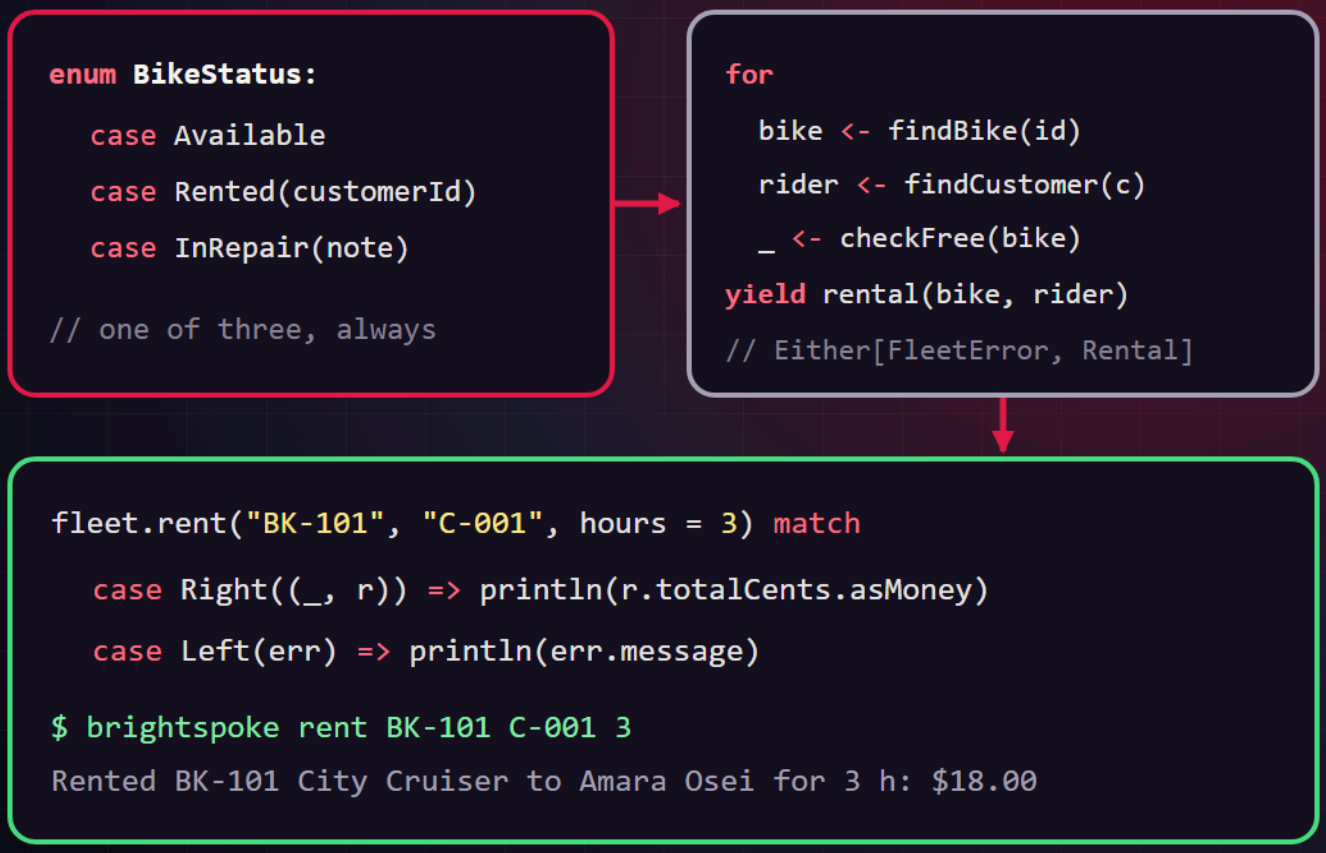


CASE CLASSES · ENUMS · TRAITS · GIVENS · TYPE CLASSES

FROM YOUR FIRST LINE TO REAL PROJECTS

Scala Programming from Zero

A complete, practical guide to functional and object-oriented programming

Clear explanations, a running project, and compiler errors treated as teachers — written for Scala 3

SBT · SCALA CLI · MUNIT · UPICKLE · CASK

THENEXBOOKS

Scala Programming from Zero

A Complete, Practical Guide to Functional and Object-Oriented Programming with Scala 3

First Edition

THENEXBOOKS

Scala Programming from Zero
A Complete, Practical Guide to Functional and Object-Oriented Programming with Scala 3

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Scala and its ecosystem evolve steadily; compiler messages, library APIs, and tool output may differ slightly in the versions you use.

