the same time. The memory isolation of §13.1 covers mutable cells too.

13.7 Why fibers can't escape their nursery

A detail of the type system: **`Fiber[T]` is not a movable value**. You can't return it from a function, can't store it in an `Option` or a record, can't pass it to another fiber.

```
fn doesnt_compile() : Fiber[Int] {      # ERROR
  nursery { n ->
    n.spawn(() => 42)                   # can't return
  }
}
```

Why? Because a fiber only makes sense inside the nursery that created it. If you could return it, who would await it? Who would cancel it if the nursery ends? The structure breaks.

A list of fibers inside the same nursery is legal:

```
nursery { n ->
  let fibers = [1, 2, 3] | (x) => n.spawn(() => process(x))
  fibers | (f) => n.await(f)
}
```

But that list lives inside the nursery. It can't escape.

This closes the model: each fiber has a known parent, each fiber ends before its parent ends, and the type system guarantees it syntactically, not by convention. **There's no way for a fiber to be orphaned.**

13.8 Case study: task queue with a worker pool

We close with a classic pattern: a task queue served by a pool of worker fibers. Each worker takes the next task, processes it, and goes back for another. When the queue is empty, they all exit.

```
import spawn

effect State[T] {
  get() : T
  set(v: T) : Unit
}

fn next() : Option[String] / State[[String]] {
  match State.get() {
    []          -> None
    [h, ...rest] -> {
      State.set(rest)
      Some(h)
    }
  }
}

fn worker(id: Int) : Unit / Stdout + Spawn + State[[String]] {
  match next() {
    None       -> println("worker #{id}: queue empty, exiting")
    Some(task) -> {
      println("worker #{id}: processing '#{task}'")
```

```
      spawn.yield()
      worker(id)
    }
  }
}
```

Up to here, pure effect code: three operations (`get`, `set`, `next`), a recursion that ends when the queue is empty. No locks, no atomics, no `Arc<Mutex<...>>`.

The `main` installs the handlers and starts the pool:

```
fn main() : Unit / Stdout + Spawn + Cancel {
  handle {
    nursery { n ->
      let a = n.spawn(() => worker(1))
      let b = n.spawn(() => worker(2))
      let c = n.spawn(() => worker(3))
      n.await(a)
      n.await(b)
      n.await(c)
    }
  } with State[[String]](["alpha", "bravo", "charlie", "delta", "echo", "foxtrot"]) {
    get(resume)    -> resume(state)
    set(v, resume) -> resume((), v)
    return(x)      -> x
  }
  println("(all tasks processed)")
}
```

Output:

```
$ kai run examples/ch13/06_task_queue.kai
worker 1: processing 'alpha'
worker 2: processing 'bravo'
worker 3: processing 'charlie'
worker 1: processing 'delta'
worker 2: processing 'echo'
worker 3: processing 'foxtrot'
worker 1: queue empty, exiting
worker 2: queue empty, exiting
worker 3: queue empty, exiting
(all tasks processed)
```

Three fibers share six tasks under cooperative concurrency. Each one accesses the "shared queue" via the `State` effect, but underneath there's no shared memory: the `State` handler lives in `main`, and the fibers' operations are messages to that handler. The queue is serialized by construction.

Concurrency, not parallelism

A precise word matters here. kaikai in v1 runs **on a single OS thread**: one scheduler, one ready queue. Fibers interleave cooperatively, but never two fibers execute instructions at the same wall-clock moment. That's **concurrency**, not **parallelism**.

Why does this matter? Because if your program is CPU-bound (numerical computation, compression, rendering), running it with a hundred fibers won't make it faster: they'll take turns on the same core. For problems like that, fibers give you structure (a natural way to express concurrent work, cancellation, timeouts) but no speedup.

Where fibers do pay off in speed is when the bottleneck is **IO**: reading files, waiting on the network, waiting on messages. While one fiber is blocked waiting for bytes, other fibers run. The same core uses the time that would otherwise be idle.

Real parallelism (several physical cores working at once) requires multi-threading, which is out of v1's scope. When it lands, it'll be on the same fibers-and-actors model: one scheduler per thread, cooperative fibers inside, messages between threads. For now, the guarantee is that any code you write today with fibers and actors will keep working when multi-threading arrives, just faster on multi-core machines.

13.9 Philosophy: two invariants worth remembering

If you want to keep two things from this chapter, let it be these:

1. **Each fiber has its own heap.** No shared memory between fibers. Communication is by message (actor mailbox, `await` result). Data races don't exist by construction, not by discipline.
2. **Each fiber has a life tied to a lexical scope.** It can't escape the `nursery` that created it; the type system rejects any attempt. If the parent ends, every child has ended. If a child fails, the siblings are canceled.

These two invariants support each other. Memory isolation is what lets Perceus run per fiber without synchronization. Lexical structure is what lets memory be freed predictably at scope exit. Any model that breaks one of the two breaks the other.

In return, you get whole classes of bugs you no longer have to think about:

- No `volatile`, no `Atomic`, no memory ordering.
- No `lock`, no `mutex`, no `RwLock`.
- No borrow checker, no `'a` lifetimes, no `Rc<RefCell<T>>`.
- No `async fn`, no `Future`, no function colors.
- No GC pause-the-world.

It's a trade-off: you give up the freedom to have arbitrary pointers between fibers. But the gain — in safety, in predictability, in mental simplicity — is what justifies the model.

Exercises

13.1. Modify the §13.3 example (two cooperative fibers) so that `worker("A")` does 5 iterations and `worker("B")` does 2. How does the output change? What happens if you remove the `spawn.yield()`s from one worker only?

13.2. A fiber creates an `Array[Int]` locally and modifies it with `a[i] := v`. Then it exits without passing it to anyone. Why doesn't this program introduce data races even if another fiber is running concurrently? Answer in two lines, in terms of the per-fiber memory model from §13.1.

13.3. Implement a function `with_timeout[T](ms: Int, f: () -> T / Spawn) : Option[T] / Spawn + Cancel + Time`. Use `n.select` to run `f` against a fiber that does `Time.sleep(ms)` and returns `None`. Hint: you'll need a local sum type to distinguish "completed" from "timeout".

13.4. In §13.8, the `State` is a FIFO queue with no priorities. Modify the example so some tasks have high priority and the workers process those first. Hint: the `State` can be a record with two lists.

13.5. A fiber that enters an infinite loop without `spawn.yield()` blocks all the others. Write that code and observe what happens. Then add yields every N iterations. How often? How do you choose N?

13.6. In your usual language, find a concurrent program you wrote or maintain. Count how many lines are "real work" (the program's logic) vs how many are "concurrency plumbing" (mutex, queues, atomics, async/await, callbacks). Estimate what percentage of the code would remain under kaikai's model.

Chapter 14 · Actors

Chapter 13 showed how to start fibers and coordinate them with a nursery. That solves internal concurrency: many units of work inside the same program, sharing CPU cooperatively.

For many cases that structure is enough. But there's a pattern that pure fibers leave awkward, and it's worth naming before the syntax.

An actor is a process; a fiber is a computation

Imagine two different tasks:

- **Task A:** parse a large file and return the list of errors found. Starts, works, ends, returns a value.
- **Task B:** maintain an in-memory cache that answers queries (`get(key)` and `put(key, value)`). Starts, stays alive, answers messages for as long as the program lives, ends when someone asks it to stop.

Both are concurrent in the sense that the main program can keep working while they happen. But their shape is very different.

Task A is a **computation**: it has an input, produces an output, ends. That's a **fiber**. You create it with `spawn`, you wait for its result with `await`, you receive the value. Once the result is returned, the fiber ceases to exist.

```
let f = spawn.spawn(() => parse_file("input.txt"))
# ... other work in parallel ...
let errors = spawn.await(f)   # one value, and you're done
```

Task B is a **live process**: it has no single return value, just an endless sequence of interactions. For that, kaikai borrows the **actor** model, drawn in spirit from Erlang and BEAM.

```
let cache = spawn_actor(() => cache_loop())
Actor.send(cache, Put("user:42", "ada"))
Actor.send(cache, Put("user:43", "turing"))
match Actor.receive() {
  Found(v) -> ...
```

```
  Missing -> ...
}
```

An actor is **a fiber with a typed mailbox on top**. The fiber is the substrate (a cooperative thread of execution); the mailbox is what makes it an actor (a channel where messages pile up for the fiber to process in order).

Side by side:

Aspect	Fiber	Actor
What is it?	Unit of execution	Fiber with typed mailbox
Communication	One return value via `await`	Messages via `send` / `receive`
Lifecycle	Start, compute, return, die	Start, loop processing, die when it decides
How to create	`spawn.spawn` or `n.spawn`	`spawn_actor`
How you talk to it	`await` to get its `T`	`send` any number of times
When to pick	Discrete concurrent computation	Long-lived service, internal state, queries

The mental rule:

- **Does the task end with a value the parent needs?** Fiber.
- **Does the task live and respond to messages from several clients?** Actor.

Both models are **concurrent but not parallel** in v1: the runtime runs a single OS thread, and fibers and actors interleave cooperatively on it (chapter 13 §13.8 *Concurrency, not parallelism*). The benefit is structural and for IO-bound loads, not multi-core speedup.

Cases where an actor is natural: a cache server, a connection controller, a process supervisor, a notification router, a task queue, a logger actor. Cases where a fiber suffices: an expensive computation the parent wants to run concurrently with other work, a `with_timeout` measuring how long something takes, a concurrent map over a list of IO.

Actors are not language primitives

One more thing to pin down before the syntax: in kaikai, actors aren't a core construct. They're a **layer built with algebraic effects**: the `Actor[Msg]` effect declares the operations (`self`, `send`, `receive`); the stdlib provides functions like `with_mailbox` and `spawn_actor` that install the `Actor[Msg]` handler on top of an ordinary fiber.

It's the same principle as `nursery` from chapter 13: the language core has only effects; the use patterns (fibers, actors, supervision) appear in libraries any reader can read. If after this chapter you wonder how `spawn_actor` works inside, the answer is a `spawn.spawn` plus a `handle ... with Actor[Msg]`.

14.1 `Actor[Msg]`: the effect

An actor is a fiber inside a `handle ... with Actor[Msg]` that grants it access to three operations. The effect is declared in the stdlib:

```
# Declared in stdlib/actor.kai, accessible via `import actor`.
pub effect Actor[Msg] {
  self()                    : Pid[Msg]
  send(pid: Pid[Msg], msg: Msg)  : Unit / Cancel
  receive()                 : Msg / Cancel
}
```

- **`self()`** returns the `Pid` of the current actor. The `Pid` is the handle others use to send it messages.
- **`send(pid, msg)`** enqueues `msg` in `pid`'s mailbox. If the mailbox is full, the behavior depends on the policy (§14.4).
- **`receive()`** takes the next message from the current actor's mailbox. If there's nothing, the fiber suspends until one arrives. Because it suspends, it's a yield point and carries `Cancel`.

`Msg` is the concrete message type the actor receives. **An actor handles a single message type.** If you need to mix shapes, unify them with a sum type:

```
type ServerMsg
  = Ping(Pid[Pong])
  | Stop
  | Tick
```

The actor knows it'll receive only one of those three constructors, and the exhaustive `match` on `receive()` guarantees you cover every case.

`Pid[Msg]`: typed handle

A `Pid[Msg]` is a mailbox identifier. It has a type, so the compiler doesn't let you send a `Notification` message to a `Pid[Task]`. This is the strongest difference from Erlang: in Erlang, PIDs are untyped; in kaikai, they're specific to the message type.

Like `Fiber[T]` from chapter 13, `Pid[Msg]` is **tied to the scope that created it**. You can't store it in a record that outlives the nursery, return it from a function outside the standard family, or pass it between data structures that aren't approved. The compiler rejects it. This closes the model: every PID has a known parent, and dies with it.

14.2 `with_mailbox`: give the current fiber a mailbox

The simplest way to start is to give a mailbox to the fiber you're already in. `with_mailbox` installs the `Actor[Msg]` handler and hands control to its body:

```
import actor

fn main() : Unit / Console {
```

```
  with_mailbox {
    Actor.send(Actor.self(), "hello")
    Actor.send(Actor.self(), "world")
    Stdout.print(Actor.receive())
    Stdout.print(Actor.receive())
  }
}
```

Output:

```
$ kai run examples/ch14/01_with_mailbox.kai
hello
world
```

`with_mailbox { ... }` is a call with trailing-lambda syntax: the block in braces is the body that runs with the mailbox installed. Because `with_mailbox` passes no arguments to the body (zero-parameter lambda), the block needs no arrow or binder. Inside, `Actor` is the capability: `Actor.self()` returns the newly created mailbox's `Pid`, `Actor.send(pid, msg)` enqueues, `Actor.receive()` takes the next one.

In this example the actor talks to itself. It's the "hello world" of the model; the real exercise is communicating two distinct actors.

14.3 `spawn_actor`: start a new actor

To start an actor that runs in its own fiber:

```
import actor

fn worker() : Unit / Actor[String] + Console {
  let t1 = Actor.receive()
  Stdout.print("working: " ++ t1)
  let t2 = Actor.receive()
  Stdout.print("working: " ++ t2)
  let t3 = Actor.receive()
  Stdout.print("working: " ++ t3)
}

fn main() : Unit / Console + Spawn + Cancel + Actor[String] {
  with_mailbox {
    let pid = spawn_actor(() => worker())
    Actor.send(pid, "task-1")
    Actor.send(pid, "task-2")
    Actor.send(pid, "task-3")
    spawn.yield()
    spawn.yield()
    spawn.yield()
    spawn.yield()
  }
}
```

`spawn_actor` starts a new fiber, installs a mailbox on it, and returns the `Pid` so the parent can send it messages. The worker's signature declares `Actor[String]`: it needs the effect to call `Actor.receive()`.

Notice that `main` also has `Actor[String]` in its row. Why? Because `Actor.send(pid, "task-1")` is an invocation of an operation of the `Actor[String]` effect, and like every effect operation it requires the effect to be available in context. The `with_mailbox` in `main` provides that capability. The `pid` already identifies the destination mailbox; the handler only dispatches the operation.

The `spawn.yield()`s at the end give the scheduler a chance to run the worker. Without them, `main` would exit before the worker processed anything.

14.4 Mailbox policies: what happens when it fills

By default, `with_mailbox` and `spawn_actor` create an **unbounded** mailbox: it never fills, messages pile up while they go unread. Reasonable to start, dangerous in practice: if a producer sends faster than a consumer processes, memory grows without bound.

For real cases, you pick a policy. The stdlib (module `actor`) exposes them as two sum types:

```
# Defined in stdlib/actor.kai, accessible via `import actor`.
pub type MailboxPolicy = Unbounded | Bounded(Int, Overflow)
pub type Overflow      = DropOldest | DropNewest | BlockSender
```

`Bounded(capacity, on_full)` gives a fixed-size mailbox. The `on_full` parameter decides what to do when a message arrives with no room:

- **`DropOldest`**: the oldest message is evicted, the new one enters. Useful for snapshots: only the latest state matters (telemetry, tick events, GPS).
- **`DropNewest`**: the new message is rejected, the mailbox stays as it was. Useful for "first wins" protocols (leader election, lock acquisition).
- **`BlockSender`**: the sender suspends until space frees up. Useful for **backpressure**: the producer slows down when the consumer can't keep up. It's a yield point, so a blocked sender can receive `Cancel.raise()`.

```
import actor

fn main() : Unit / Console {
  with_mailbox_policy(Bounded(2, DropOldest)) {
    Actor.send(Actor.self(), "a")
    Actor.send(Actor.self(), "b")
    Actor.send(Actor.self(), "c")
    Stdout.print(Actor.receive())
    Stdout.print(Actor.receive())
  }
}
```

Output:

```
$ kai run examples/ch14/04_mailbox_policy.kai
b
c
```

The mailbox has capacity 2. We send `a`, `b`, `c` without reading anything. When `c` arrives, there's no room, and `DropOldest` evicts `a`. The two reads recover `b` and `c`.

`DropOldest` and `DropNewest` **don't notify the sender** that their message was dropped. If you need to know, use `BlockSender` (or design the protocol with an ack). The silence is deliberate: the policy expresses a global preference of the mailbox, not a per-message negotiation.

14.5 Request/reply pattern

The most common pattern between actors is to ask one of them for something and wait for the answer. Each side of the dialogue has its own message type: the client sends a `Request` and receives a `Reply`; the server receives `Request` and sends `Reply`. The `Request` includes the client's `Pid[Reply]` so the server knows where to reply.

```
import actor

type Request = Query(String, Pid[Reply])
type Reply   = Answer(String)

fn server() : Unit / Actor[Request] + Actor[Reply] + Console {
  match Actor.receive() {
    Query(q, client) -> {
      Stdout.print("server: got '#{q}'")
      Actor.send(client, Answer("reply to '#{q}'"))
      server()
    }
  }
}

fn main() : Unit / Console + Spawn + Cancel + Actor[Reply] {
  with_mailbox {
    let s = spawn_actor(() => server())

    Actor.send(s, Query("two+two", Actor.self()))
    match Actor.receive() {
      Answer(r) -> Stdout.print("client: " ++ r)
    }
  }
}
```

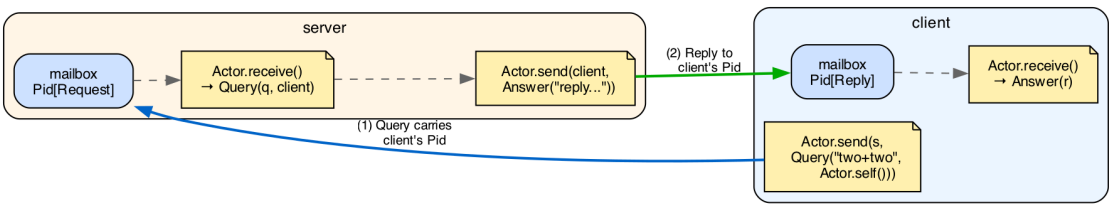

Figure 14.1 · *Request/reply between two actors. The client holds a `Pid[Reply]` mailbox; the server holds a `Pid[Request]` mailbox. Arrow (1) carries the `Query` — which includes the client's PID — into the server's mailbox; arrow (2) returns the `Answer` to the client's mailbox. Two typed mailboxes, two messages, one round trip.*

A few things worth noting about the structure:

- **Two distinct types, `Request` and `Reply`**, each with its own mailbox. The server declares `Actor[Request] + Actor[Reply]` in its row: it receives `Request` from its own mailbox and sends `Reply` to the client's mailbox. The client declares only `Actor[Reply]`: it has a `Reply` mailbox, not a `Request` one. The types tell you exactly which mailbox is which.
- **`Query` includes the return `Pid[Reply]`.** Without it, the server doesn't know whom to reply to. The `Pid`'s type guarantees that only `Reply` messages can be sent to it.
- **The server recurses** after processing the message. Without that recursion, the server would process a single message and exit. Tail recursion compiles to a loop (chapter 6), so the actor can run indefinitely without blowing the stack.

This replaces, in practical terms, synchronous API calls: you ask, wait, continue. The difference is that the "server" can be serving multiple clients at once, its internal state is encapsulated, and the type system guarantees that no client is waiting for an answer of the wrong type.

14.6 Supervision: links and monitors

In BEAM, actors are supervised with **links** (bidirectional) and **monitors** (unidirectional). When an actor crashes, the actors watching it find out and decide what to do.

kaikai ships the same model, expressed as two stdlib effects:

```
# Declared in stdlib/actor.kai, alongside Actor[Msg].
pub effect Link {
  link(pid: Pid[_]) : Unit
}

pub effect Monitor {
  monitor(pid: Pid[_]) : MonitorRef
  demonitor(ref: MonitorRef) : Unit
}
```

`Pid[_]` is an "existential" PID: any message type works. That's because `link` and `monitor` don't send or receive messages; they only register observation on the actor's life.

Links: bidirectional

`Link.link(pid)` declares that the current actor and `pid` are bound: if either fails, the other receives `Cancel.raise()`. It's the pattern for two actors that depend on each other symmetrically (a worker and its queue, two sides of a handshake). It is not what you want for "supervisor watches worker": that's the monitor case.

Monitors: unidirectional

`Monitor.monitor(pid)` declares that the current actor wants to know when `pid` ends, without coupling its own life to the observed actor's. When `pid` ends (normal return, crash, or cancellation), the observer receives a `MonitorDown` message in its mailbox. The stdlib exposes the relevant types:

```
# Defined in stdlib/actor.kai along with the Monitor effect.
pub type MonitorDown = MonitorDown(MonitorRef, TerminationCause)
pub type TerminationCause
  = Normal
  | Crashed(String)
  | Cancelled
```

For an actor to receive `MonitorDown`, its message type must include it as a variant:

```
type SupervisorMsg
  = Tick
  | Stop
  | Down(MonitorDown)
```

The supervisor does `Monitor.monitor(worker)` after creating it, and any worker termination arrives as `Down(ev)` to the supervisor's main `match`. The supervisor decides what to do (restart, escalate, ignore) without its own life being tied to the worker's.

When to pick each

- **Without `Link` or `Monitor`**, when the natural flow is for the actor itself to report how it went before exiting. It sends a `Done(...)` or `Failed(...)` message to its supervisor and exits cleanly. The supervisor sees it like any other message. That's the pattern the case study in §14.7 uses.
- **`Monitor`**, when the supervisor needs to react to terminations the actor doesn't control: crashes, cancellation from outside, panics. The supervisor stays alive and decides.
- **`Link`**, when two actors form a unit and it doesn't make sense for one to survive without the other.

The §14.7 pattern that follows uses explicit notification because it's the simplest. The variants that use `Monitor` or `Link` are refinements worth introducing only when the message protocol becomes insufficient.

14.7 Case study: supervisor with retries

We close with a complete program: a supervisor that launches a worker with a batch of tasks, watches whether the batch succeeded, and retries with an alternate batch if it failed.

```
import actor

#[derive(Show)]
type BatchResult
```

```
  = Done(Int)          # successful total
  | Failed(String)     # reason for the failure

fn process(tasks: [(Int, Int)], acc: Int) : BatchResult {
  match tasks {
    []               -> Done(acc)
    [(a, 0), ...]    -> Failed("division by zero at (#{a}, 0)")
    [(a, b), ...rest] -> process(rest, acc + (a / b))
  }
}

fn worker(supervisor: Pid[BatchResult], tasks: [(Int, Int)])
    : Unit / Actor[BatchResult] + Console {
  let r = process(tasks, 0)
  Stdout.print("worker: batch ended as #{r}")
  Actor.send(supervisor, r)
}

fn attempt(me: Pid[BatchResult], batch: [(Int, Int)])
    : BatchResult / Console + Spawn + Cancel + Actor[BatchResult] {
  let _ = spawn_actor(() => worker(me, batch))
  Actor.receive()
}

fn supervisor() : Unit / Console + Spawn + Cancel + Actor[BatchResult] {
  with_mailbox {
    let me = Actor.self()
    let first_batch  = [(10, 2), (20, 4), (30, 0)]
    let second_batch = [(10, 2), (20, 4), (30, 5)]
    match attempt(me, first_batch) {
      Done(total)    -> Stdout.print("supervisor: success on first try, total=#{total}")
      Failed(reason) -> {
        Stdout.print("supervisor: first try failed (#{reason}), retrying")
        match attempt(me, second_batch) {
          Done(total)    -> Stdout.print("supervisor: success on second try, total=#{total}")
          Failed(reason) -> Stdout.print("supervisor: second try failed too (#{reason}), giving up")
        }
      }
    }
  }
}

fn main() : Unit / Console + Spawn + Cancel {
  supervisor()
}
```

Output:

```
$ kai run examples/ch14/05_supervisor.kai
worker: batch ended as Failed(division by zero at (30, 0))
supervisor: first try failed (division by zero at (30, 0)), retrying
worker: batch ended as Done(16)
supervisor: success on second try, total=16
```

Three pieces worth commenting on:

- `process` **doesn't touch the actor system.** It's pure list logic: pattern match, recursion, returns a value. That means `process` is completely testable without starting fibers. The actor layer lives only in `worker`, `attempt`, and `supervisor`.
- `attempt` **separates a concern.** Before extracting it, the second-attempt code was inline inside the first `match`. Pulling that logic into its own function makes the "spawn + receive" pattern explicit.
- **The supervisor decides policy.** The worker is blind to the decision: it only reports. Changing the policy from "two tries with different data" to "three tries with exponential backoff" touches one function. That separation is the real benefit of the model.

What's missing for this to be production-ready? A few things:

- **Per-attempt timeout.** Today if the worker hangs, the supervisor hangs too. The solution is `with_timeout` (chapter 13 exercise) around `attempt`.
- **Cancel the worker if the supervisor gives up.** If we say "I give up" while the worker is still processing, we want the worker to end too. With `Link.link(worker)` after spawn, the supervisor's decision to return cancels the worker automatically. The explicit alternative is to add a `Stop` message to the worker's protocol and send it before giving up.
- **Persistent logging.** Instead of `Stdout.print`, send to a logger actor with its own bounded mailbox (`Bounded(1024, DropOldest)`).

Each is an afternoon's work. But the foundation is there: three actors, two message types, a type system that guarantees mailboxes are respected.

14.8 Philosophy: actors are a library

If you want to keep two things from this chapter, let it be these:

1. **Actors are not language primitives.** They're a library built on the `Actor[Msg]` effect, which is in turn an ordinary effect declaration. The stdlib provides `with_mailbox`, `spawn_actor`, the policies. The library's code is there for you to read. If you ever wonder "what does `spawn_actor` do underneath", the answer is a `handle` and a `spawn.spawn`.
2. **Every actor has a fixed message type.** The compiler guarantees you don't send the wrong message to the wrong mailbox. `Pid[Msg]` is typed, not a string. And so an actor system in kaikai is more statically verifiable than its equivalent in Erlang.

The most important consequence of the first idea: when you want a supervision pattern the stdlib doesn't provide, you can write it. The syntax hides nothing: `handle`, `receive`, `send`. Any actor pattern you've seen in Erlang, Akka, or any other actor framework should be expressible as an ordinary function in kaikai with these pieces.

Exercises

14.1. Modify the §14.3 example so the worker processes five tasks instead of three. What happens if you reduce the parent's `spawn.yield()`s? What's the minimum that lets all tasks get processed?

14.2. Take `BoundedDropOldest` from §14.4 and change it to `BoundedDropNewest`. What's the expected output? Justify by reasoning about which messages remain in the mailbox as each `send` arrives.

14.3. In §14.5's request/reply pattern, the client sends one question and leaves. If you wanted a client that asks five questions in series, what would you change? And if you wanted to fire them all at once and receive replies as they come in (concurrently, not in parallel: the scheduler interleaves them on the same thread)? Hint: you'd need to open several fibers in a nursery or group replies with a correlation id.

14.4. In the §14.7 case study, the supervisor retries twice. Generalise: write a function `with_retries[T, e](n: Int, attempt: () -> BatchResult / e) : BatchResult / e` that retries `n` times before giving up.

14.5. A "logger" actor receives `Info(String)` and prints them to stdout. Design its message type, its signature, and an appropriate `with_mailbox_policy`. Justify the choice of policy by considering: what happens if the producer sends faster than it can be printed?

Chapter 15 · Holes and kaikai with AI agents

This chapter is about a small tool with a big idea behind it. The tool is called a **hole**: a gap you leave in your code instead of an expression, with a `?` or a `?name`. The big idea is that the compiler can **talk to you** as you write: tell you what type is expected there, what names are in scope, what expressions could fit. The program keeps compiling with holes inside; it only aborts if execution reaches one.

Holes are useful even if you never use an LLM. They let you design top-down (write the signature first, fill in the body later) and make progress piece by piece without the whole file failing to compile. That's the human side.

But holes are also the piece around which kaikai is designed for **AI agents**. The compiler can emit its report as JSON, and an LLM reads that JSON to understand what's being asked of it. This is the language's strategic bet (`design.md` calls it Tier 3): a new language, without a large training corpus, can be writable by an agent if the tooling is designed well.

We'll take it in order. Humans first.

15.1 Typed holes: `?` and `?name`

A hole is a legal expression in kaikai. You write it as either `?` alone, or `?name` if you want to give it an identity:

```
fn circle_area(r: Real) : Real = ?formula
```

This compiles. The function `circle_area` exists, has the correct signature, and can be called from other parts of the program. What happens when execution reaches `?formula` is that the program aborts with a clear message:

```
$ kai run examples/ch15/01_basic_hole.kai
panic: unfilled hole: ?formula
```

It's not a compile-time error. It's a **deferred promise**: you left a gap that you'll fill later, and the system stays with you until you do.

The difference between `?` and `?name` is that the name helps identify the hole in messages and, more importantly, makes **two holes with the same name within the same function share a type**:

```
fn classify(n: Int) : String {
  if n < 0 {
    ?word
  } else {
    ?word
  }
}
```

The two `?word`s unify: if you decide one is `String`, the other is too. That reduces the temptation to write inconsistent implementations across branches of an `if` or a `match`.

Anonymous (`?` with no name) are each independent.

15.2 The conversation with the compiler

The point of holes isn't in aborting nicely; it's in what the compiler tells you about them. For each hole, it emits a **report**:

```
$ kai build examples/ch15/01_basic_hole.kai --holes
examples/ch15/01_basic_hole.kai:1:32: type hole

  expected: Real

  in scope:
    r : Real

  candidates that fit:
    r
    real_mul(r, r)

  replace `?formula` with one of the candidates or a literal Real.
```

Four pieces of information:

- **`expected`**: the type the hole's position demands. The compiler infers it from context: here, the function returns `Real`, the body is a single expression, therefore the hole must be `Real`.
- **`in scope`**: every name reachable from the hole's point, with its type. Here only `r : Real` (the parameter).
- **`candidates that fit`**: expressions the compiler can synthesize that have the expected type. For `Real` with `r` in scope: `r` itself, `real_mul(r, r)` which is also `Real`. Synthesis is **bounded**: at most one function application. It doesn't give you the full body; it gives you hints.
- **`replace`**: the final suggestion, in one line.

That's the conversation. While the signature is all you know, the compiler helps you see what could go inside.

Like any compiler report, the cost of invoking it is low: you run `kai build --holes` and read. You don't have to guess.

15.3 Top-down design: start from the signature

The most natural use of holes is **top-down design**. You start by writing the signature of what you want, without knowing how it'll be implemented. You put a `?` in the body. The program compiles. You move on to the next function.

```
fn tokenize(s: String) : [Token] = ?tokens

fn parse(ts: [Token]) : Expr = ?ast

fn evaluate(e: Expr) : Int {
  match e {
    Lit(n)    -> n
    Sum(a, b)  -> evaluate(a) + evaluate(b)
    Mul(a, b)  -> evaluate(a) * evaluate(b)
  }
}

fn compute(s: String) : Int = evaluate(parse(tokenize(s)))
```

Three signatures, one complete function (`evaluate`). The file compiles. You can run tests against `evaluate` with hand-built ASTs, before implementing `parse` or `tokenize`:

```
fn main() : Unit / Console {
  let ast = Sum(Lit(3), Mul(Lit(4), Lit(5)))
  println("3 + 4*5 = #{evaluate(ast)}")
}
```

Output:

```
$ kai run examples/ch15/05_top_down_design.kai
3 + 4*5 = 23
```

The program runs. Evaluation works. Now you'll fill the two holes one by one: implement `parse` (takes `[Token]`, returns `Expr`), then `tokenize` (takes `String`, returns `[Token]`). And when you fill the last one, `main` can call `compute("3 + 4*5")` directly.

Contrast this with two more common ways of working:

- **Bottom-up:** implement the smallest pieces first, combine them. It works, but sometimes you discover the pieces don't fit at the end.
- **Big bang:** write everything at once, it doesn't compile until the end. Works if you have a crystal-clear problem. If not, it's painful.

Top-down with holes is the middle ground: the structure exists from the start, the correctness of each piece is verified as you fill it.

15.4 Partial programs: keep moving with the rest compiling

A consequence of designing with holes is that **you can always run what you have**. If a function is complete, you can test it without waiting for the whole file:

```
fn double(x: Int) : Int = x * 2

fn average(a: Int, b: Int) : Int = ?formula

fn main() : Unit / Console {
  println("double(5) = #{double(5)}")
  # average isn't done, but the file still compiles.
}
```

`double` works and `kai run` prints its result. `average` waits to be implemented; as long as we don't call `average`, the program doesn't hit the hole and runs cleanly.

This reduces a lot of friction in keeping a program "almost compiling" as you develop. Other languages push you toward stubs with `unimplemented()`, `todo()`, or `return null`; in kaikai the `?` is a primitive of the idiom, and the compiler understands it has a type.

15.5 Holes in patterns: the incomplete match

A hole in pattern position works too:

```
type Shape
  = Circle(Real)
  | Square(Real)
  | Triangle(Real, Real)

fn area(s: Shape) : Real {
  match s {
    Circle(r)      -> 3.14 * r * r
    Square(l)      -> l * l
    Triangle(b, h) -> ?triangle_formula
  }
}
```

Here the `match` already covers all three constructors; what's missing is the expression in the last arm. The compiler verifies exhaustiveness (chapter 5 §5.4), tells you the `match` is complete, and reports the hole with `Real` as expected type and `b`, `h` in scope.

If you haven't yet decided whether you want `Triangle` in the type, you remove it and the compiler tells you `match` is no longer exhaustive. Holes are orthogonal to pattern checking; each one does its job.

15.6 The LLM bet: a language designed for agents

Up to here holes are a tool for humans. The most strategic part of chapter 15 is that **holes are also the entry point for an AI agent to write kaikai**.

The reasoning is simple. An LLM learns a language from the **corpus** in its training data. Languages with a lot of corpus (Python, JavaScript) are easy for models to generate; new languages with little public code are hard. If kaikai waited until it had a large corpus, it would miss the experiment.

But there's an alternative: design the language so that the compiler is what teaches the agent, not the corpus. If when the LLM produces incorrect code, the compiler can say precisely what's missing and where, the LLM can iterate to the correct program in a few steps.