Scala is a trademark of the École Polytechnique Fédérale de Lausanne (EPFL). Java is a trademark of Oracle and/or its affiliates. Their use in this book is for identification only and does not imply endorsement or affiliation. Other product names are trademarks of their respective owners.

Brightspoke, its staff, customers, partner shops, and bikes are fictional. Any resemblance to real companies or people is coincidental.

Preface

Scala sits at an interesting crossroads. It is an object-oriented language, with classes, objects, and traits that will feel familiar if you know Java, C#, or Kotlin. It is also a functional language, where functions are values, data is immutable by default, and almost everything is an expression that produces a result. Many languages have borrowed ideas from one side or the other; Scala was designed from the start to combine both, and Scala 3 makes that combination cleaner and friendlier than ever.

That combination is also why people sometimes find Scala intimidating. You can write the same program in several styles, and older tutorials are full of advanced features before they have explained the basics. This book takes the opposite approach. We start from zero — installing Scala and writing your first program — and add one idea at a time, always explaining *why* Scala works the way it does. When the compiler rejects your code, we read the message together and treat it as a teacher. By the end you will be comfortable with case classes, enums, pattern matching, traits, collections, `Option` and `Either`, for-comprehensions, generics, givens and extension methods, type classes, testing, futures, and JSON — and you will have built a complete, tested command-line application and a small JSON API.

Who This Book Is For

- **Developers new to Scala** who know another language — Java, Python, JavaScript or TypeScript, Kotlin, or C# — and want to learn Scala 3 properly.
- **Motivated beginners** with a little programming experience who want a language that teaches clear, careful thinking about data.
- **Java developers** who want a more expressive language on the JVM they already know.
- **Anyone curious about functional programming** who wants to learn it through practical code rather than theory.

What You Will Learn

- Installing Scala with Coursier and Scala CLI, using the REPL, editors with Metals or IntelliJ, and building projects with sbt.
- Values, types, expressions, control flow, functions, classes, objects, and companion objects.
- Modeling data with case classes and enums, and taking it apart with pattern matching.
- Traits and composition, packages and imports, and organizing a growing project.
- Functional programming: higher-order functions, the collections library, `Option`, `Either`, `Try`, for-comprehensions, recursion, and folds.
- Generics and variance, givens and extension methods, type classes, opaque types, and union and intersection types.

- Testing with MUnit, concurrency with futures, files and JSON with os-lib and uPickle, command-line apps, and Java interoperability.
- A first look at effect systems (Cats Effect and ZIO), idiomatic style, performance basics, and publishing libraries.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) installs Scala and covers values, types, expressions, functions, classes, and objects. **Part II — Core Language** (Chapters 6–10) covers case classes, enums, pattern matching, traits, and sbt projects. **Part III — Functional Programming** (Chapters 11–15) covers functions as values, collections, error handling without exceptions, for-comprehensions, and recursion. **Part IV — Types and Abstractions** (Chapters 16–20) covers generics, contextual abstractions, type classes, Scala 3's newer type features, and testing. **Part V — Practical Scala** (Chapters 21–25) covers futures, files and JSON, command-line apps, Java interop, and an introduction to effect systems. **Part VI — Projects and Next Steps** (Chapters 26–28) brings everything together in two projects and a guide to writing idiomatic Scala. The appendices provide a syntax cheat sheet, a tooling and library reference, a field guide to compiler errors, and a glossary.

Each chapter includes practice exercises and ends with a **Key Takeaways** box for quick review.

The Running Example: Brightspoke

Throughout the book we write software for **Brightspoke**, a small neighborhood bike rental and repair shop. Maya owns the shop, Jonas is the mechanic who asks the practical questions, and Theo — Maya's nephew, learning Scala alongside you — helps build the tools. Chapter by chapter, you grow `brightspoke`, a command-line fleet manager that tracks bikes, customers, prices, and rentals: first as a few values, then case classes and enums, then an immutable `Fleet` with error handling in `Either`, then saved to JSON and CSV, and finally as a complete, tested CLI with a small JSON API on top.