The key piece is **structured output**: the compiler emits JSON that the agent reads directly, without heuristic parsing. Three relevant channels:

1. `kai build --holes-json`: the hole report in JSON.
2. `kai type --json`: the type of any expression.
3. **Compiler diagnostics in JSON**: type errors, non-exhaustive matches, unhandled effects.

Chapter 5 §5.4 showed how a non-exhaustive match message looks in human form. The JSON version has the same fields as data: scrutinee type, list of missing variants, list of covered variants, suggestion. An agent can parse it and produce the code that covers the missing variant without needing to read natural language.

15.7 The JSON output of holes

The JSON report has a stable schema:

```
[
  {
    "file": "area.kai",
    "line": 1, "col": 32,
    "name": "formula",
    "expected_type": "Real",
    "in_scope": [
      {"name": "r", "type": "Real"}
    ],
    "candidates": [
      {"expr": "r", "kind": "local"},
      {"expr": "real_mul(r, r)", "kind": "application"}
    ]
  }
]
```

Each hole is an object. The array has as many elements as there are holes in the file. The fields are the same as the human report in §15.2, but as structured data.

For a human, this is noisy. For an agent, it's exact. And exact matters: the difference between "the agent got it on the third try" and "the agent got it on the first" is the difference between a tool that's practical and one that feels magical.

15.8 Beyond holes: rich information as interface

Holes are the first piece of a general pattern. Other compiler tools follow the same principle: emit structured information that an agent can consume.

- **`kai type <expr>`** returns the type of an expression in the context of a file. With `--json`, the result is an object with fields `type`, `effects` (the row), and `notes` (references to the definition).
- **`kai check`** runs the file's properties and tests and reports results. With `--json`, each test/check/bench is an object with `name`, `status`, `duration_ms`, and a `counterexample` field for `check`s that failed (with the exact value that broke the property).
- **Compiler diagnostics** (errors and warnings) have a `--json` form: error type, location, span, human message, structured message, list of suggestions.

The common rule: the agent never has to parse text. The information arrives already structured.

15.9 A workflow with an agent

When a programmer works with an AI agent on kaikai, the reasonable flow is this:

1. **The human writes the signature and the tests.** Defines what the program is expected to do. Puts holes in the bodies.
2. **The agent reads `--holes-json`.** It knows what type is expected, what bindings are in scope, what candidates are reasonable. It generates an implementation proposal.
3. **The human runs `kai check`.** If the tests pass, things proceed. If not, the counterexample tells the agent what's wrong.
4. **The agent iterates.** Reads the counterexample, adjusts, proposes another implementation.

Three things worth noting about this flow:

- **The human decides the what.** Signature, tests, properties are what say what the program should do.
- **The agent decides the how.** Function bodies, intermediate structures, implementation details.
- **The compiler mediates.** It's the referee: it says whether the proposal meets the types, the tests, the properties. The agent never accepts anything without the compiler having said it passes.

It's the opposite of the "LLM writes the code and the human reviews" pattern. Here the human writes **the specification** (signature + tests), the agent writes **the code**, the compiler **verifies** the code satisfies the specification.

15.10 What the language doesn't automate

Holes are a tool for talking to the compiler. They're not a tool for talking to the programmer's judgment. There are things the compiler can't tell you, that no `--holes-json` will deliver:

- **Which function your program needs.** If you decide that `circle_area` should exist, that's your decision. The compiler won't invent the signature for you.
- **What to name things.** If you call your function `process_data` instead of `transform_records`, no hole will correct you. Taste and readability are yours.
- **Whether the architecture makes sense.** That the compiler accepts a piece doesn't mean the piece belongs where you put it. Deciding what belongs in each module, what effects each function should expose, when to extract an abstraction: that's design, and design is human.

The author has a recurring idea about this on the blog: tools don't replace judgment, they amplify it when they're well designed. The compiler with holes and agents reads a mechanical part of the work (the details that satisfy the type constraints). The other part (what to build, what to abstract, what to prioritize) remains the programmer's.

15.11 Case study: completing a non-trivial function

We close with a realistic exercise. Imagine you want to write a function that takes a list of pairs `[(String, Int)]` representing student grades, and returns the names of those who passed (grade >= 4) in alphabetical order.

The human writes the signature and two tests:

```
fn passed(grades: [(String, Int)]) : [String] = ?body

test "empty list" {
  assert passed([]) == []
}

test "filter and sort" {
  let r = passed([("Carmen", 5), ("Ana", 3), ("Berta", 6)])
  assert r == ["Berta", "Carmen"]
}
```

The human runs `kai build --holes-json`. The agent receives:

```
{
  "name": "body",
  "expected_type": "[String]",
  "in_scope": [
    {"name": "grades", "type": "[(String, Int)]"}
  ],
  "candidates": [
    {"expr": "[]", "kind": "literal"}
  ]
}
```

The agent knows: expected type `[String]`, an input `grades` of type `[(String, Int)]`. Candidates are thin because the compiler's synthesis is bounded; the agent has to propose something more substantial. A first proposal:

```
fn passed(grades: [(String, Int)]) : [String] =
  grades
    |? . .1 >= 4
    | . .0
    |> list.sort
```

The agent runs `kai check`. Both tests pass: the filter keeps Carmen and Berta (grades 5 and 6) and drops Ana (grade 3), then `list.sort` over `[String]` puts them in ascending lexicographic order. If both pass, the agent is done.

The human never touched the body. The compiler validated types and tests. The agent iterated if needed. Each one did what it does best.

Is it always this clean? No. There are functions where the agent fails three times before getting it. There are functions where the test counterexample is ambiguous and the agent doesn't know what to adjust. But each iteration is cheap (seconds), and the cost of being wrong is transparent (the compiler or the test reports exactly what's wrong).

15.12 Philosophy: three ideas worth remembering

1. **Holes are a primitive for dialogue.** They're not `null` or `unimplemented()`: they're a legal form of expression that compiles, that the compiler understands, and for which it emits structured information. The human uses them for top-down design; the agent uses them as entry point.
2. **The compiler is the language's interface.** What the compiler says (expected types, errors, counterexamples, candidates) is what defines what you can do in kaikai. Designing that output well (both human and JSON) is what makes the language accessible to humans and agents alike.
3. **The human says the what; the agent says the how; the compiler verifies.** That's the division kaikai proposes for AI-assisted work. Each part does what it does best. None replaces the others.

Exercises

15.1. Take a simple function you know (say `fn sum_evens(xs: [Int]) : Int`). Write the signature with a `?body` and the tests you'd expect. Without looking at the implementation, write three body proposals by hand and run them. How many tries until you find the right one?

15.2. Write a function with two branches of an `if`, where each branch is a `?name` with the same name. Verify that the compiler unifies the types of the two. Then change one of the names and observe: now each branch has its own type. In which cases is each form better for you?

15.3. Read `kai build --holes-json` on a file with three holes at different positions. What information do all the holes share? What information is specific to each one?

15.4. (Requires access to an AI agent.) Take a function from chapter 5 (say the expression evaluator from §5.7) and delete the body of one of the `match` branches, replacing it with

a `?name`. Ask the agent to complete it using only the compiler's JSON output as input. How fast does it resolve?

15.5. Discuss with a colleague: what parts of the work you do today programming could an agent do if you gave it enough structured information from the compiler? What parts definitely couldn't? Why?

Chapter 16 · Tooling: the `kai` binary

Every chapter so far has focused on the language: syntax, types, effects, the memory model. But a language without tooling doesn't get used. This chapter covers the other side: the `kai` binary, which is the face every programmer interacts with every day.

It's a short reference chapter. No exercises. The point is for you to know which command to use when, and to have the list at hand to come back to.

16.1 Compiling and running: `kai run`, `kai build`

The command you'll live with is `kai run`:

```
$ kai run hello.kai
hello, kaikai
```

`kai run` compiles the file to a native binary, runs it, and forwards any extra arguments to the program. It's the edit-save-run cycle of the day-to-day. Underneath there's a compiler (`kaic2`) that produces C, then invokes `cc` to compile to an executable, and finally runs the executable.

If you want the binary without running it, use `kai build`:

```
$ kai build hello.kai
$ ./hello
hello, kaikai

$ kai build hello.kai -o build/hello
$ ./build/hello
hello, kaikai
```

`kai build` doesn't run the program: it leaves the executable on disk. With `-o` you specify where. The binary is **essentially static**: it doesn't depend on the kaikai compiler, only on the system's libc. You can copy it to another machine with the same OS and architecture and it will run.

Fast compilation

`kai run` and `kai build` are designed to feel immediate. A program of a few hundred lines compiles in less than a second on a reasonable machine. That speed isn't an accident: the compiler is self-hosted (kaikai compiled in kaikai), avoids costly passes like global type inference where it doesn't need to, and emits C directly instead of going through LLVM. For larger programs there's a cache (chapter 8 §8.8 covers the package cache; the per-file compilation cache is another story).

To put it in perspective: a Rust program of comparable size can take 30 seconds to compile. A kaikai program of the same size takes less than a second. The difference shows.

16.2 Tests, properties and benchmarks

Three subcommands cover the three verification constructs from chapter 7:

- **`kai test`** runs `test "..." { ... }` blocks.
- **`kai check`** runs `check "..." with x: T { ... }` blocks (properties verified with randomly generated values).
- **`kai bench`** runs `bench "..." { ... }` blocks and reports timings.

```
$ kai test calculator.kai
  ok   sum of zero
  ok   unit product
  ok   literal evaluation

3/3 tests passed
```

```
$ kai check calculator.kai
  commutativity of addition: 100 iter, OK
  associativity of multiplication: 100 iter, OK

2/2 checks passed
```

```
$ kai bench calculator.kai
  small-tree evaluation: 1000 iter / median 12 ns / MAD 1 ns / mean 13 ns / range [10, 45]
  large-tree evaluation: 1000 iter / median 8.4 us / MAD 0.2 us / mean 8.5 us / range [8.0, 12.1]

2 benches
```

`kai bench` takes `--iters N` to set the iteration count (default 1000). For costly benchmarks, lower it; for more stable measurements, raise it.

The three commands share two important properties:

- **They only run the relevant blocks.** `kai test` ignores `check` and `bench`; `kai check` ignores `test` and `bench`.
- **The blocks don't land in the production binary.** `kai run` and `kai build` drop them entirely.

16.3 Formatting: kai fmt

kai fmt is the canonical formatter. gofmt style:

- One single correct way to print any file.
- No configuration options. The project doesn't want style wars.
- Idempotent: formatting an already-formatted file doesn't change it.

Three ways to use it:

```
$ kai fmt file.kai              # rewrite in place
$ kai fmt --check file.kai      # exit 0 if formatted, 1 if not
$ cat file.kai | kai fmt --stdin  # read stdin, write stdout
```

The --check form is for CI: if the code isn't formatted, the job fails and forces you to run kai fmt before merging.

The --stdin form is for editors: your editor pipes the buffer to the formatter before saving, gets the canonical result back, and writes it.

16.4 Package management: init, add, install, update

Chapter 8 §8.5-8.8 covered the package model (kai.toml manifest, kai.lock lockfile, shared cache, minimum-version selection). Here we list the subcommands that orchestrate the model:

```
$ kai init myapp
kai-pkg: wrote kai.toml for package 'myapp'

$ kai add github.com/kaikailang-org/manutara@v0.1.0
$ kai install
$ kai update              # refresh all deps
$ kai update manutara     # refresh only manutara
$ kai show                # print parsed kai.toml
```

kai run and kai build invoke kai install automatically if they detect dependencies declared in kai.toml but not resolved in kai.lock. In practice, after cloning a kaikai project, kai run is enough to download whatever's missing.

16.5 Development mode: kai watch

kai watch is useful when you're iterating intensely on a program:

```
$ kai watch main.kai
[watching main.kai...]
```

Every time you save the file, the watcher detects the change, recompiles, and runs. It lets you keep the result visible without going back to the terminal to type kai run. It's the fastest way to explore a change in a demo or a script.

16.6 Editor integration: `kai lsp`

`kai lsp` is the Language Server that kaikai exposes for editors. It implements the standard Language Server Protocol, so any editor with LSP support (VS Code, Neovim, Emacs, IntelliJ with plugin) can connect and get:

- Type-on-hover: hover over an expression and see its type.
- Goto-definition: jump to where a name was declared.
- Live diagnostics: errors and warnings from the compiler appear in the buffer as you type.

The exact editor configuration varies. For VS Code, there's an official extension that starts `kai lsp` automatically. For Neovim, configure `nvim-lspconfig` with `kai lsp` as the command.

The LSP is the piece that makes development in kaikai comparable, in everyday ergonomics, to Rust or TypeScript: feedback is instantaneous, no need to go to the terminal to discover an error.

16.7 Environment variables

A handful of environment variables control the `kai` binary's behavior for special cases:

- **`CC`** (default: `cc`): the C compiler `kai` invokes to produce the final executable. If you have multiple C versions on the system, or want to use `clang` specifically, you set it here: `CC=clang kai run file.kai`.
- **`CFLAGS`** (default: empty): extra flags for the C compiler. Useful for optimization (`CFLAGS=-O3`) or warnings (`CFLAGS=-Wall`).
- **`KAI_NO_STDLIB=1`**: skips automatic stdlib loading. For advanced cases: compiler bootstrap, embedded targets without full libc, experiments.
- **`KAI_STDLIB`**: override the stdlib root. By default, `kai` auto-detects where it lives (installed vs development checkout). If you want to use an alternate version, you point it here.
- **`KAI_INCLUDE`**: override the runtime headers (`runtime.h`) root. Same principle as `KAI_STDLIB`.

For normal use you don't need to touch any of this. The binary comes preconfigured to find its own things.

16.8 Typical project structure

A standard kaikai project looks like this:

```
myapp/
├── kai.toml          # package manifest
├── kai.lock          # lockfile (commit with the code)
├── main.kai          # entry point
├── lib/              # public modules (if it's a library)
│   ├── core.kai
│   └── parser.kai
├── tests/            # heavy tests that don't fit inline
```

```
│   └── integration.kai
└── examples/            # demos that use the library
    └── basic/
        ├── kai.toml     # with `mylib = { path = ".." }`
        └── main.kai
```

Conventions:

- **main.kai** at the root if the project produces an executable. The signature must be fn main() : ... = ...
- **lib/** for the importable code of a library project. When someone installs your package with kai add, what they see via import is what lives under lib/.
- **tests/** for tests you prefer to keep separate (for example, because they're slow or use IO). Inline tests in the source file remain the primary pattern.
- **examples/<name>/** for demos. Each demo has its own kai.toml declaring a local dependency on the main package. That lets you exercise the library as an external consumer would.

None of this is required. kai run file.kai runs any .kai file regardless of where it lives. But when the project grows, this structure pays off.

16.9 Talking to C: extern "C" and the Ffi effect

Sooner or later you need a library that already exists in C: a database driver, a graphics framework, a numeric package. kaikai's foreign function interface (**FFI**) is how you call into it from kaikai code without giving up the type system or the effect row.

Declaring an external function

The simplest case is binding a libc function directly:

```
extern "C" fn llabs(n: Int) : Int / Ffi

fn main() : Unit / Console + Ffi {
  print("|-7| = #{llabs(0 - 7)}")
}
```

```
$ kai run abs.kai
|-7| = 7
```

Reading line by line:

- **extern "C" fn name(args) : T / Ffi** declares an external symbol. The compiler emits a forward declaration for the C linker to resolve. The body is implicit: the call site lowers to a direct C function call.
- **/ Ffi** is the effect. Any function that calls an extern "C" declaration — directly or transitively — has Ffi in its row. Same discipline as Stdout or File: a function that talks to C says so in its signature.

The compiler maps kaikai's primitive types to their C equivalents at the boundary:

kaikai	C
`Int`	`int64_t`
`Real`	`double`
`Bool`	`int8_t` (0 / 1)
`Char`	`int32_t` (codepoint)
`String`	`const char *` (NUL-terminated, kaikai-owned)
`Unit`	`void` (return only)

Anything more structured — records, lists, sum types — is **not** crossable directly in FFI v1. We come back to that in a moment.

Renaming the C symbol

Sometimes the C symbol name clashes with a kaikai identifier or just reads badly inline. Use the optional parenthesised override:

```
extern "C"("strlen") fn c_strlen(s: String) : Int / Ffi
```

The kaikai-side name is `c_strlen`; the linker resolves against `strlen`. Useful when the C name is a kaikai keyword, when you want a kaikai-flavoured name on top of a generic C one, or when you need two kaikai bindings that target the same C symbol with different signatures.

Linking against your own C code

For libraries that aren't already in libc, the typical shape is: write a small C file with the functions you need, let `kai build` invoke its C compiler with that file included. The package manager doesn't automate C compilation in v1, so you wire it via the `CFLAGS` environment variable that `kai` passes through to the host C compiler.

A minimal example. The C side:

```
// shim.h
#include <stdint.h>
int64_t my_double(int64_t x);
```

```
// shim.c
#include "shim.h"
int64_t my_double(int64_t x) { return x * 2; }
```

The kaikai side:

```
extern "C" fn my_double(x: Int) : Int / Ffi

fn main() : Unit / Console + Ffi {
  print("double(21) = #{my_double(21)}")
}
```

Building:

```
$ CFLAGS="-include shim.h shim.c" kai build --backend=c app.kai -o app
$ ./app
double(21) = 42
```

The `CFLAGS` value lets you splice anything the host C compiler accepts: `-include` to expose declarations, extra `.c` sources to compile in, `-l<lib>` to link against installed libraries, `pkg-config --cflags --libs <package>` to pull in the flags of a system library. Wrap the whole thing in a `Makefile` when it grows beyond one line.