Versions and Conventions

Examples use Scala 3 (the 3.3 long-term-support line or newer) and its indentation-based syntax. Library versions in build files are illustrative; use the current versions when you add them. Compiler output is trimmed to fit the page and may differ slightly from what you see.

> **TIP**
> A recommended practice or a technique that saves you time.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake, a compiler error you are likely to meet, or a habit to avoid.

File captions such as *brightspoke/src/main/scala/brightspoke/Fleet.scala* tell you where a snippet belongs.

Contents

Preface

PART I — GETTING STARTED

1 Why Scala? What It Is, Where It Is Used, and Who It Is For

2 Installing Scala: Scala CLI, sbt, the REPL, and Your First Program

3 Values, Variables, and Basic Types

4 Expressions, Control Flow, and Functions

5 Classes, Objects, and Companion Objects

PART II — CORE LANGUAGE

6 Case Classes and Immutable Data

7 Enums and Algebraic Data Types

8 Pattern Matching in Depth

9 Traits, Mixins, and Composition

10 Packages, Imports, and Projects with sbt

PART III — FUNCTIONAL PROGRAMMING

11 Functions as Values: Lambdas, Higher-Order Functions, and Currying

12 Collections: List, Vector, Set, Map, and the Collections API

13 Option, Either, and Try: Handling Absence and Errors Without Exceptions

14 For-Comprehensions: map, flatMap, and Sequencing Computations

15 Recursion, Folding, and Thinking Immutably

PART IV — TYPES AND ABSTRACTIONS

16 Generics, Variance, and Type Bounds

17 Contextual Abstractions: given, using, and Extension Methods

18 Type Classes in Scala 3

19 Opaque Types, Union and Intersection Types, and Other Scala 3 Type Features

20 Testing with MUnit and Property-Based Testing

PART V — PRACTICAL SCALA

21 Concurrency with Futures

22 Files, JSON, and Persistence with os-lib and uPickle

23 Building Command-Line Applications

24 Working with Java: Interoperability in Both Directions

25 An Introduction to Effect Systems: Cats Effect and ZIO

PART VI — PROJECTS AND NEXT STEPS

26 Project: The Brightspoke Fleet CLI End to End

27 Project: A Small JSON API with Cask

28 Idiomatic Scala, Performance Basics, Publishing Libraries, and Where to Go Next

APPENDICES

A Scala 3 Syntax Cheat Sheet

B sbt, Scala CLI, and Libraries Reference

C Compiler Error Field Guide

D Glossary

PART I

Getting Started

What Scala is and why it blends object-oriented and functional programming, installing the tools, your first programs, and the building blocks every Scala program uses: values, types, expressions, functions, classes, and objects.

1 · Why Scala?

2 · Installing Scala

3 · Values, Variables, and Types

4 · Expressions and Functions

5 · Classes and Objects

CHAPTER 1 · PART I — GETTING STARTED

Why Scala? What It Is, Where It Is Used, and Who It Is For

Before we install anything, we take a calm look at what Scala is, why it blends object-oriented and functional programming, where it fits in the software world, and what you will have built by the last page: a working command-line fleet manager for a small bike rental shop called Brightspoke.

A Language That Scales With You

The name Scala comes from "scalable language." The idea behind the name is simple and still a good summary of the language today: Scala should feel comfortable for a ten-line script and still hold up when a program grows to hundreds of files maintained by a team. The same small set of ideas — values, expressions, types, functions, and objects — carries you from the first line you type into the REPL all the way to a structured application with tests, JSON storage, and a command-line interface.

Scala was created by Martin Odersky and his group at EPFL in Switzerland, and its first public release appeared in 2004. Scala 3, a major redesign that cleaned up the syntax and added several powerful features, was released in 2021. This book teaches **Scala 3** from the first page. You will occasionally see a short aside that says "you may see this in older code," because a lot of Scala 2 code still exists in the world, but everything you write while following along is modern Scala 3.