The `--backend=c` flag here is required because the LLVM backend (chapter 16 §16.1) doesn't expose the same `CFLAGS` plumbing in v1.

What FFI v1 doesn't do

The list is short but important:

- **Records / structs by value across the boundary.** You can't declare `extern "C" fn draw(c: Color)` where `Color` is a kaikai record matching a C `struct`. v1 passes primitives only.
- **Out-parameters and pointer arguments.** No `int *out` style: every value crosses by value.
- **Variadic C functions.** No direct binding to `printf`'s family; you wrap them in a fixed-arity C helper.
- **C callbacks back into kaikai.** A C function that takes a function pointer can't call back into a kaikai function. Post-FFI-v2.

The canonical workaround for the struct case is a **C shim**: a thin C function that flattens the struct into primitives at the kaikai boundary and reconstructs it before calling the real library. The cost is one C function per struct-using entry point in the library you're binding. Worth it for v1; FFI v2 will remove the shim layer for the common case.

When to reach for FFI

The honest rule: **only when you genuinely need the C library**. Each `extern "C"` is a hole in the kaikai-side guarantees. The compiler can't check what the C function does with its arguments, can't prove its effects, can't reason about its memory model. The `Ffi` effect at least makes the hole visible in the signature, but the audit weight of that signature is "trust the C library author" plus "trust the C compiler".

For pure computation, prefer a kaikai implementation. For IO and OS facilities, prefer the stdlib's effects (`Stdout`, `File`, `NetTcp`, etc.) — those are already wired to C internally but in a way the language designers control. FFI is the right tool for binding existing C ecosystems you don't want to rewrite: drivers, native UI toolkits, hardware-specific libraries.

A small heuristic: if you find yourself writing more than ten `extern "C"` declarations to wrap something, and the library has a stable C API, that's a candidate for a proper kaikai package once `kai bindgen` lands. Until then, the manual approach (one `extern "C"` per function, one C shim per struct-using entry point) works fine.

16.10 Editions: stability without stagnation

There's one decision the rest of the book takes for granted without quite explaining it: kaikai uses **editions** to separate *what we promise won't change* from *what we reserve the right to move*. The idea isn't new — Rust formalized it in 2014 — but kaikai takes it seriously from the start.

What an edition is

An edition is a name — `tongariki`, `hanga-roa`, `orongo` — that pins a version of the **language contract** between kaikai and your code. Within one edition, the following don't change in incompatible ways:

- syntax and reserved keywords;
- type and effect system semantics;
- `pub` signatures in stdlib;
- the `kai` CLI flags and behaviour;
- the `kai.toml` schema.

Outside the contract — and therefore free to change between releases — is everything that doesn't touch your source: internal variant layout, fiber stack format, on-disk cache format, exact diagnostic wording, typer passes, Perceus internals, performance characteristics.

The commitment to you is simple: **upgrading the compiler is painless**. Read the release notes, install the new version, recompile. The commitment to the kaikai team is also simple: we can iterate hard on internals as long as we don't break what's outside. Both sides win.

How you declare it

In your `kai.toml`:

```
name = "myapp"
version = "0.1.0"
edition = "hanga-roa"

[dependencies]
```

And to check the active edition of your installation:

```
$ kai --version
kaikai 0.76.0 - hanga-roa (stage 2, self-hosted)
```

If `kai.toml` omits the field, the compiler assumes the installation's default edition. Recommendation: as soon as a package is going to live past a weekend, pin it explicitly. It's the difference between "compiles today" and "will keep compiling."

Multi-edition: old code, new compiler

The kaikai compiler accepts **any edition it knows about**. If your package declares `edition = "tongariki"` and the installation is on `hanga-roa`, the compiler applies the tongariki rules

to that package — even when another package on the same machine builds against hanga-roa. That's the mechanic behind "stability without stagnation": you don't have to migrate your whole world at once.

When an edition is sunset (after the ecosystem has migrated), later kaikai releases may drop support for it. Until then, old and new coexist.

The escape hatch: #[unstable]

Sometimes a module wants to expose a new API *for real* without yet committing to the exact signature. The #[unstable] annotation marks declarations as outside the edition contract:

```
#[unstable]
pub fn from_stdin() : Source[String, Stdin + Spawn] / Spawn =
  ?from_stdin

#[unstable]
pub type Source[t, e] = { pid: Pid[Demand] }
```

Consuming an #[unstable] API has to be declared in **your own** kai.toml:

```
[unstable]
ahu = true
```

The idea: nobody uses an API in flux without knowing. The edition contract still covers everything else.

Existing editions

At the time this book ships, kaikai knows three:

Edition	Status	Notes
tongariki	closed	Fast-iteration phase before 2026. Only packages that haven't migrated.
hanga-roa	active (default)	The first public edition. This book is written against it.
orongo	future	Next edition. Items deferred from Hanga Roa land here.

The names follow the Rapa Nui geography used across the ecosystem: places on Rapa Nui in chronological order. When hanga-roa is sunset, you'll get an announcement, a migration guide, and kai migrate to automate the mechanical changes. Until then, what you wrote against hanga-roa keeps compiling.

16.11 Philosophy: three principles of the tooling

If you want to keep the overall feel of the tooling, three ideas:

1. **Speed first.** Compiling and running must feel immediate. If the edit-save-run cycle is slow, the programmer writes less code, tests less, and builds less confidence. All of kaikai's tooling is designed against that clock.

2. **One right way for each thing.** A canonical formatter with no options. A package manager with MVS and no complex resolution. A test system integrated into the language. The philosophy is the same as Go's: minimize the decisions the programmer has to make about how to use the tools, to free time for deciding what to build.
3. **What matters is what comes out of the compiler.** Precise diagnostics, exact counterexamples, structured formats (JSON for holes and types). The compiler is the real interface of the language. Making it clear and fast is what makes kaikai usable, before any sophisticated IDE.

These aren't empty words. Every time the language grows a feature, the question "how is this going to feel in the tooling?" gets asked before "is it theoretically elegant?". Sometimes elegance wins (algebraic effects do pay some tooling complexity); sometimes tooling wins (exhaustiveness rules and local inference get tuned to make the messages good). The balance is live, and this chapter is its visible face.

Chapter 17 · Case study: HTTP server

We've reached the first of two case studies that close the book. The point is to see, in one place, how the pieces from the previous chapters fit into something resembling real software.

This chapter covers **an HTTP server**: the family of problems where what matters is concurrency, modularity, and the separation between domain logic and IO. Chapter 18 will cover the other end of the industry spectrum: **a ledger**, where what matters is precise types (currencies with units), business invariants (`requires` / `ensures` contracts), and strict immutability. Two cases, same language, different emphases.

The program: a notes server with a simple HTTP interface (`GET /notes`, `POST /notes`, `GET /notes/<id>`, `DELETE /notes/<id>`), keeps notes in memory, and writes each change to a log file. The "real" part isn't the logic (it's simple), it's how it's wired: effects in the signatures, actors to encapsulate state, fibers for concurrent connections, modules to separate domain, HTTP parser, storage, persistence.

Program size: about 250 lines, across five files.

17.1 The shape of the program

Before the code, the pieces and their responsibilities:

```
notes/
├── kai.toml          # project manifest
├── main.kai          # entry point and accept loop
├── domain.kai        # types: Note, Command, Response
├── store.kai         # actor that holds the notes
├── persistence.kai   # actor that writes the log
└── web.kai           # minimal HTTP parser and serializer
```

Five files, five distinct concerns:

- **`domain.kai`** is the center. Pure types, no effects, no IO. What the domain "is": what a note is, what commands can be issued, what responses can be produced.

- **store.kai** is an actor. Receives commands, holds the list of notes as internal state, replies to whoever asked. The manipulation logic inside is pure (function `process`), wrapped in an `Actor.receive()` loop that connects it to the world.
- **persistence.kai** is another actor. Receives events (creation, deletion), writes them to a log file. Isolating the disk in an actor lets the store keep responding even if the write is slow.
- **web.kai** is pure functions: parsing HTTP bytes into a `HttpReq` structure, translating requests into domain commands, serialising responses to bytes. No actors, no IO.
- **main.kai** wires it all: starts the actors, opens the TCP socket, opens a nursery, and spawns a fiber for each new connection.

This separation is the natural shape in kaikai. Each module is orthogonal: the pure domain logic can be tested without starting fibers, the HTTP parser without opening sockets, the store without touching disk. `main` just connects the pieces.

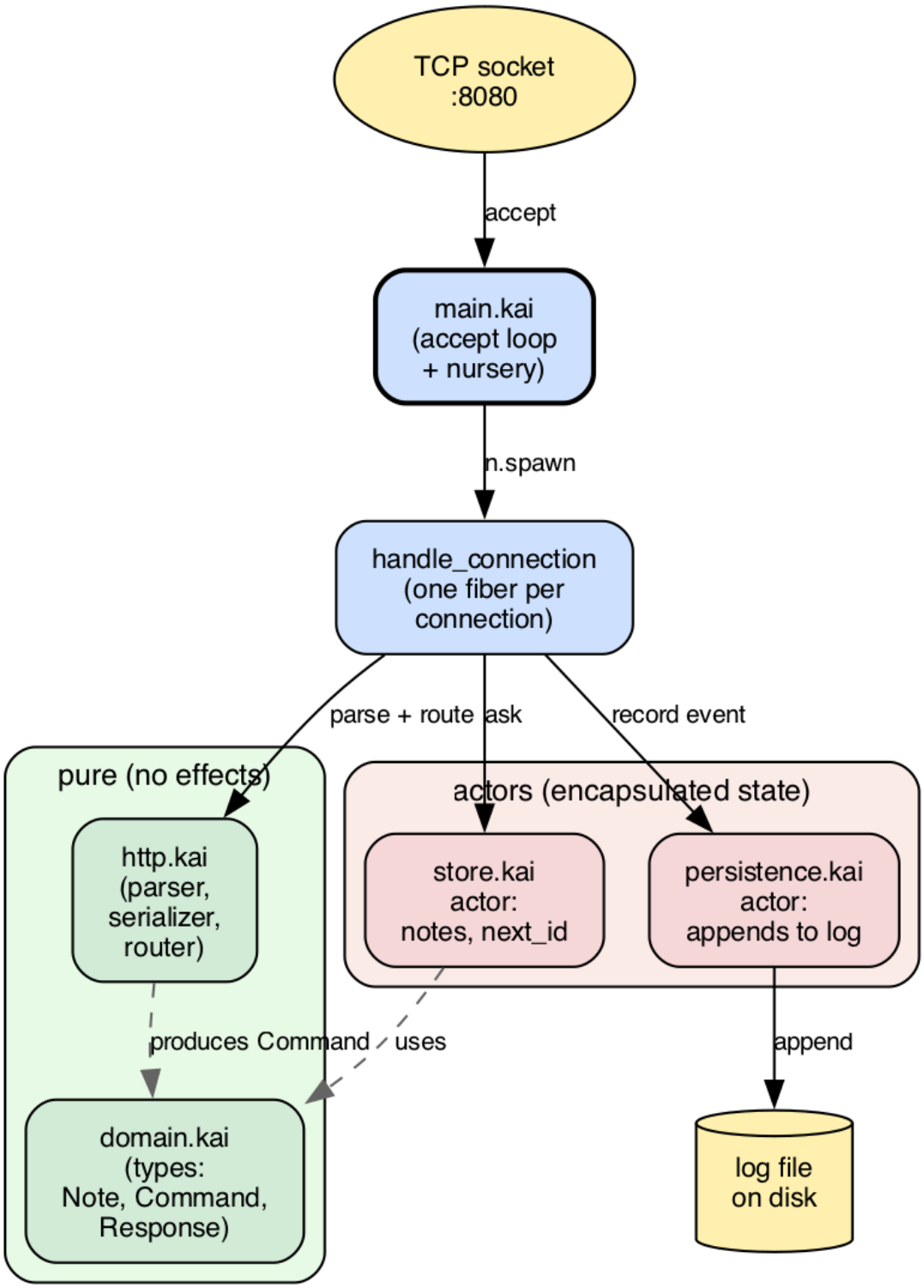

Figure 17.1 · *Notes server architecture. Five modules, two actors (store and persistence), one fiber per incoming connection. The pure modules (`domain.kai`, `web.kai`, green cluster) have no effects; the stateful modules (`store.kai`, `persistence.kai`, red cluster) hide their mutation behind a mailbox; `main.kai` is glue.*

17.2 The domain: pure types

We start in the center. `domain.kai`:

```
#[derive(Show)]
pub type Note = { id: Int, body: String }

pub type Command
  = List
  | Get(Int)
  | Create(String)
  | Delete(Int)

pub type Response
  = Ok(String)
  | Created(Note)
  | NotFound
  | ClientError(String)
  | ServerError(String)
```

Three declarations. A note has an id and a body. Four commands can be issued against the domain (list, get one, create, delete). Five possible responses, which map conceptually to HTTP 200, 201, 404, 400, and 500.

What's **not** here: no HTTP, no fibers, no files. If one day you decide to expose the API over gRPC instead of HTTP, this file doesn't change. If you decide to switch storage from memory to SQLite, this file doesn't change. It's the program's invariant.

The `#[derive(Show)]` on `Note` is what lets us interpolate `#{note}` in a string (chapter 9). Without it, we'd have to write `impl Show for Note` by hand.

17.3 The store: actor with state

`store.kai` defines an actor that keeps the list of notes and answers commands. Its message type is the command plus the `Pid` to reply to:

```
import actor
import domain

pub type StoreMsg = Ask(domain.Command, Pid[StoreResp])
pub type StoreResp = Reply(domain.Response)
```

`StoreMsg` is what the store receives; `StoreResp` is what it returns. The client, before sending, opens its own mailbox with `with_mailbox`, puts the `Pid` in the message, and then waits for the reply. It's the request/reply pattern from §14.5.

The heart of the module is the **pure logic** of processing:

```
pub fn process(c: domain.Command, notes: [domain.Note], next_id: Int)
    : (domain.Response, [domain.Note], Int) {
  match c {
    List -> {
```

```
      let bodies = list.map(notes, (n) => n.body)
      (domain.Ok(serialize_list(bodies)), notes, next_id)
    }
    Get(id) ->
      match find(notes, id) {
        Some(n) -> (domain.Ok(n.body), notes, next_id)
        None    -> (domain.NotFound, notes, next_id)
      }
    Create(body) -> {
      let n = domain.Note { id: next_id, body: body }
      (domain.Created(n), [n, ...notes], next_id + 1)
    }
    Delete(id) ->
      match find(notes, id) {
        Some(_) -> {
          let remaining = list.filter(notes, (n) => n.id != id)
          (domain.Ok("deleted"), remaining, next_id)
        }
        None -> (domain.NotFound, notes, next_id)
      }
  }
}
```

A single function, no effects in its signature. Takes the command, the current notes, and the next id; returns the response, the new note list, and the new next id. Exhaustive pattern match over the four variants of `Command`. Lists built with `[h, ...tail]`. `list.map` and `list.filter`. None of this is new in chapter 17: it's the chapter-5 constructs (sum types and match) and chapter-6 (functions and pipelines) put to work.

Because `process` is pure, it's **directly testable**:

```
test "create and get" {
  let (r1, n1, id1) = process(domain.Create("first"), [], 1)
  let created_ok = match r1 {
    domain.Created(_) -> true
    _                 -> false
  }
  assert created_ok
  assert id1 == 2

  let (r2, _, _) = process(domain.Get(1), n1, id1)
  let get_ok = match r2 {
    domain.Ok(c) -> c == "first"
    _            -> false
  }
  assert get_ok
}
```

No fibers, no IO, no sockets. Just logic. If tomorrow we want to parallelize note creation, add indexes, change the search algorithm, all those changes go through this pure function and get tested here.

On top of `process` sits the **actor loop** that connects it to `Actor.receive()`:

```
fn loop(notes: [domain.Note], next_id: Int)
    : Unit / Actor[StoreMsg] + Actor[StoreResp] {
  match Actor.receive() {
    Ask(command, client) -> {
      let (resp, new_notes, new_id) = process(command, notes, next_id)
      Actor.send(client, Reply(resp))
      loop(new_notes, new_id)
    }
  }
}
```

Three lines of work:

1. Receives an ask.
2. Processes it (pure logic).
3. Replies and recurses with the new state.

Tail recursion compiles to a loop (chapter 6), so the actor can run indefinitely. And the signature declares the two effects the actor produces: `Actor[StoreMsg]` to receive, `Actor[StoreResp]` to reply.

The `start` helper wires it together:

```
pub fn start() : Pid[StoreMsg] / Spawn + Cancel + Actor[StoreMsg] + Actor[StoreResp] {
  spawn_actor(() => loop([], 1))
}
```

And a synchronous wrapper for clients:

```
pub fn ask(store: Pid[StoreMsg], c: domain.Command)
    : domain.Response / Actor[StoreMsg] + Actor[StoreResp] + Cancel {
  Actor.send(store, Ask(c, Actor.self()))
  match Actor.receive() {
    Reply(r) -> r
  }
}
```

`ask` is what the HTTP handlers in `main` call: "send this question to the store and give me the answer". Underneath it's a send followed by a receive. We expose it as a simple function, not an open protocol.

17.4 Persistence: writing actor

`persistence.kai` is simpler. An actor that receives log lines and appends them to a file:

```
import actor
import fs.file

pub type Event = Line(String)

fn loop(path: String) : Unit / Actor[Event] + File {
```

```
  match Actor.receive() {
    Line(s) -> {
      file.append(path, s ++ "\n")
      loop(path)
    }
  }
}

pub fn start(path: String) : Pid[Event] / Spawn + Cancel + Actor[Event] + File {
  file.write(path, "")    # truncate at start
  spawn_actor(() => loop(path))
}
```

Isolating the file write in its own actor has two benefits:

- **The store doesn't wait on disk.** When the store processes a `Create`, it sends a message to the persistence actor and goes back to its work. The write happens in another fiber.
- **Write order is guaranteed.** All events go through the same mailbox, which is processed in FIFO order. No races even if several handlers write to the log at the same time.

In a real system, this actor would have a `Bounded(N, DropOldest)` mailbox to protect against flooding. Here we use the default (Unbounded) for simplicity. The decision is explicit and lives in one line, easy to change.

17.5 HTTP parser

`web.kai` is pure string manipulation. The central piece is `route`, which translates an HTTP request into a domain command:

```
pub fn route(req: HttpReq) : Result[domain.Response, domain.Command] {
  if req.method == "GET" {
    if req.path == "/notes" {
      Ok(domain.List)
    } else {
      route_id(req.path, (id) => domain.Get(id))
    }
  } else if req.method == "POST" {
    if req.path == "/notes" {
      Ok(domain.Create(req.body))
    } else {
      Err(domain.NotFound)
    }
  } else if req.method == "DELETE" {
    route_id(req.path, (id) => domain.Delete(id))
  } else {
    Err(domain.NotFound)
  }
}
```

The return type uses `Result` in a slightly unorthodox way: `Ok` carries a command to execute; `Err` carries an immediate response (404, 400). The convention keeps the signature compact: either the request translates to a valid command, or we have the response directly.

There's also a parser for the first HTTP line (`GET /path HTTP/1.1`) and a serializer that produces the response bytes. Pure functions with no effects, testable with input strings and output comparison.

17.6 main: assemble the pieces

`main.kai` is the glue:

```
import actor
import spawn
import fs.file
import domain
import store
import persistence
import net.tcp
import web

const PORT : Int = 8080
const LOG_PATH : String = "notes.log"

fn main() : Unit / Console + NetTcp + File + Spawn + Cancel + Actor[store.StoreMsg] +
Actor[store.StoreResp] + Actor[persistence.Event] {
  let store_pid = store.start()
  let log_pid   = persistence.start(LOG_PATH)

  match NetTcp.listen("0.0.0.0", PORT) {
    Err(msg) -> println("failed to start the server: " ++ msg)
    Ok(listener) -> {
      println("server listening on port #{PORT}")
      accept_loop(listener, store_pid, log_pid)
    }
  }
}
```

Four lines of "business":

1. Start the store (actor that holds the notes).
2. Start the persister (actor that writes the log).
3. Open a TCP socket on the port.
4. Enter the accept loop.

`main`'s effect row lists everything the program uses: `Console` to print, `NetTcp` for sockets, `File` to write, `Spawn + Cancel` for fibers, `Actor[X]` for each of the three message channels. The signature hides nothing: if `main` did more things, its row would grow accordingly.

The accept loop opens a nursery and spawns a fiber per connection. Note: `n` is not a `Nursery` value that can travel to another function — the compiler rewrites every `n.spawn(...)` to `Spawn.spawn(...)` tagged with *this* nursery's brand, so the `spawn` has to appear lexically inside the block. That's why the accept loop is inline:

```
nursery { n ->
  forever(() => match NetTcp.accept(listener) {
    Err(_)   -> ()
    Ok(conn) -> {
      let _ = n.spawn(() => handle_connection(conn, store_pid, log_pid))
      ()
    }
  })
}
```

Each connection lives in its own fiber. The nursery guarantees that when the loop ends (because someone cancels the listener, or the program receives SIGINT), the child fibers end too. No zombie connection handlers.

And per connection, the handler:

```
fn handle_connection(conn, store_pid, log_pid) {
  let raw = read_request(conn)
  let resp = match web.parse_request(raw) {
    Err(msg) -> domain.ClientError(msg)
    Ok(req)  -> match web.route(req) {
      Err(r)       -> r
      Ok(command)  -> {
        record(log_pid, command)
        store.ask(store_pid, command)
      }
    }
  }
  NetTcp.send(conn, string_to_bytes(web.serialize_response(resp)))
  NetTcp.close(conn)
}
```

Read bytes, parse HTTP, route to a command, log it, ask the store, serialize the response, write to the socket, close. Each step is a pure function or a message to an actor. No shared memory, no locks.

17.7 How this maps to the book

It's worth pointing out which pieces of the book are in play, one by one:

- **Ch. 2** (thinking in kaikai): functions are expressions; `process` returns a tuple in one expression.
- **Ch. 4** (compound types): return tuples (`(Response, [Note], Int)`), records (`Note`), lists with head/tail patterns.
- **Ch. 5** (sum types and match): `Command`, `Response`, `Event` are sum types; matches cover all variants; exhaustiveness is verified by the compiler.
- **Ch. 6** (functions and pipelines): `list.map`, `list.filter` over the list of notes; closures passed to those functions.
- **Ch. 7** (tests): tests over `process` that verify the logic without starting fibers.
- **Ch. 8** (modules): five files each with its `pub`, imports between them.
- **Ch. 9** (protocols): `#[derive(Show)]` for interpolating notes.

- **Ch. 12** (effects): each function declares its row; `handle` doesn't appear directly because the `handle`s live inside `with_mailbox` and `spawn_actor` from the stdlib.
- **Ch. 13** (fibers): `nursery` to structure the accept loop; each connection is a fiber.
- **Ch. 14** (actors): the store and persistence are actors; `with_mailbox` in each client; `spawn_actor` to start them; typed messages.
- **Ch. 16** (tooling): `kai run main.kai` starts it; `kai test` runs the tests in the `store` module.

Nothing here is new in terms of syntax. What's new is the combination: small, orthogonal pieces fitting into a program with real responsibilities.

17.8 How to extend it

Several directions to take this program further:

- **Persistence with recovery.** Today the log is write-only. If the server restarts, notes are lost. A natural extension: at startup, read the log and replay events to rebuild state.
- **Search by content.** The `Get` command searches by id. Add a `Search(String)` that filters by body substring. The pure logic goes in `process`; the `route` `match` gains a branch.
- **TTL per note.** Each note has an expiration. The store, on each `Get`, checks whether the note expired and removes it if so. A `created_at` field joins `Note`; the `Time` effect appears in `loop`'s row.
- **Multiple instances.** Today there's one store. For a bigger service, partition notes across several actors by id hash. `main` starts N stores and routes each request to the right one.
- **Metrics.** A fourth actor that receives events (`request_received`, `note_created`, `error_emitted`) and accumulates counters. `main` starts it, the handlers send it events, an endpoint `GET /metrics` reads.
- **Integration test.** A client program that opens a TCP connection to the server, sends a request, reads the response, verifies it's what's expected. Puts the server in a nursery, runs the client, closes.

Each is an afternoon's work. None requires changing the basic structure: a pure domain, actors with state, fibers for concurrency, modules for separation.

17.9 What this case shows

There's a clear pattern in what we've just put together:

- **The domain is pure.** `Command`, `Response`, `process`: types and functions with no effects. They're tested with inputs and outputs, without starting anything.
- **Actors encapsulate the mutable state.** The store holds the note list; the persister holds the log file. Mutation stays locked inside each actor, invisible to the rest of the program.
- **Fibers handle concurrent work cooperatively.** One fiber per connection. The nursery guarantees that none survives the server.
- **Modules separate responsibilities.** Five files, five topics. Each one can be swapped without touching the other three.

Chapter 18 will apply exactly the same pattern to a very different domain (financial accounting) and you'll see how the structure holds even when the problem changes. The book's closing comes there.

Chapter 18 · Case study: accounting ledger

Chapter 17 showed an HTTP server: many clients, one fiber per connection, actors for state encapsulation. That's the family of problems where concurrency dominates.

This chapter closes the book with a very different case: **a double-entry accounting ledger**. Where precision matters, not concurrency. Where a debit that doesn't match its credit is a bug your auditor will find before you do. Where "it works and the tests pass" isn't enough: you have to show **why** it works, and the type system has a lot to say about that.

Why fintech deserves its own chapter: it's one of the domains where getting it wrong is most expensive and where having guarantees in the type pays off most. Mixing USD with EUR costs money. Adding debits with credits costs money. Allowing a withdrawal without checking the balance costs money. The language's tools (units of measure, contracts, branding) are designed so that these mistakes don't compile, not so that you discover them in production.

The program: a double-entry ledger that holds accounts with balances, records transactions (each with debits and credits), validates that transactions balance, and persists an immutable audit log. Size: about 280 lines across four modules.

18.1 The shape of the program

Four files:

```
ledger/
├── kai.toml           # manifest
├── main.kai           # entry point, runs example operations
├── domain.kai         # types: Account, Movement, Transaction
├── balance.kai        # validation of the debit = credit invariant
├── store.kai          # actor that holds accounts and transactions
└── persistence.kai    # actor that writes the audit log
```

The structure deliberately mirrors chapter 17. What changes is the emphasis:

- **domain.kai** is richer than chapter 17's. Units of measure appear (`Real<USD>`), branded types (`Int<AccountId>`, `Int<TransactionId>`), and a `Movement` sum type that distinguishes debits from credits.
- **balance.kai** is new. A module dedicated to one invariant: the sum of debits in a transaction must equal the sum of credits. It appears both as a pure testable function AND as a **contract** (`requires`) on the registration operation.
- **store.kai** is the same actor pattern, but now it validates balance before accepting a transaction and keeps per-account balances. Transactions are **immutable**: only ever added, never modified.
- **persistence.kai** is the audit log. A strong conceptual idea: in accounting, what's written stays. The file is legal evidence.

18.2 The domain: units, branding, algebraic types

The center of the program is the types:

```
pub unit USD

pub unit AccountId
pub unit TransactionId

#[derive(Show)]
pub type Account = {
  id:      Int<AccountId>,
  name:    String,
  balance: Real<USD>,
}

#[derive(Show)]
pub type Movement
  = Debit(Int<AccountId>, Real<USD>)
  | Credit(Int<AccountId>, Real<USD>)

#[derive(Show)]
pub type Transaction = {
  id:          Int<TransactionId>,
  description: String,
  movements:   [Movement],
}
```

Three type-system decisions worth listing:

- **`Real<USD>` instead of `Real`.** Chapter 10 covered units of measure. They matter especially in this domain: an accounting program that mixes USD and EUR without converting produces numbers that look right but mean nothing. With UoM, that mix doesn't compile. To extend the program to multiple currencies, you declare `unit EUR`, define an exchange rate as `Real<USD / EUR>`, and the type system walks you through it. Without UoM, you discover the bug when a customer complains.
- **`Int<AccountId>` vs `Int<TransactionId>`.** Chapter 10 also covered branded types. Having raw `Int` as ids means passing a transaction id where an account id is expected compiles

fine and blows up at runtime (or worse, silently produces a wrong result). With branding, the type system tells you first.

- `Movement` **as a sum type.** A debit and a credit have the same physical fields (account + amount) but mean different things. Modelling them as two constructors of the same sum type has two effects: the exhaustive pattern match ensures every operation handles them explicitly, and the type system won't let us add "debit amount" with "credit amount" without going through a clear conversion.

Compare with the version without rich types: a record `{ account: Int, amount: Real, kind: String }` where `kind` is `"debit"` or `"credit"`. It works, but `kind: String` allows `"DEBIT"`, `"CR"`, `""`, all invalid. Every function that touches movements has to validate. With the sum type, the invalid **can't be constructed**.

18.3 The central invariant: balance

The `balance.kai` module is small but important. It defines the double-entry invariant: in every transaction, the sum of debits must equal the sum of credits.

```
pub fn total_debits(ms: [domain.Movement]) : Real<USD> {
  match ms {
    []                          -> 0.0<USD>
    [domain.Debit(_, m), ...rest]   -> m + total_debits(rest)
    [domain.Credit(_, _), ...rest]  -> total_debits(rest)
  }
}

pub fn total_credits(ms: [domain.Movement]) : Real<USD> { ... }

pub fn balances(ms: [domain.Movement]) : Bool =
  total_debits(ms) == total_credits(ms)
```

Three pure functions, a boolean invariant. Tests verify the contract piece by piece:

```
test "balances with one debit and one credit" {
  let ms = [
    domain.Debit(1<AccountId>, 100.0<USD>),
    domain.Credit(2<AccountId>, 100.0<USD>),
  ]
  assert balances(ms)
}

test "doesn't balance when amounts differ" { ... }
test "balances with multiple lines" { ... }
```

No actors, no IO, no sockets. Just logic. If six months from now we change how movements are represented, these tests ensure the invariant still holds.

And where chapter 11 pays off: we also declare a **version with a contract**:

```
pub fn apply_if_balanced(ms: [domain.Movement]) : [domain.Movement]
  requires balances(ms)
```

```
  ensures balances(result)
= ms
```

`requires balances(ms)` declares that **calling this function with a movement set that doesn't balance is an error**. If the compiler can prove it statically, it rejects the call at compile time; if not, it inserts a runtime assert. `ensures balances(result)` declares that **the function's result also balances** (trivial here: returns the same list). The two contracts together form the function's legal signature: the preconditions it demands and the postcondition it guarantees.

In a real accounting system, this pattern multiplies: each operation that touches movements carries the domain's contracts in its signature.

18.4 The store: actor with invariants

The store holds the ledger's state: known accounts, registered transactions, next-ID counters.

```
type State = {
  accounts:       [domain.Account],
  transactions:   [domain.Transaction],
  next_account_id: Int,
  next_tx_id:     Int,
}
```

The `process` function takes a command and the state, returns a response and the new state. The important part: **every transaction goes through balance validation before being registered**.

```
Register(desc, movs) ->
  if not balance.balances(movs) {
    (domain.UnbalancedError("..."), s)
  } else {
    match verify_accounts(movs, s.accounts) {
      Some(missing_id) -> (domain.UnknownAccount(missing_id), s)
      None -> {
        let tx = domain.Transaction { id: s.next_tx_id<TransactionId>, ... }
        let updated_accounts = apply_movs(s.accounts, movs)
        let s2 = State { ...s, transactions: [tx, ...s.transactions], ... }
        (domain.TransactionRegistered(tx), s2)
      }
    }
  }
```

Two validations before accepting:

1. **Balance**: the sum of debits equals the sum of credits.
2. **Known accounts**: every account mentioned in the movements is registered.

If either fails, we return an error without modifying state. That's an **atomic transaction**: either it applies in full, or it doesn't apply at all. There's no "half- registered transaction with broken balance".

Transactions **accumulate, are never modified**. The list grows. This is deliberate: a ledger is by design an immutable history. Erasing a past transaction doesn't exist; correcting errors means writing a **new** inverse transaction, which also stays in the history.

That immutability is free in kaikai. Lists are immutable by construction; adding an element creates a new list. "Destructive modification" isn't available to the actor's loop unless we asked for it, with `var` or `Array[T]`. And since the actor recurses with the new state, every "version" of the ledger survives intact until the next step.

18.5 The audit log

The `persistence.kai` module is practically identical to chapter 17's: an actor that receives log lines and appends them to the file. The conceptual difference: for an accounting system, **the file is the truth**.

In real accounting, transaction records are **append-only**: once an entry is written, it stays. Corrections are done by adding new inverse entries, not by modifying the originals. This is regulatory (auditors require it) but also architectural: an event-sourced system that persists every event lets you reconstruct any intermediate state.

```
pub type Event = Line(String)

fn loop(path: String) : Unit / Actor[Event] + File {
  match Actor.receive() {
    Line(s) -> {
      file.append(path, s ++ "\n")
      loop(path)
    }
  }
}
```

Each event the store produces (account created, transaction registered) is sent to the log via `Actor.send`. The log writes them in strict FIFO order. If the disk write falls behind, events accumulate in the mailbox; the store keeps responding.

A production improvement left as exercise: instead of strings, persist a structured format (JSON, CBOR, TLV) that can be replayed at startup to reconstruct the in- memory state. This is **event sourcing** and kaikai allows it naturally.