Here is a first taste, so you have something concrete in mind while we talk about ideas. Do not worry about every detail yet; each piece gets its own careful explanation in the coming chapters.

taste.scala

```
val hourlyRateCents = 900
val hours = 8
val DailyCapHours = 6

val billedHours = math.min(hours, DailyCapHours)
val totalCents = hourlyRateCents * billedHours

println(s"Trail Runner for $hours h costs $totalCents cents")
// prints: Trail Runner for 8 h costs 5400 cents
```

Even this tiny snippet shows several things that make Scala pleasant. There are no type declarations cluttering the code, yet every value has a precise type: the compiler works out that `hourlyRateCents` is an `Int` and refuses to let you treat it as text by accident. The word `val` declares a value that never changes. The `s"..."` string drops values straight into text. And the calculation reads almost like the business rule it implements: bill the hourly rate for the number of hours, but never for more than six.

What Kind of Language Is Scala?

People often describe Scala as a "hybrid" language, and that is accurate, but the word can sound like a compromise. It is better to think of Scala as a language that took two strong traditions seriously and found a way for them to work together.

Object-oriented, all the way down

In Scala, every value is an object. The number `42` is an object with methods; you can write `42.max(7)` and get `42` back. Even the familiar `1 + 2` is really a method call, `1.+(2)`, written in a friendlier way. Scala has classes, objects, inheritance, and a flexible building block called a **trait** that lets you mix behavior into classes. If you know Java, C#, or Kotlin, most of these concepts will feel familiar, and Chapters 5 and 9 cover them in detail.

Functional, by default

At the same time, Scala treats functions as ordinary values. You can store a function in a variable, pass it to another function, and return it from a function. The standard library collections — `List`, `Vector`, `Map`, and friends — are immutable by default: when you "add" an element, you get a *new* collection, and the original stays exactly as it was. Errors are often modeled as ordinary values (`Option` and `Either`) rather than exceptions that fly across your program. These habits come from functional programming, and Part III of this book is devoted to them.

Expressions instead of statements

One idea ties the two traditions together: in Scala almost everything is an **expression** that produces a value. An `if` produces a value. A `match` produces a value. A block of code in braces or indentation produces the value of its last line. This sounds like a small detail, but it changes how you write code. Instead of declaring a variable and then filling it in from several branches, you simply write the branches and let the result flow out.

```
val label =
  if billedHours < hours then "capped at the daily maximum"
  else "full hours billed"
```

Chapter 4 explores this style properly. For now, notice that `label` is a `val`: it is assigned once, from an expression, and never changes. You will see this pattern everywhere in idiomatic Scala.

Strong, static types with inference

Scala is statically typed: the compiler checks types before your program runs, so a whole category of mistakes is caught while you are still editing. Unlike some statically typed languages, though, Scala rarely asks you to repeat yourself. Type **inference** means the compiler works out most types on its own. You write types where they help a human reader — on function parameters and public results — and leave them out where they would only add noise.

> **NOTE**
>
> Scala's type system is famous for being powerful, and you may have heard that it is complicated. The parts you need for everyday programming are small and friendly, and this book introduces the advanced features (given instances, type classes, opaque types, variance) gradually in Part IV, only after you are comfortable with the basics. You can write useful Scala long before you need any of them.

Where Scala Runs

Scala's main home is the **Java Virtual Machine** (JVM). When you compile Scala code, the compiler produces JVM bytecode, the same kind of bytecode that Java produces. That has three practical consequences:

- **Mature runtime.** Your program benefits from decades of JVM work on garbage collection, just-in-time compilation, monitoring, and profiling tools.
- **Java libraries are available.** Any Java library can be called directly from Scala. If you need to read a date, talk to a database driver, or parse a spreadsheet, a Java library probably exists, and you can use it without wrappers. Chapter 24 shows interoperability in both directions.
- **Familiar deployment.** A Scala application is deployed like any other JVM application, as a JAR file or a packaged launcher.

The JVM is not the only target. **Scala.js** compiles Scala to JavaScript, so you can run Scala in a browser or on Node.js. **Scala Native** compiles Scala ahead of time to native machine code, which is useful for command-line tools that should start instantly. This book focuses on the JVM, because that is where most Scala code runs and where the tooling is most complete, but almost everything you learn about the language itself applies to all three targets.