18.6 The main: run a scenario

`main.kai` doesn't open a socket this time: it just executes a sequence of operations to show the system in action.

```
fn main() : Unit / Console + File + Spawn + Cancel + ... {
  let store_pid = store.start()
```

```
  let log_pid   = persistence.start(LOG_PATH)

  step(store_pid, log_pid, domain.CreateAccount("cash"))
  step(store_pid, log_pid, domain.CreateAccount("sales"))
  step(store_pid, log_pid, domain.CreateAccount("expenses"))

  step(store_pid, log_pid, domain.Register("card sale", [
    domain.Debit(1<AccountId>, 50.0<USD>),
    domain.Credit(2<AccountId>, 50.0<USD>),
  ]))

  step(store_pid, log_pid, domain.Register("team coffee", [
    domain.Debit(3<AccountId>, 8.0<USD>),
    domain.Credit(1<AccountId>, 8.0<USD>),
  ]))

  # Unbalanced attempt: the store rejects it.
  step(store_pid, log_pid, domain.Register("error", [
    domain.Debit(1<AccountId>, 100.0<USD>),
    domain.Credit(2<AccountId>, 50.0<USD>),
  ]))

  step(store_pid, log_pid, domain.QueryBalance(1<AccountId>))
  ...
}
```

`step` is a helper that calls the store, prints the response, and records the event in the audit log. After running `main`, the ledger has three accounts, two registered transactions (sale and coffee), one rejected (the unbalanced attempt), and the final balances reflect the entries. The file `ledger.audit.log` contains one line per account and transaction.

18.7 What makes this case different from chapter 17

Same overall pattern, different emphasis. What appears here that didn't in the previous chapter:

- **Units of measure.** `Real<USD>` in every amount. The type system rejects mixing currencies without explicit conversion (chapter 10).
- **Branded types.** `Int<AccountId>` vs `Int<TransactionId>`. The compiler won't let us mix them up (chapter 10).
- **Contracts.** `requires balances(ms)` in `apply_if_balanced`. The domain invariant lives in the function's signature, verifiable statically or dynamically (chapter 11).
- **Immutability by construction.** Transactions are never modified, only added. The ledger structure guarantees event sourcing.

What appears identically:

- **Sum types and exhaustive match.** `Command`, `Response`, `Movement`. The compiler ensures every operation handles every case.
- **Actors with state.** The store and persistence are actors, same as chapter 17.

- **Modularity.** Pure types separated from the actor machinery. `process` is tested without fibers.
- **Audit log via actor.** The pattern "an actor encapsulates costly IO" repeats.

That **symmetry between the two cases** is the lesson. The domain changes (HTTP vs accounting), the language tools that matter most change, but the program's structure stays the same: pure domain, actors with state, separate persistence, modules by responsibility.

18.8 How to extend it

Ideas to take the example further, in roughly increasing difficulty:

- **Multiple currencies.** Declare `unit EUR`, `unit CLP`, let each account carry its currency as type parameter. Cross-currency transactions require an explicit exchange rate (`Real<USD / EUR>`).
- **Account types.** Distinguish assets, liabilities, equity, income, expenses. Each type has different rules for what "debit" and "credit" mean (in an asset account, debit increases; in a liability, debit decreases). Branded types like `Account<Asset>` capture this.
- **Accounting periods.** Close the fiscal year: every income and expense account transfers to retained earnings, and the ledger starts the new period with opening balances.
- **Reconstruction from the audit log.** At startup, read the log file and replay events to rebuild the in-memory state. This is canonical event sourcing.
- **Reports.** Income statement, balance sheet, ledger by account. Each is a pure function that walks the transactions and produces a formatted string.
- **Non-negative balance enforcement.** If you model a "non-negative balance" account as `Real where self

> *= 0.0` (refinement type from chapter 11), the type system rejects any movement that would push it negative.*

- **Serve over HTTP.** Combine this program with chapter 17's: the accept loop stays the same, the handler that used to call the notes store now calls the ledger store. The program's structure changes very little.

Each one moves the system closer to a production system. None forces you to rewrite the earlier pieces. Orthogonality pays off.

18.9 Why fintech is a good testbed

Fintech is one of the few domains where the industry welcomes tooling that weighs in on the development process. If a function disburses the wrong amount, the company would rather the compiler catch it than discover it in production. If the system guarantees that debits match credits, the regulator sleeps better. If types distinguish currencies, next quarter's exchange-rate change won't introduce a subtle bug.

Languages that offer those guarantees — Haskell, OCaml, F# — have had a good reception in fintech historically. kaikai aims for the same level of guarantees with less

ceremony: units without external packages, contracts in the signature, branding without macros. The promise is that you can write code as directly as in Python or Go, with the guarantees a strong type system gives you.

Does the promise hold? That's for the person writing the code to decide. This chapter tried to show a slice of a domain where the bet pays off.

18.10 Philosophy: the book's closing

There's a pattern this book has been proposing, chapter after chapter, without naming it explicitly until now. Worth naming at the end.

A real program is made of small, orthogonal pieces. Pure types describing the domain. Pure functions transforming the domain. Actors wrapping the mutable state that needs to persist between calls. Fibers running concurrent work cooperatively. Modules separating responsibilities. Contracts placing domain invariants in the signatures of functions that preserve them.

Each piece is tested in isolation. Each piece declares in its signature everything it does. Each piece can be replaced without touching the rest.

This isn't exclusive to kaikai. It's described, in different words, in Brooks's *No Silver Bullet*, Hickey's *Simple Made Easy*, Marlow and Goldsmith's *Out of the Tar Pit*. What kaikai does is offer a syntax and a type system that **make this style natural**. Signatures that hide nothing are free because effects are in the type. Pure functions are cheap because immutability is the default. Actors are a library because algebraic effects make them possible. Domain invariants live in the signature because contracts live in the language.

If you keep one idea from reading this book, let it be this: **the language isn't what matters; what matters is what it lets you build, and what it helps you avoid building wrong**. kaikai bets that with effects, fibers, contracts, units, and holes in place, the programmer writes less wrong code and more code that deserves to be in production. If the bet works for you, this book has done its job.

Thanks for reading this far. The compiler, the stdlib, the design documents, and the examples live at `github.com/kaikailang-org/kaikai`. There's an emerging community, issues to close, language pieces still taking shape. If you find this experiment interesting, there's room for you to help make it better.

Appendices

Appendix A · Three-stage bootstrap

This appendix tells how the kaikai compiler is built from nothing but a plain C compiler. It's not part of the language the reader needs to know to program; it's part of the project's story, an engineering decision with lasting consequences.

If you find yourself asking "why does it matter how the compiler is compiled?", the short answer: because it defines whom you trust and why. A language whose compiler is built from a plain `cc` and nothing else is a language anyone can audit, reproduce, and maintain. That's technical freedom few modern languages offer.

A.1 The bootstrap problem

A compiler is a program. Like any program, it has to be compiled to run. New compilers run into an immediate paradox: what compiles the compiler the first time?

Three historical answers:

- **Write it in C (or assembly) the first time.** The classical route. GCC, MRI Ruby, CPython, V8 were born this way. The cost: the new compiler carries C's surface forever, or gets rewritten in itself later through a costly migration.
- **Write it in an existing language that already has a compiler.** That's what Rust did with OCaml initially, what Swift did with C++. It inherits the host language's dependencies: compiling Rust required installing OCaml, until Rust was rewritten in itself.
- **Incremental self-hosting.** Start with a small compiler (for a subset of the language), written in something portable, use it to compile a bigger compiler (written in the language), and so on. That's what kaikai does with its three stages.

A.2 Stage 0: the compiler in C

`stage0` is a compiler written in standard C. Its only dependency is any C compiler:

```
$ cc stage0/*.c -o kaic0
```

No frameworks, no generators, no exotic libraries. Plain C. The file `stage0/runtime.h` is the runtime for compiled programs: reference counters, list and string primitives, panic. All of that fits in a few thousand lines.

What does `kaic0` compile? **kaikai-minimal**, a deliberate subset of the language. The grammar and constructs in kaikai-minimal are documented in `docs/kaikai-minimal.md`. What's **not** in kaikai-minimal: algebraic effects, fibers, protocols, contracts, units of measure. The minimum to write a compiler.

Stage 0 is built to be auditable. A programmer who wants to understand what's going on can read the C source line by line: lexer, recursive-descent parser, simple type checker, C emitter. Five files.

A.3 Stage 1: the compiler in kaikai-minimal

Once `kaic0` works, we write a new compiler **in kaikai-minimal**. This compiler, `stage1`, does something `kaic0` can't: it compiles the **full** language. Effects, fibers, protocols, contracts, all of it.

```
$ kaic0 stage1/main.kai -o kaic1
```

`kaic0` compiles `stage1` to produce an executable `kaic1`. From here on, the C compiler is out of the picture: `kaic1` is enough to process programs that use everything kaikai offers.

It's the first "self-validation": the compiler written in kaikai-minimal proves that kaikai-minimal is expressive enough to implement a full compiler. If it weren't, this step wouldn't terminate.

A.4 Stage 2: full kaikai, self-hosted

`stage2` is the definitive version. Written in **full kaikai** (not the minimal subset), using effects, fibers, and everything the language offers. End-to-end idiomatic kaikai code.

```
$ kaic1 stage2/main.kai -o kaic2
```

`kaic1` compiles `stage2`, producing `kaic2`. This is the compiler that gets distributed. The user installing kaikai gets `kaic2`, not `kaic0` or `kaic1`.

The three stages, end to end:

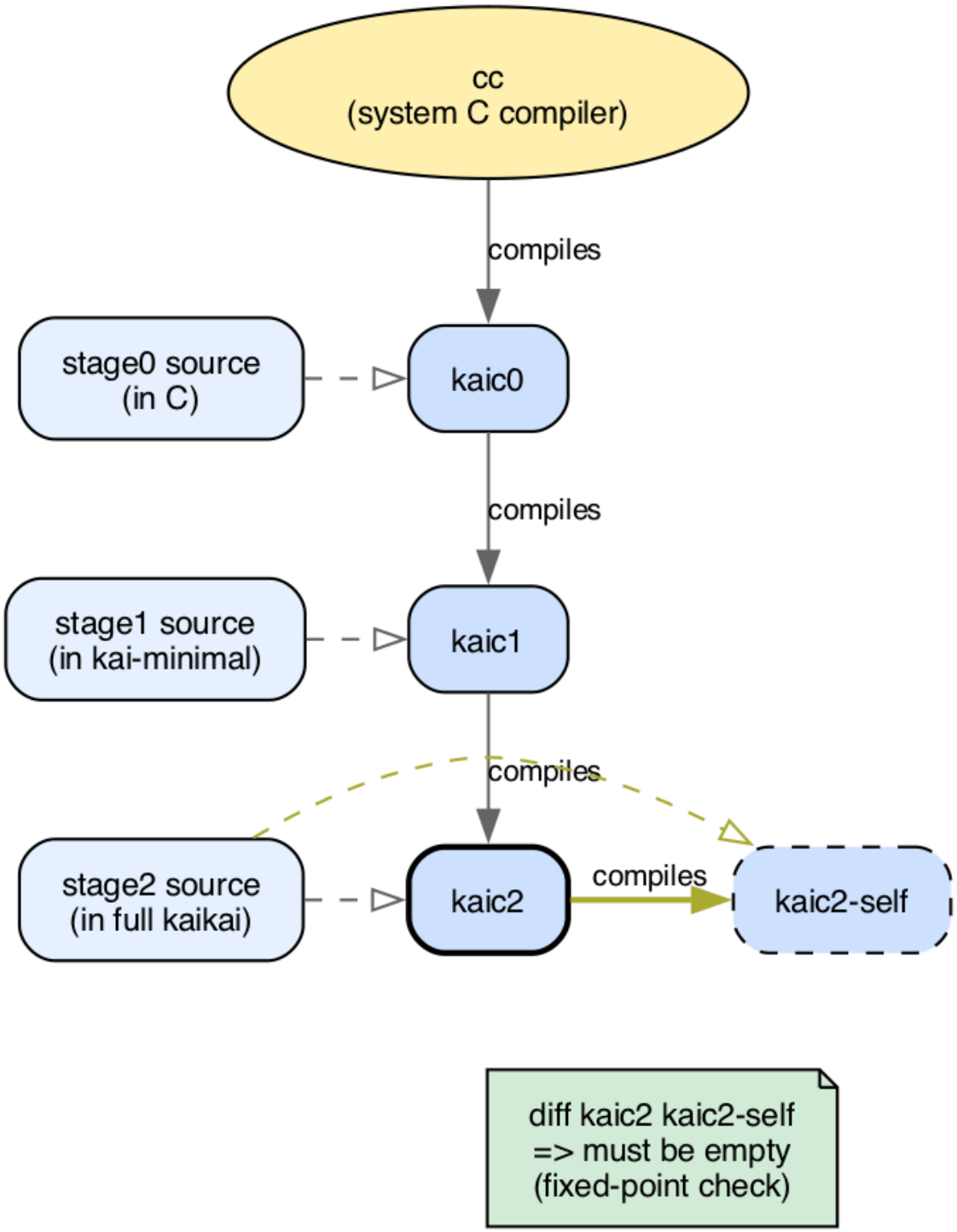

Figure A.1 · *The three-stage bootstrap. Solid arrows show which compiler produces which binary; the dashed arrows point from each stage's source to the binary it becomes. Only* `cc` *is external; once* `kaic0` *exists, the chain is self-contained. The bottom row's fixed-point check (*`kaic2` *compiling itself) is what §A.5 covers.*

A.5 The fixed point: bootstrap validation

There's a critical check at the end of the process:

```
$ kaic2 stage2/main.kai -o kaic2-self
$ diff kaic2 kaic2-self
```

They must be **bit-for-bit identical**. This means `kaic2` compiled by `kaic1` produces exactly the same thing as `kaic2` compiled by itself. In other words: `kaic2` is a fixed point of the compilation process.

Why does this matter? Two reasons:

- **Detection of subtle compiler bugs.** If two compiler versions (`kaic1` and `kaic2`) produce different binaries from the same source, one of them has a bug. The fixed point confirms the whole chain converges to the same answer.
- **Resistance to the Thompson attack.** Ken Thompson published in 1984 a famous essay ("Reflections on Trusting Trust") showing that a malicious compiler can insert invisible backdoors that survive recompilation. The classical defense is **diverse double-compiling**: compile with two distinct chains and compare. The fixed point is the modern version of that idea: if the whole chain converges to one binary, and that binary is reproducible from auditable sources, trust is justified.

A.6 Reproducible from a `cc`

The practical consequence of the three-stage bootstrap is this: **with just a C compiler, you can rebuild kaikai entirely from source**. No need to have kaikai previously installed. No need to trust a binary downloaded from a web page. No `curl | bash`.

```
$ git clone https://github.com/kaikailang-org/kaikai
$ cd kaikai
$ make
```

Underneath, `make` runs the three steps: stage 0 with cc, stage 1 with stage 0, stage 2 with stage 1. It ends with a `kaic2` in the `bin/` directory, and the fixed-point verification ensures it's the right one.

This is technical freedom few modern languages give you. Rust requires a previous Rust (downloaded as a blob). Go requires a previous Go. Swift requires a previous Swift. kaikai doesn't: your C compiler is the entry point.

A.7 Costs and trade-offs

The three-stage bootstrap has costs. Worth listing because every engineering decision has them:

- **Stage 0 has to be maintained whenever kaikai-minimal changes.** If a new language feature falls within the minimal subset, stage 0 has to learn it. That means new C code, written by hand.
- **It's expensive to add dependencies to the compiler.** Stage 2 might benefit from an external library (a parser combinator, an exotic data structure). But that library would have to be available in stage 1, which only understands kaikai-minimal. The pressure is toward keeping stage 2 self-contained.

- **Building everything from scratch takes time.** Not much (the three steps together take less than a minute on a reasonable machine), but more than a single `cargo build`. For continuous CI, that matters.

Is it worth it? The project's answer is yes, and the reason is philosophical: the compiler is the most trusted piece of software in the ecosystem. If the chain of trust can be audited end-to-end from plain C, then the rest follows.

A.8 Going further

The design decisions live in `docs/design.md` at `github.com/kaikailang-org/kaikai`. The physical files are under `stage0/`, `stage1/`, `stage2/` of the repository. Ken Thompson's essay *Reflections on Trusting Trust* (CACM 1984) is worth reading if you care about the philosophical justification for this approach.

Appendix B · Perceus in depth

§13.2 covers Perceus in one page: the compiler analyzes the program, knows the exact point where each value stops being used, and inserts a `drop` there that decrements the reference counter and frees the memory if it reaches zero. No GC, no borrow checker, no lifetime annotations.

This appendix goes deeper. What it's for: if you come from Rust and wonder why kaikai doesn't need a borrow checker; if you come from Java and wonder why kaikai doesn't need GC pauses; or if you simply want to understand how an idea published in 2021 changed the balance between RC, GC, and ownership.

You don't need to read this appendix to program in kaikai. §13.2's model is enough. This text is for whoever wants to see the gears.

B.1 The paper, in one sentence

The paper that invented Perceus is **"Perceus: Garbage Free Reference Counting with Reuse"** (Reinking, Xie, de Moura, Leijen — PLDI 2021). The central sentence:

> *Inserting reference count operations* after *type checking, using* precise last-use information*, makes RC competitive with tracing GC while keeping deterministic deallocation.*

Three words carry weight:

- **After type checking.** The analysis runs over the already-typed program, not the raw AST. That gives enough information to reason about shape and uniqueness.
- **Precise last-use.** For each variable, the compiler identifies the exact place where it's used for the last time. After that point, the `drop` is safe.
- **Reuse.** This is the trick that makes Perceus competitive with GC: when a value is about to be freed and a value of the same shape is about to be created, the compiler reuses the same memory.

Let's take it apart.

B.2 Step by step: where the drops go

Take this simple function:

```
fn example(xs: [Int]) : Int {
  let n = list.length(xs)
  let s = list.sum(xs)
  s + n
}
```

What has to happen to `xs` when the function ends? It depends on who created it. If the list is owned by the function (it was created inside, or was moved in as an argument), it has to be freed. If it's shared with others, not.

Perceus analyzes the body and finds:

- `xs` is used in line 2 (`list.length`).
- `xs` is used again in line 3 (`list.sum`).
- After that it isn't used again.

The compiler inserts:

```
fn example(xs: [Int]) : Int {
  let n = list.length(dup(xs))   # dup: increment xs's rc
  let s = list.sum(xs)           # last use: passes ownership
  s + n
}
```

`dup(xs)` increments the RC because `list.length` will "consume" a reference (doing its own drop at the end). The second use, `list.sum(xs)`, no longer needs `dup`: it's the last use, and the reference the function already holds is transferred.

`list.sum` and `list.length` internally do the same: whenever they walk the list's tail, they decide whether they're at the last use. If so, they don't duplicate. If not, they do.

The final program has **the minimum number of operations possible** on the counters. That minimisation is the paper's core.

B.3 Reuse in place

The most interesting part of Perceus is **reuse in place**. When a function is about to free a value of some shape and create another one of the same shape right after, the compiler reuses the same memory block. No free, no allocation: overwrite.

The canonical example is `map` over lists:

```
fn map[a, b](xs: [a], f: (a) -> b) : [b] {
  match xs {
    []          -> []
    [h, ...rest] -> [f(h), ...map(rest, f)]
  }
}
```

Without reuse, this `map` would:

1. Free the cons cell `[h, ...rest]` (after extracting both).

2. Allocate a new cons `[f(h), ...]`.

With reuse:

1. Overwrite the existing cons with the new head.

Same cons, same memory position, same cost as a mutation. But **the program's result is identical**: the function is still pure from the programmer's view.

The conditions for reuse in place are three, all verifiable at compile time:

1. **The receiver has a unique reference** (RC == 1).
2. **The new value has the same shape**: same tag, same fields.
3. **No live aliasing elsewhere**: nobody else holds a pointer to the original cons.

When all three hold, the compiler emits code indistinguishable from a destructive mutation. But conceptually it's still immutable: if the program changes and two references to the cons appear, reuse silently turns off and falls back to "free + allocate". The programmer doesn't notice.

This optimization is what makes algorithms like `map`, `filter`, AVL trees, list parsing, **as fast as the mutable versions in imperative languages**. It's why Koka and Lean 4 (which also use Perceus) can compile competitively with C.

B.4 Compared to Rust's `Rc<RefCell<T>>`

If you come from Rust, this rings a bell: Rust also has RC (`Rc<T>`, `Arc<T>`). How is Perceus different?

Aspect	Rust `Rc<T>`	Perceus in kaikai
Who decides increments	Programmer (clone)	Compiler
Who decides decrements	Automatic destructor	Compiler
Annotations required	`Rc<T>`, `Arc<T>`, `Rc::clone(&x)`	none
Content mutability	Needs `RefCell<T>`	Immutable by default
Cycles	Silent memory leak	Impossible (no mutation)
Single-use cost	Pays the `Rc` always	No cost: compiler emits no RC

The strongest difference is the last one. In Rust, when you declare `Rc<T>` you pay the counter ALWAYS, even when the value has a single use. In kaikai, **if the value has a single use, no RC is emitted**. The compiler inserts the `dup`s only where they're needed, after last-use analysis.

That means a pure functional kaikai program, where nothing is genuinely shared between several variables, has zero RC overhead. Lists, records, closures all get freed at their last use without going through a counter. RC only appears when the program actually needs to share.

B.5 What about cycles?

The classical critique of RC is that **it doesn't handle cycles**: if two values reference each other and nobody else references them, neither one ever reaches 0, and they leak. That's why Python has a "cycle collector" on top of its RC, and Rust forces you to use `Weak` manually.

Why doesn't Perceus need either?

Because values in kaikai are immutable.

To create a cycle, you need mutation: A points to B, then you modify B to point to A. Without mutation, you can't build the second step. When you build B, A doesn't exist yet; when you build A pointing to B, A isn't reachable from B.

The only exceptions are mutation mechanisms kaikai provides deliberately: local `var` (which the compiler masks and restricts), `Ref[T]`, actor mailboxes, mutable arrays. All of them have specific disciplines that prevent cycles (mailboxes, for example, live in a specific actor and the runtime's GC manages them).

In practice: a typical kaikai program doesn't build cycles. Data structures are trees (lists, AVL, JSON, ASTs). When a programmer wants something with cycles (a graph, an LRU cache), they reach for explicit structures encoding adjacency without direct pointers: indices in an array, identifiers in a map. It's more work, but the intellectual cost stays visible where it belongs.

B.6 Why per-fiber separation simplifies RC

Chapter 13 mentions that each fiber has its own heap. This isn't just error isolation: **it's what makes Perceus lock-free**.

If two fibers shared pointers to the same value, the counter's increments and decrements would have to be **atomic**. Atomic means the CPU emits a memory barrier, synchronizes with other cores, pays overhead. In programs with many fibers, that overhead is enormous.

When each fiber has its own heap, counters are local. **No synchronization**. A `dup` is a `++` on an integer, without barriers. A `drop` is a `--`, ditto.

This is why kaikai's fiber model and Perceus are designed together: each enables the other. Isolated fibers allow RC without synchronization; efficient RC allows millions of fibers without paying GC.

When two fibers need to share a value, they do so via an explicit operation (`send` to a mailbox, return from an `await`). The runtime copies the value (or moves it, if safe) to the receiver's heap. The transfer is explicit in the program, and therefore predictable.

B.7 What it costs and what you get

Perceus has costs. Worth listing:

- **Static analysis is compiler work.** Slower than raw types. In kaikai, the analysis is integrated with inference and runs in milliseconds for typical programs.

- **When a value is genuinely duplicated, it pays the RC.** If you have many heavy shared structures, the `dup` / `drop` costs.
- **Reuse in place depends on static uniqueness.** If the analysis can't prove uniqueness, the optimization doesn't apply and you fall back to the safe version.

What you get:

- **Determinism.** You know exactly when each value gets freed. No pauses, no "the GC ran now", no nondeterminism between runs.
- **Memory predictability.** Peak memory can be estimated by looking at the program. There's no "the GC waited too long and it piled up".
- **Zero annotations.** No `'a`, no `&mut`, no `Rc::clone(&x)`. The compiler does the work.
- **Composes with effects.** RC doesn't interact with effects in weird ways. `handle` and `resume` have no hidden GC overhead.

B.8 Going further

If this appendix leaves you wanting more, the sources:

- **Reinking, Xie, de Moura, Leijen, *"Perceus: Garbage Free Reference Counting with Reuse"*** (PLDI 2021). The paper. It's readable.
- **Lorenz, Leijen, *"Reference Counting with Frame Limited Reuse"*** (ICFP 2023). An extension that improves reuse when the shape doesn't quite match.
- **Koka language documentation** (Daan Leijen et al.). Koka is where Perceus was born; much of the vocabulary ("reuse in place", "borrowed binds", "drop specialisation") comes from there.
- **Lean 4 RC implementation** (de Moura et al.). Lean 4 also uses Perceus, with an approach closer to formal certification.

In the kaikai source, the analysis lives in `stage2/perceus.kai`. If you want to see how it's implemented, that's the starting point.

Appendix C · Operator table and precedence

This appendix lists every operator in the language with its precedence and associativity. It's meant as a quick reference: when you're not sure how `a |f|> g` parses, you look it up and move on.

Precedence is numbered from highest (1, binds first) to lowest (8, binds last). Each level resolves against the level immediately above.

C.1 Main table

Level	Operators	Associativity
1	call `f(args)`, field `.`, index `[i]`	Postfix
2	unary `-`, `not`	Prefix
3	`*`, `/`, `//`, `%`	Left
4	`+`, `-` (binary), `++` (string concat)	Left
5	`==`, `!=`, `<`, `>`, `<=`, `>=`	**Non-associative**
6	`and`	Left (short)
7	`or`	Left (short)
8	`|>`, `|`, `||`, `|?`	Left

Per-level details

- **Level 1 (postfix).** Left-associative: `a.b.c` means `(a.b).c`; `f(x)(y)` means `(f(x))(y)`.
- **Level 2 (prefix).** Unary `-x` and `not x`. Apply before any binary operator.
- **Level 3 (multiplicative).** `*` and `/` on `Int` and `Real`, `//` is integer division, `%` is modulo. Left- associative: `a / b / c` is `(a / b) / c`.
- **Level 4 (additive).** Binary `+` and `-` on numbers, `++` to concatenate strings.
- **Level 5 (comparisons). Don't chain.** Writing `a < b < c` is a syntax error. The correct form is `a < b and b < c`. This eliminates the bug class where `1 == 1 == true` parses unexpectedly.
- **Level 6 (`and`) and 7 (`or`).** Short-circuit logical. `and` binds tighter than `or`, so `a or b and c` is `a or (b and c)`.

- **Level 8 (pipes).** Four operators at the same level: `|>` (apply), `|` (map), `||` (flat-map), `|?` (filter). All left-associative. `xs | f |> g` is `(xs | f) |> g`.

C.2 Forms that aren't operators

Some forms that look like operators in other languages are separate syntactic constructs in kaikai:

- `=` **(`let` binding)**: part of `let`'s syntax, not an operator. `let x = expr` is a declaration.
- `:=` **(write to mutable cell)**: part of `var`'s syntax. `name := expr` is the sugared form of `State.set(name, expr)`. It doesn't participate in the precedence table.
- `@name` **(read from mutable cell)**: sugared read of `State.get(name)`. It's a prefix of the name, doesn't participate in the table.
- `->` **(in `match`, in lambdas, in `handle`)**: syntactic separator, not an operator.
- `/` **(in effect signature)**: separates the return type from the effect row: `fn f() : Int / Log`. Only appears in signature position.
- `!` **(propagation of `Option` / `Result`)**: postfix on expressions returning `Option[T]` or `Result[E, T]`, performs early propagation of the error or `None`. Rough equivalent of Rust's `?`. It isn't a binary operator.
- `?` **and `?name` (holes)**: hole markers, not operators.
- `...` **(spread)**: appears in record and list literals (`[h, ...t]`, `Point { ...p, x: 10 }`); also in patterns. Not an operator.
- `<>` **(unit-of-measure annotation)**: `1.50<USD>`, `Real<EUR>`. The unit sits inside the angle brackets; these aren't the operators `<` and `>`.

C.3 Useful equivalences

Some operators are syntactic sugar for calls to stdlib functions. Knowing the equivalences helps when the operator doesn't seem to work, or when you want to see the underlying mechanism.

Operator	Equivalent to
`` `x ``	`> f` ``
`` `x ``	`> f(_, b)` ``
`` `xs ``	`f` ``
`` `xs ``	
`` `xs ``	`? p` ``
`a ++ b`	`string_concat(a, b)` (when they're `String`)
`a[i]`	`Mutable.array_get(a, i)`
`a[i] := v`	`Mutable.array_set(a, i, v)`
`@name`	`State.get(name)` (with `var`)
`name := v`	`State.set(name, v)` (with `var`)

C.4 Reminder: comparison is non-associative

The rule at level 5 deserves its own callout because it's where readers from other languages get caught most.

In most languages:

```
1 == 1 == 1     # Python: false (1 == 1 ⟶ true, true == 1 ⟶ false)
1 < 2 < 3       # C: 1 (1 < 2 ⟶ 1, 1 < 3 ⟶ 1; not what it looks like)
```

In kaikai, neither one compiles. The compiler requires you to spell out the intent:

```
1 == 1 and 1 == 1       # clearly: two comparisons
1 < 2 and 2 < 3         # range: each side compared explicitly
```

This eliminates a small but irritating class of bugs. For anyone coming from Python who finds writing more annoying, remember the rule is the same in Pascal, Eiffel, and SQL. It isn't a kaikai oddity.

Appendix D · Stdlib effects catalog

This appendix summarises the effects the stdlib exposes. Reference material: when a function's signature in the documentation says `: Unit / X`, you come here to confirm what `X` provides.

The full specification lives at `github.com/kaikailang-org/kaikai/docs/effects-stdlib.md`. Here we show the effect's declaration and what it's for.

D.1 Basic IO

Console

```
effect Console {
  print(s: String)  : Unit
  eprint(s: String) : Unit
}
```

Print to stdout and stderr. Each operation appends a newline. The runtime's default handler writes to the corresponding file descriptor.

Stdin

```
effect Stdin {
  read_line() : Result[String, String]
}
```

Read a line from standard input. Returns `Err(reason)` on EOF or read error.

Env

```
effect Env {
  get(name: String) : Option[String]
  args()            : [String]
}
```

Access to environment variables (`PATH`, `HOME`, etc.) and to the command-line arguments (`argv`).

File

```
effect File {
  read_file(path: String)                : Result[String, String]
  write_file(path: String, content: String): Result[Unit, String]
  append(path: String, content: String)    : Result[Unit, String]
  exists(path: String)                   : Bool
  delete(path: String)                   : Result[String, Unit]
  rename(from: String, to: String)       : Result[String, Unit]
}
```

Operations on files. For anything non-trivial (streams, directories, permissions), see `fs.dir` and the auxiliary modules.

D.2 Time and randomness

Clock

```
effect Clock {
  now()        : Int       # nanoseconds since epoch
  sleep(ms: Int)  : Unit / Cancel
}
```

Clock and sleep. `sleep` is a yield point (carries `Cancel`).

Random

```
effect Random {
  int(min: Int, max: Int)   : Int
  real()                    : Real
  shuffle[a](xs: [a])       : [a]
}
```

Pseudo-random generation, not crypto-safe. For secrets, see `SecureRandom`.

SecureRandom

```
effect SecureRandom {
  bytes(n: Int) : [Byte]
}
```

Cryptographically secure random bytes (via `/dev/urandom` or the system equivalent).

D.3 Network

NetTcp

```
effect NetTcp {
  connect(host: String, port: Int)   : Result[Conn, String]
```

```
  listen(host: String, port: Int)    : Result[Listener, String]
  accept(l: Listener)                : Result[Conn, String]
  send(c: Conn, data: [Byte])        : Result[Int, String]
  recv(c: Conn, max: Int)            : Result[[Byte], String]
  close(c: Conn)                     : Unit
}
```

Byte-level TCP sockets. Blocking operations (`connect`, `accept`, `send`, `recv`) suspend the fiber via the runtime's reactor when it lands (v1 makes them block the OS thread).

`NetUdp` and `NetDns`

UDP (`bind` / `send` / `recv` / `close`) and DNS (`resolve`). Same style as `NetTcp`. The alias `Net = NetTcp + NetUdp + NetDns` is useful when a function uses all three.

D.4 Processes and signals

`Process`

```
effect Process {
  run(cmd: String, args: [String]) : Result[ProcessResult, String]
  pid()                            : Int
}
```

Run external commands as subprocesses. `ProcessResult` holds `exit_code`, `stdout`, and `stderr`.

`Signal`

```
effect Signal {
  handle(sig: SignalKind, action: () -> Unit) : Unit
}
```

Register handlers for system signals (`SIGINT`, `SIGTERM`, etc.). Useful for orderly cleanup on process shutdown.

D.5 State

`State[T]`

```
effect State[T] {
  get() : T
  set(v: T) : Unit
}
```

The canonical effect for carrying mutable state in an encapsulated way. The syntax `var name = init` is sugar over a `State[T]` `handle` (chapter 12 §12.7).

`Reader[T]` and `Writer[W]`

```
effect Reader[T] {
  ask() : T
}

effect Writer[W] {
  tell(w: W) : Unit
}
```

Read-only environment (immutable configuration) and output accumulation (logging, tracing). Classical effect-calculus patterns.

`Mutable`

```
effect Mutable {
  ref_make[T](init: T)            : Ref[T]
  ref_get[T](r: Ref[T])           : T
  ref_set[T](r: Ref[T], v: T)     : Unit
  array_make[T](n: Int, init: T)  : Array[T]
  array_length[T](a: Array[T])    : Int
  array_get[T](a: Array[T], i: Int)     : T
  array_set[T](a: Array[T], i: Int, v: T)  : Unit
  array_grow[T](a: Array[T], n: Int, init: T): Unit
}
```

The effect behind `Ref[T]` and `Array[T]`. Follows the discipline of **observable effects** (chapter 12 §12.7): an `array_set` requires `Mutable` in the row only when the mutation is visible to the caller. An array created locally and returned doesn't require `Mutable`.

D.6 Errors and control

`Fail`

```
effect Fail {
  fail(msg: String) : Nothing
}
```

Abort with a message. The operation returns `Nothing` (the empty type), so the type system guarantees you can't call `resume` after `Fail.fail`. It's the canonical pattern for "lightweight exception" in kaikai.

`Cancel`

```
effect Cancel {
  raise() : Nothing
}
```

Cooperative cancellation. The scheduler injects `Cancel.raise()` into a canceled fiber at the next yield point. The fiber can install a `Cancel` handler for cleanup (chapter 13).