Where Scala Is Used

We will not throw adoption numbers at you; such figures change quickly and are hard to verify. Here are a few areas where Scala has a long, well-known track record:

- **Backend services.** Scala is used to write web services and APIs, often where correctness and concurrency matter. Libraries for HTTP, JSON, and databases are widely available. Chapter 27 builds a small JSON API.
- **Data engineering.** Apache Spark, a widely used engine for large-scale data processing, is written in Scala, and Scala is one of its primary APIs. This book is not a Spark book, but the collection operations you learn in Chapters 11 and 12 (`map`, `filter`, `groupBy`, folding) are the same style of thinking Spark uses.

- **Concurrent and distributed systems.** Scala has a strong ecosystem for concurrency, from the standard library's `Future` (Chapter 21) to effect systems such as Cats Effect and ZIO (introduced in Chapter 25).
- **Tools and compilers.** Scala's expressive types and pattern matching make it a comfortable language for writing parsers, interpreters, and developer tooling. The Scala 3 compiler itself is written in Scala.

The common thread is *modeling*. Scala is very good at describing a domain precisely — "a bike is either available, rented by a specific customer, or in repair with a note" — and then letting the compiler check that every part of the program respects that description. That strength is exactly what we lean on throughout this book.

Who This Book Is For

This book assumes you are new to Scala. It helps if you have written a little code in some language before, so that words such as "variable," "loop," and "function" are not entirely new, but every Scala concept is explained from scratch, with small examples you can type and run.

You may be coming from a specific background. Throughout the book we include short asides for readers who already know another language. They are brief on purpose: they point out a difference or a trap, and then the main text continues for everyone.

- **If you come from Java**, you already know the JVM, classes, and generics. Scala will feel like Java with far less ceremony, plus pattern matching, case classes, and a much richer type system.
- **If you come from Python**, you will appreciate the concise syntax and significant indentation of Scala 3. The big change is static types: the compiler checks your work before the program runs.
- **If you come from JavaScript or TypeScript**, union types, structural thinking, and arrow-style functions will feel familiar. Scala's collections and pattern matching go further than you are used to.
- **If you come from Kotlin or C#**, many ideas carry over directly: data classes become case classes, extension methods exist, and nullable types become `Option`.

If you are learning Scala as your first serious language, that works too. Take your time with Part I, type every example, and do the Practice exercises at the end of each chapter. They are short, and their solutions are included.

Meet Brightspoke

Learning a language through disconnected examples gets dull quickly, and it hides the most important skill: putting pieces together into a program that does something real. So instead of a different toy in every chapter, we follow one small business from start to finish.

Brightspoke is a neighborhood bike rental and repair shop. It is run by **Maya**, the owner, who knows exactly what she wants from her software and has no patience for tools that lose track of a rented bike. **Jonas** is the shop's mechanic; he spends his days fixing brakes and adjusting gears, and he asks the practical questions — "what happens if someone types the wrong bike number?" **Theo** is Maya's nephew, who is learning Scala alongside you. When Theo gets confused, you will probably be confused at the same spot, and his questions give us a natural place to slow down.

For years Maya tracked her fleet in a paper notebook. The notebook has served her well, but it cannot add up the day's revenue, it does not warn her when she tries to rent out a bike that is waiting for brake pads, and it lives on one counter in one shop. She wants a small command-line program, which the shop calls `brightspoke`, to take over the bookkeeping. Over the course of this book, you will build it.

The fleet

Brightspoke starts with four bikes. Each bike has an id (a string such as `BK-101`), a name, a kind, an hourly rate, and a current status. This table is the canonical data for the whole book; you will see these exact values in examples, tests, and output from Chapter 3 to Chapter 28.

Id	Name	Kind	Hourly rate (cents)	Status at start
BK-101	City Cruiser	City	600	Available
BK-102	Trail Runner	Mountain	900	Available
BK-103	Swift E	Electric	1500	InRepair("brake pads")
BK-104	Little Rider	Kids	400	Rented("C-002")

The shop also has three regular customers: `C-001` Amara Osei (amara@example.com), `C-002` Leo Brandt (leo@example.com), and `C-003` Priya Nair (priya@example.com). As you can see from the table, Leo already has the Little Rider out for his daughter. All of these people and addresses are fictional.

Every rental gets an id of its own: `R-0001` for the first rental, `R-0002` for the second, and so on, always padded to four digits.