D.7 Concurrency

`Spawn`

```
effect Spawn {
  spawn[T, e](f: () -> T / e) : Fiber[T]
  await[T](f: Fiber[T])       : T
  select[T](fs: [Fiber[T]])   : T
  yield()                     : Unit
  cancel[T](f: Fiber[T])      : Unit
}
```

Create fibers, await them, race them, yield, cancel. `nursery { n -> ... }` from chapter 13 is sugar over `handle ... with Spawn as n { ... }`.

`Actor[Msg]`

```
effect Actor[Msg] {
  self()                      : Pid[Msg]
  send(pid: Pid[Msg], msg: Msg)  : Unit / Cancel
  receive()                   : Msg / Cancel
}
```

The effect underlying chapter 14's actor model. `with_mailbox { ... }` and `spawn_actor(...)` are the stdlib wrappers that install this handler.

`Link` and `Monitor`

```
effect Link {
  link(pid: Pid[_]) : Unit
}

effect Monitor {
  monitor(pid: Pid[_])        : MonitorRef
  demonitor(ref: MonitorRef)  : Unit
}
```

BEAM-style supervision (chapter 14 §14.6). Links are bidirectional: if one falls, the other receives `Cancel.raise()`. Monitors are unidirectional: the observer receives a `MonitorDown` message when the observed actor terminates.

D.8 Interoperability

`Ffi`

```
effect Ffi
```

The effect carried by every function declared with `extern "C" fn`. No operations of its own: it's a marker so the type system knows which functions touch code not audited by kaikai.

Chapter 16 §16.9 covers the declaration syntax, type mapping at the boundary, linking with C shims, and what FFI v1 does and doesn't support.

D.9 Composition: the `Io` alias

```
type Io = Console + Stdin + Env + File
```

Bundle of the most common effects for IO to the operating system. A function that says `/ Io` declares it can touch the console, read stdin, read env vars, manipulate files. It's the equivalent of "this function is not pure, it does things with the system".

D.10 Default handlers

When `main` declares any of these effects in its row, the runtime automatically installs a default handler:

- `Console`, `Stdin`, `Env`, `File` → IO to the system.
- `Clock`, `Random`, `SecureRandom` → system clock and RNG.
- `NetTcp`, `NetUdp`, `NetDns` → POSIX sockets.
- `Process`, `Signal` → POSIX calls.
- `Mutable` → real heap assignments.
- `Spawn`, `Cancel` → the runtime's fiber scheduler.

These handlers can be intercepted: any `handle … with X { … }` the user installs wins over the runtime's handler for the duration of the `body` block. That's what enables mocking in tests, capturing output, simulating the clock, etc.

Appendix E · Glossary

Terms the book uses with a specific meaning. Definitions are short and point to the chapter where the term appears.

A

Actor. A fiber with a typed mailbox on top. Processes messages it receives in order, keeps internal state between messages. In kaikai, actors are a library built on the `Actor[Msg]` effect.

Algebraic effect. An effect in the sense of chapter 12: a language construct that declares operations (with their signatures) without deciding how they're implemented, and lets a `handle` decide what each operation means at any point in the program.

Array. Data structure with constant-time indexed access. In kaikai, `Array[T]` is mutable; mutation lives under the `Mutable` effect.

Assert. Construct used inside `test`, `check`, and contract blocks. If the condition is `false`, the block or the program aborts.

B

Backpressure. Mechanism by which a slow consumer slows down a fast producer. In kaikai, the mailbox policy `Bounded(N, BlockSender)` blocks the sender when the mailbox is full.

Bottom type. A type with no inhabitants, written `Nothing`. A function returning `Nothing` can't return normally: it either loops forever, aborts, or calls a non-returning effect. That's why `Fail.fail(...) : Nothing` is the natural signature for an operation that doesn't resume.

Bounded. A mailbox policy with fixed capacity. When full, behavior depends on the overflow rule: `DropOldest`, `DropNewest`, or `BlockSender`.

Branded type. A numeric or string type marked with a symbolic unit (`Int<UserId>`) so the type system distinguishes it from other types with the same underlying representation. A special case of the units-of-measure system.

C

Cancellation. Asking a fiber to end before its natural completion. In kaikai it's an effect: `Cancel.raise()`. The fiber can install a handler to clean up before unwinding.

Capability. The binding that a `handle ... with Effect` gives the body to invoke the effect's operations. By default, the capability's name is the effect's name: inside a `Log` handler, you call `Log.log(...)`. The `with X as name` syntax lets you rename it.

Closure. A function value that captures variables from the scope where it was created.

Continuation. "The rest of what's left to do" at a point in the program. In kaikai it appears as the `resume` argument that effect handlers receive: calling `resume(v)` continues the body's computation with `v`.

Contracts. `requires` (precondition) and `ensures` (postcondition) in a function's signature, verified statically when possible and dynamically when not. Chapter 11.

D

Default handler. The handler the runtime installs automatically around `main` for certain effects (`Console`, `File`, `Spawn`, etc.). User-installed handlers via `handle ... with X` take priority over the default while in scope.

Double-entry. Accounting system in which every transaction has debits equal to credits. Appears in chapter 18's case study.

Drop. The operation Perceus inserts at the last use of each value to decrement its reference count. If it reaches zero, memory is freed.

E

Effect. See *Algebraic effect*.

Effect row. The list of effects in a signature: `Int / Log + Fail + State[Int]`. Built with `+` and treated as a set, not a sequence.

Event sourcing. Architectural pattern where the system's state is the sum of events that occurred; the event log is the source of truth and the in-memory state is reconstructed by replaying it. Appears in chapter 18.

Exhaustiveness. Property the compiler verifies on `match`: every inhabitant of the scrutinee's type must be covered. Chapter 5.

F

Fiber. A unit of cooperative execution. In kaikai a fiber is lightweight (hundreds of bytes), has its own heap, shares no memory with other fibers, and yields control only at explicit yield points. Chapter 13.

Function coloring problem. The problem where a language feature (typically `async/await`) splits functions into two incompatible colors, and changing one function infects

every function that calls it. Discussed in Bob Nystrom's 2015 essay *"What Color is Your Function?"*; kaikai solves it by routing everything through effect rows.

H

Handler. The `with Effect { ... }` block that decides what to do with invocations to an effect. For each operation it receives the arguments and a `resume`, and decides whether to continue the body.

Hole. A `?` or `?name` expression that compiles but aborts at runtime if execution reaches it. Used for top- down design (human) and as an interface for AI agents. Chapter 15.

I

Immutability by default. Values in kaikai are immutable by construction: `let x = ...` declares a binding that doesn't change. Mutable constructs (`var`, `Ref[T]`, `Array[T]`) are the explicit exception.

L

Last-use analysis. The compiler phase where, for each variable, the exact point of its last use is identified. Perceus uses this analysis to insert `drop`s in the right place.

Linked (in actors). Two actors linked with `Link.link(pid)` find out about each other's termination: if one falls, the other receives `Cancel.raise()`. Chapter 14.

M

Mailbox. The message queue associated with an actor. The actor processes messages in FIFO order. The mailbox's **policy** decides what to do when it fills up (unbounded, drop-oldest, drop-newest, block-sender).

Match. Pattern-matching expression. Covers every constructor of a sum type (exhaustive) and lets you extract components from records and lists. Chapter 5.

Monitor (in actors). An actor monitoring another receives a `MonitorDown` message when the monitored actor terminates, without coupling its own life to the observed's. Chapter 14.

MVS (minimum-version selection). Dependency resolution algorithm used by the package manager. When one project declares `manutara@v0.1.0` and another declares `manutara@v0.2.0`, MVS picks the **maximum** of the declared versions. Chapter 8.

N

Nothing. See *Bottom type*.

Nursery. A lexical scope that contains child fibers and guarantees none survives the block. Built with `nursery { n -> ... }`, which is sugar over `handle ... with Spawn as n { ... }`. Chapter 13.

O

Operation. A named declaration inside an `effect`: a name, parameters, return type. Operations are invoked with `Effect.op(args)`.

P

Pattern matching. The `match` construct and patterns in `let`. Lets you decompose structured values (sum types, records, lists) into components. Chapter 5.

Perceus. The static reference-counting system kaikai uses to free memory without GC or borrow checker. Invented by Reinking, Xie, de Moura, and Leijen (PLDI 2021). Appendix B and chapter 13 §13.2.

Pid (process id). Typed handle of an actor: `Pid[Msg]`. Identifies a mailbox and also guarantees that only messages of type `Msg` can be sent to it.

Pipe. Operators `|>`, `|`, `||`, `|?`. Chain transformations. Chapter 6.

Polymorphism. A function's ability to operate over multiple types. In kaikai it appears as generics (`fn map[a, b](xs: [a], f: (a) -> b) : [b]`) and as polymorphic rows (`/ e` where `e` is a row variable).

Protocol. An interface declared with `protocol`, implemented by types via `impl Protocol for T`. It's single-dispatch (resolves on a single type). Chapter 9.

Pure. A function is pure if it produces no effects (its row is empty). Pure functions are easy to test, parallelize, and reason about.

R

Ref. A mutable cell with arbitrary lifetime, accessed via `Mutable.ref_make` / `Mutable.ref_get` / `Mutable.ref_set`. Chapter 12 §12.7.

Refinement type. A type with a restricting predicate: `Int where self >= 0`. Chapter 11.

Resume. See *Continuation*.

Reuse in place. Perceus optimization: when a unique value is about to be freed and another of the same shape is about to be created, the same memory is reused without touching counters. Appendix B.

Row variable. A polymorphic variable representing "the rest of the effects" in a signature: `fn map[A, B, e](xs: [A], f: (A) -> B / e) : [B] / e`.

S

Self-hosting. The state where a compiler is written in the same language it compiles. The kaikai compiler `kaic2` is written in kaikai. Appendix A.

Span. The byte range in the source file where a construct lives. Compiler messages use spans to point to where an error occurred.

Spawn. Create a new fiber or actor. Operation of the `Spawn` effect.

Stage 0/1/2. The three compilers that make up kaikai's bootstrap. Stage 0 in C, stage 1 in kaikai-minimal, stage 2 in full kaikai. Appendix A.

State[T]. Stdlib effect for carrying encapsulated mutable state. The sugared form `var name = init` desugars to `handle ... with State[T](init)` (chapter 12 §12.7).

Stdout, Stderr, Stdin. Standard output, standard error, standard input. In kaikai these live under the `Console` effect (for the first two) and `Stdin` (for the last).

Sum type (algebraic data type). A type with several constructors, each carrying different data. The construct `type Shape = Circle(Real) | Square(Real)`. Chapter 5.

T

Tail call. A function call in "last thing the current function does" position. The compiler compiles it to a jump, not a stack-pushing call, so tail recursion runs without consuming stack memory. Chapter 6.

Top-down design. Design style starting from the signatures of top-level functions, leaving holes in the bodies, and filling the inner pieces afterward. Chapter 15.

Trap exit. An actor's ability to **not** propagate cancellation automatically when a sibling fiber falls. Enabled with `fiber_set_trap_exit(true)`; the actor receives an informational message instead of `Cancel.raise()`. Chapter 14.

U

Unit of measure. A symbolic unit declared with `unit` that annotates a numeric value (`Real<USD>`, `Int<Seconds>`). The type system rejects operations that mix incompatible units. Chapter 10.

V

Var. Construct that declares a local mutable cell. Syntactic sugar over `State[T]`. Chapter 12 §12.7.

Y

Yield. The act of a fiber handing control back to the scheduler so another fiber can run. `fiber_yield()` does it explicitly; IO operations and `Spawn.await` do it implicitly.

Appendix F · Going further

The book covered kaikai but barely touched the family of ideas it's built on. If you want to dig in, here's a short list of sources worth reading, grouped by topic. Not meant to be exhaustive — meant to be useful.

F.1 Algebraic effects

The piece of the language that most rewards outside reading is algebraic effects. The literature is accessible and the originals are worth going to directly.

- **Plotkin, Pretnar, *"Handlers of Algebraic Effects"*** (ESOP 2009). The paper that introduced handlers as we know them today. Technical but short.
- **Bauer, Pretnar, *"Programming with Algebraic Effects and Handlers"*** (J. Logical and Algebraic Methods in Programming, 2015). More pedagogical than the previous one; good as a second read.
- **Koka language** (Daan Leijen, Microsoft Research). `koka-lang.github.io`. Probably the best implementation of algebraic effects today. Much of kaikai's syntactic inspiration comes from here.
- **Effekt language.** `effekt-lang.org`. Has an effect system based on capabilities. Comparable to Koka with a different aesthetic.
- **Eduardo Díaz, *Revelaciones*** (lnds.net, 2015, in Spanish, `https://lnds.net/blog/lnds/2015/10/01/revelaciones/`). The historical preface to kaikai's path: category theory and monads as the bridge from classical functional programming toward algebraic effects. Mentioned in the prologue.

F.2 Perceus and reference counting

- **Reinking, Xie, de Moura, Leijen, *"Perceus: Garbage Free Reference Counting with Reuse"*** (PLDI 2021). The paper that invented the system kaikai uses to free memory. It's readable.
- **Lorenz, Leijen, *"Reference Counting with Frame Limited Reuse"*** (ICFP 2023). Extension that improves reuse when the shape doesn't quite match.
- **Lean 4** (`leanprover.github.io`). Proof system that uses a Perceus variant in its runtime. More complex than Koka but also well-documented.

F.3 Actor model and BEAM

- **Joe Armstrong, *Programming Erlang*** (Pragmatic Bookshelf, 2007/2013). The canonical Erlang book written by its creator. Covers the "let it crash" philosophy more deeply than any recent introduction.
- **Saša Jurić, *Elixir in Action*** (Manning, 2019). The actor model explained for modern programmers, with Elixir code. The OTP part is excellent.
- **Cesarini, Vinoski, *Designing for Scalability with Erlang/OTP*** (O'Reilly, 2016). For when you want to think about serious distributed systems.

F.4 Structured concurrency

- **Nathaniel J. Smith, *"Notes on structured concurrency, or: Go statement considered harmful"*** (vorpus.org, 2018). The seminal essay on structured concurrency. Required reading if you'll write any concurrent code in any modern language.
- **Bob Nystrom, *"What Color is Your Function?"*** (journal.stuffwithstuff.com, 2015). The essay on the coloring problem that motivated kaikai's bet on effects instead of `async` / `await`. Short, funny, with bite.
- **Trio (Python)**, **Kotlin coroutines**, **Swift structured concurrency**, **OCaml 5 Eio**. Four concrete implementations of the same model. Comparing how each one expresses it helps the idea stick.

F.5 Language design

- **Fred Brooks, *"No Silver Bullet"*** (IEEE Computer, 1986). The classic essay on essential vs accidental complexity in software. Still relevant forty years later because it was right.
- **Rich Hickey, *"Simple Made Easy"*** (Strange Loop, 2011, video on infoq.com). An hour of Rich Hickey distinguishing *simple* from *easy*. Changes how you evaluate APIs and languages.
- **Marlow, Goldsmith et al., *"Out of the Tar Pit"*** (paper 2006, accessible online). Diagnosis of why software grows complicated and proposal for how to avoid it. Very influential in modern functional thinking.
- **Steele, Sussman, *"Lambda: The Ultimate Imperative"*** (MIT AI Memo, 1976). One of the founding papers showing that functional programming with closures and recursion covers everything imperative languages do.

F.6 Type systems

- **Benjamin Pierce, *Types and Programming Languages*** (MIT Press, 2002). The reference book for type systems. Not read at one sitting, but consulted chapter by chapter.
- **Robert Harper, *Practical Foundations for Programming Languages*** (Cambridge University Press, 2016, 2nd ed.). More modern than Pierce. Covers effects, row polymorphism, things Pierce's didn't have.

- **Pierce et al., *Software Foundations*** (online volumes at `softwarefoundations.cis.upenn.edu`). Interactive course on type systems verified in Coq. For those who want full rigour.

F.7 Contracts and design by contract

- **Bertrand Meyer, *Object-Oriented Software Construction*** (Prentice Hall, 1997, 2nd ed.). The original Eiffel and design-by-contract book. Although the OO context isn't kaikai's, the arguments for why contracts matter are the same.
- **John Barnes, *Programming in Ada 2012 with a Preview of Ada 2022*** (Cambridge University Press, 2014). Ada is the other major exponent of contracts in an industrial language. To see what they look like in production.

F.8 Community and code

- **Official repository**: `github.com/kaikailang-org/kaikai`. The compiler, the stdlib, the design documents. Bug reports and proposals are welcome as issues.
- **This book**: `github.com/kaikailang-org/kaikai-book`. PRs with fixes are welcome. The book is in Spanish and English; both editions are maintained in parallel.
- **Author's blog**: `lnds.net`. Where ideas appear before they make it into the book, with less discipline and more personal judgment.

F.9 Closing

If the book left you wanting to try something, the best way to learn is to write code. Take any program you've written in another language (any language) and try porting it to kaikai. You'll run into things the book didn't cover, you'll open issues, you'll learn what no book can teach.

The compiler is alive, the language is evolving, and the community is small but attentive. There's room for more.