The business rules

Maya's rules are short, and they give us plenty to program:

1. **Money is counted in cents.** A rate of `600` means $6.00 an hour. We never store money as a floating-point number, for reasons Chapter 3 explains with a concrete example.
2. **A rental lasts from 1 to 24 hours.** Anything outside that range is rejected.
3. **There is a daily cap.** A customer never pays for more than 6 hours. Renting the City Cruiser for 3 hours costs 600 × 3 = 1800 cents, or $18.00. Renting the Trail Runner for 8 hours costs 900 × 6 = 5400 cents, or $54.00, because the last two hours are free.

4. **Only an available bike can be rented.** A bike that is in repair or already rented cannot go out again.
5. **Returning a rented bike makes it available.** Returning a bike that is not rented is an error.

Jonas read the list and asked the first practical question: "What does the program say when someone tries to rent Swift E while I'm still waiting for brake pads?" It is a fair question, and the answer is part of the goal below.

The Goal: A Target CLI Session

Here is what the finished `brightspoke` tool does. This exact session is our roadmap. In Chapter 26 we will run the completed program and get precisely this output, character for character.

terminal

```
$ brightspoke list
ID      NAME          KIND       RATE/H  STATUS
BK-101  City Cruiser  City        $6.00  available
BK-102  Trail Runner  Mountain    $9.00  available
BK-103  Swift E       Electric   $15.00  in repair (brake pads)
BK-104  Little Rider  Kids        $4.00  rented by C-002
$ brightspoke rent BK-101 C-001 3
Rented BK-101 City Cruiser to Amara Osei for 3 h: $18.00 (R-0001)
$ brightspoke rent BK-103 C-003 2
error: BK-103 is not available (in repair)
$ brightspoke report
Rentals: 1, revenue $18.00
```

Let us read the session line by line, because each line quietly depends on several language features.

- **`list`** prints the fleet as an aligned table, sorted by id. Producing it requires a model of bikes (case classes in Chapter 6, enums in Chapter 7), a way to turn a status into a readable label (pattern matching in Chapter 8), collections and sorting (Chapters 12 and 17), and formatted strings (Chapter 3).
- **`rent BK-101 C-001 3`** rents the City Cruiser to Amara for three hours. The program looks up the bike and the customer, checks the hours, checks that the bike is available, computes the price with the daily cap, assigns the next rental id, and records the rental. That needs `Option` and `Either` (Chapter 13) and for-comprehensions (Chapter 14).
- **`rent BK-103 C-003 2`** is Jonas's question answered: the program refuses politely with a precise message, and it does not crash. Errors are values in our design, not exceptions.
- **`report`** summarizes the day: one rental and $18.00 of revenue. Adding up rentals uses folding (Chapter 15).

Between those commands, the fleet is saved to a JSON file (Chapter 22) and the command-line arguments are parsed into a small `Command` type (Chapter 23). Along the way we write a test suite with MUnit (Chapter 20) so we can change the code with confidence, and in Chapter 27 we put the same

fleet behind a small HTTP API.

> **TIP**
> Keep this target session handy. When a chapter introduces a new feature, ask yourself which line of the session it helps produce. Connecting each concept to a visible result is the fastest way to make it stick.

How the program grows

We do not build the final design in one leap. The program grows the way real programs do: a little at a time, with each step working before the next begins. Here is the path in outline.

```
Ch 2   Brightspoke.scala: one file, prints a banner
Ch 3-4 plain vals for bikes, formatCents, rentalCost
Ch 5   object Pricing collects the pricing rules
Ch 6-7 case class Bike, enums BikeKind and BikeStatus
Ch 8   statusLabel with pattern matching
Ch 10  an sbt project with several files
Ch 12  Fleet holds Maps of bikes and customers
Ch 13  FleetError and Either: rent, returnBike
Ch 14  rent rewritten as a for-comprehension
Ch 20  MUnit tests for the fleet
Ch 22  fleet.json and rentals.csv on disk
Ch 23  parsing commands from the terminal
Ch 26  the full CLI, matching the target session
Ch 27  a small JSON API with Cask
```

Some early versions are deliberately simpler than the final ones. In Chapter 6, for example, a bike's kind is temporarily a plain `String`; in Chapter 7 we replace it with a proper enum and see how the compiler helps us find every place that needs updating. Those rewrites are not wasted effort. Watching a design improve is one of the most useful things a book can show you, and it is how you will work on your own projects.

A Peek at the Final Model

To make the destination concrete, here is a small part of the model you will write in Part II. You are not expected to understand all of it now; read it the way you would look at a map before a trip.

`brightspoke/src/main/scala/brightspoke/Model.scala (preview)`

```
enum BikeKind:
  case City, Mountain, Electric, Kids

enum BikeStatus:
  case Available
  case Rented(customerId: String)
  case InRepair(note: String)

final case class Bike(
```

```
    id: String,
    name: String,
    kind: BikeKind,
    hourlyRateCents: Int,
    status: BikeStatus
)
```

Read it aloud and it almost sounds like Maya describing her shop: a bike's kind is city, mountain, electric, or kids; a bike's status is available, rented by some customer, or in repair with a note. That closeness between the code and the way people talk about the business is one of Scala's quiet superpowers. When Maya later says "we've started renting cargo bikes," you add one case to `BikeKind`, and the compiler points you at every place in the program that must handle the new kind.

Theo looked at the code and asked, "Why is there no `available: Boolean` field? Wouldn't that be simpler?" It would be simpler to type, but it would allow nonsense: a bike that is "not available" but has no customer and no repair note. With an enum, every bike is in exactly one meaningful state, and each state carries exactly the data it needs. Chapter 7 tells this story in full.

Reading the Compiler Error: A First Look

Throughout this book we treat compiler errors as teachers, not as punishments. The Scala 3 compiler produces some of the clearest messages of any mainstream language, and learning to read them early will save you hours. Here is a typical one. Suppose Theo tried to store a price as text:

```
val total: Int = "18.00"
```

The compiler answers:

```
-- [E007] Type Mismatch Error: Brightspoke.scala:12:19 -------------------
12 |  val total: Int = "18.00"
   |                   ^^^^^^^
   |                   Found:    ("18.00" : String)
   |                   Required: Int
```

Every Scala 3 error has the same anatomy, so it pays to learn it once:

1. **The header line** names the error kind and, often, a code in brackets — `[E007]` means "type mismatch." Then comes the file, the line (12), and the column (19).
2. **The source line** is reprinted with its line number, so you do not have to go looking for it.
3. **The carets** (`^^^^^^^`) underline exactly the part of the line the compiler is unhappy about.
4. **The explanation** says what was found and what was required. Here the compiler found the string `"18.00"` — notice that it even shows the precise literal — but the declared type demands an `Int`.

The fix is obvious once the message is read calmly: store `1800` (cents) instead. The message also hints at one of Scala's strengths. The mistake was caught before the program ever ran, before any customer ever saw a wrong price.

> **WARNING**
>
> When a file has several errors, fix the *first* one first and compile again. Later errors are often side effects of the first, and they may disappear on their own. Trying to fix them all at once from the bottom up is a common way to make things worse.

How to Use This Book

The book has six parts and four appendices:

Part	Chapters	What you gain
I — Getting Started	1–5	Tools, values, control flow, objects
II — Core Language	6–10	Data modeling, matching, projects
III — Functional Programming	11–15	Functions, collections, errors
IV — Types and Abstractions	16–20	Generics, givens, type classes, tests
V — Practical Scala	21–25	Concurrency, files, CLI, Java
VI — Projects and Next Steps	26–28	The full CLI, an API, idioms

The appendices collect a syntax cheat sheet, a reference for sbt, Scala CLI, and libraries, a field guide to common compiler errors, and a glossary. Use them whenever you need a quick reminder.

Each chapter follows the same rhythm. It explains a concept with small examples, applies the concept to Brightspoke, walks through at least one realistic compiler error where that makes sense, lists Common Mistakes, offers four or five Practice exercises with brief solutions, and ends with Key Takeaways. The chapters build on each other, so we recommend reading Parts I through III in order. After that, the chapters in Parts IV and V are more independent, although the Brightspoke code still grows from chapter to chapter.

Type the code

It is tempting to read code and nod along. Resist the temptation. Typing examples by hand, making small changes, and watching what the compiler says is how the syntax moves from the page into your fingers. When an example does not compile, that is not a failure; it is a free lesson with a precise explanation attached.

Code conventions in this book

- Code blocks that belong to a file start with a caption showing the file name, such as `Brightspoke.scala` or `brightspoke/src/main/scala/brightspoke/Fleet.scala`.
- Output appears in comments (`// prints: ...`) or in a separate block labeled `terminal`, where lines beginning with `$` are commands you type.
- We use Scala 3's indentation-based syntax by default. Braces still work, and we mention them where they help readers of older code.
- Version numbers for tools and libraries are illustrative. Scala moves steadily, so always check for the current release when you set up a project. We target Scala 3.3 LTS or newer.

Scala's Design in Five Ideas

Before moving on, here are five ideas that explain most of Scala's personality. You will meet each one many times.

1. **Prefer immutability.** Values declared with `val` do not change, and the standard collections are immutable. Programs built from immutable data are easier to reason about, easier to test, and safe to share between threads. Brightspoke's `Fleet` follows this rule: renting a bike returns a *new* fleet rather than modifying the old one.
2. **Everything is an expression.** Conditionals, matches, and blocks produce values. You compose small expressions into larger ones instead of stringing statements together.
3. **Make illegal states unrepresentable.** Use types — enums, case classes, `Option`, `Either` — so that impossible situations cannot even be written down. A bike cannot be "rented by nobody."
4. **Let the compiler help.** Static types plus inference give you safety without noise. When the model changes, the compiler finds the code that must change too.
5. **Mix paradigms with purpose.** Use objects to organize code into modules and give names to concepts. Use functions to transform data. Neither style is "the right one"; Scala lets you choose the best tool for each job.

Maya, who does not program, summarized it in her own words after Theo explained it over coffee: "So the program refuses to write nonsense in the notebook in the first place." That is not a bad summary of statically typed functional programming.

> **NOTE**
>
> Scala is sometimes said to have a steep learning curve. Much of that reputation comes from advanced libraries and from Scala 2's heavier syntax. Scala 3 smoothed many rough edges, and this book introduces ideas one at a time, always tied to a concrete problem. If something feels abstract, look for the Brightspoke example that follows; it is never far away.

Common Mistakes

- **Learning from Scala 2 tutorials without realizing it.** Older material uses `implicit`, braces everywhere, and `sealed trait` hierarchies where Scala 3 uses `given`, indentation, and `enum`. That code usually still works, but mixing styles at the start is confusing. When in doubt, check the date and the Scala version of a tutorial.
- **Treating Scala as "Java with shorter syntax."** You can write Scala that way, but you miss most of the benefits. Lean into immutable values, expressions, and pattern matching from the start.
- **Jumping straight to advanced libraries.** Effect systems and type-level programming are fascinating, but they build on the core language. Master Parts I to III first; Chapter 25 gives you a gentle bridge when you are ready.
- **Using floating-point numbers for money.** It looks harmless and produces wrong totals sooner or later. Brightspoke uses `Int` cents everywhere, and Chapter 3 shows why.
- **Ignoring compiler messages.** Reading them carefully, top to bottom, is a skill worth practicing from day one.

Practice

1. **Price check.** Using the business rules, compute by hand the cost in cents of renting Swift E (once it is repaired) for 4 hours, and the Little Rider for 10 hours.
   Solution: Swift E: 1500 × 4 = 6000 cents ($60.00). Little Rider: 400 × min(10, 6) = 400 × 6 = 2400 cents ($24.00).
2. **Read the session.** In the target session, why does the report say `Rentals: 1` even though two `rent` commands were typed?
   Solution: The second command failed because BK-103 is in repair. A failed rental does not create a rental record, so only `R-0001` exists and revenue stays at $18.00.
3. **Find the next id.** If Leo had a recorded rental `R-0001` before Amara rented the City Cruiser, what would Amara's rental id be?
   Solution: `R-0002`. Rental ids are sequential and padded to four digits.
4. **Spot the illegal state.** Suppose a bike were modeled with two fields, `available: Boolean` and `customerId: String`. Describe one combination of values that makes no sense.
   Solution: `available = true` with `customerId = "C-001"` (an available bike that is also rented), or `available = false` with an empty customer id and no repair note. The enum `BikeStatus` in Chapter 7 rules both out.
5. **Read an error.** In the error message shown earlier in this chapter, what does `12:19` mean, and what does the line `Found: ("18.00" : String)` tell you?
   Solution: Line 12, column 19 is where the problem starts. The compiler found a string literal whose value is `"18.00"`, while the declared type required an `Int`.