ScanLog.pm

```
use v5.36;
sub parse_scan_line ($line) {
  return undef unless
    $line =~ m{ (?<depot>NGT|SMR|HBR)
    \s+ (?<tracking>DM\d{4}) }x;
  my %scan = %+;
  return \%scan;
}

# regex · hashes · references · modules
```

regex

one-liners

CPAN

prove -l

```
$ prove -l t/ All tests successful.
$ perl -lane 'print $F[2] if /DELIVERED/' scans.log
$ dmtrack price --service express --weight 450
```

A COMPLETE, PRACTICAL GUIDE

Perl Programming from Zero

Learn modern Perl 5 and build real tools for text, automation, data, and the web

Perl 5.42 · regular expressions · references · modules
OO & class · testing · DBI & SQLite · Mojolicious

LANGUAGE · TEXT · AUTOMATION · WEB

THENEXBOOKS

Perl Programming from Zero

A Practical Guide to Modern Perl 5 for Text Processing, Automation, and the Web

From your first script to regex-powered tools, tests, databases, and a JSON web service — for Perl 5.42

First Edition

THENEXBOOKS

Perl Programming from Zero
A Practical Guide to Modern Perl 5 for Text Processing, Automation, and the Web

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. Perl and the modules discussed in this book evolve; always consult the official documentation for the versions you use.

"Perl" and the Perl onion logo are trademarks of The Perl Foundation. Visual Studio Code is a trademark of Microsoft Corporation. SQLite is in the public domain; "SQLite" is a registered trademark of Hipp, Wyrick & Company, Inc. Docker is a trademark of Docker, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by The Perl Foundation or the Perl 5 Porters.

Driftmoor Parcels, the business used as the running example in this book, is fictional, as are its people, depots, towns, customers, partners, and data. Domains such as example.com and gullwing-freight.example are reserved for examples.

Preface

Perl was built to make everyday programming jobs easy: pulling the important lines out of a log file, turning one text format into another, gluing programs together, and producing a report before the coffee gets cold. Decades later it still does those jobs remarkably well. Perl ships with almost every Unix-like system, its regular expressions set the standard that many other languages copied, and CPAN gives you a huge library of ready-made modules for nearly any task.

Perl also has a reputation: that it is cryptic, that "there's more than one way to do it" means nobody can read anybody else's code, and that modern development has passed it by. This book shows a different Perl — the one written today. Modern Perl starts with `use v5.36;`, which turns on strictness, warnings, and subroutine signatures in a single line. It has `try` / `catch`, a built-in `class` syntax, solid testing tools, and mature libraries for databases and the web. Written with care, Perl is clear, compact, and very productive.

We learn it by building real software. Each chapter adds a piece to the systems of a small courier company: scripts that parse scan logs, a module that prices parcels, a command-line tracking tool, a tested library, a SQLite database, and a small JSON web service. By the last page you will have written, tested, debugged, and packaged idiomatic Perl — and you will be able to read the older Perl you meet in the wild with confidence.

Who This Book Is For

- **Beginners** who want a practical first language for automating real work with text, files, and data.
- **Developers from other languages** such as Python, JavaScript, PHP, Bash, or Java who want to become productive in Perl quickly and write it the modern way.
- **System administrators, DevOps engineers, and data people** who inherited Perl scripts and need to understand, fix, and improve them safely.
- **Perl users with gaps** who learned by copying snippets and now want a solid understanding of context, references, regular expressions, modules, and objects.

What You Will Learn

- How to install Perl on Windows, macOS, and Linux, read the built-in documentation with `perldoc`, and run scripts and one-liners.
- The language from the ground up: scalars, arrays, hashes, control flow, subroutines with signatures, references, context, and scope.
- Modules and packages, `Exporter`, and `@INC` — how Perl code is organized and reused.
- Perl's superpower: regular expressions, from basic matching to named captures, lookarounds, and readable `/x` patterns.
- Files, directories, Unicode and encodings, CSV, JSON, and formatted reports.

- Object-oriented Perl with `bless`, the modern `class` feature, and Moo; error handling with `die`, `try` / `catch`, and exception objects.
- Sorting, closures, and higher-order functions; command-line tools with Getopt::Long and the famous one-liners.
- Testing with Test::More and `prove`, CPAN and dependency management, databases with DBI and SQLite, HTTP clients, and web services with Mojolicious.
- Automation, dates and times, debugging, performance, best practices, and packaging your code.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) installs Perl and covers the fundamentals: scalars, operators, control flow, and arrays. **Part II — Core Language** (Chapters 6–10) covers hashes, subroutines, references, context and scope, and modules. **Part III — Text Processing** (Chapters 11–15) covers regular expressions, files, Unicode, and data formats. **Part IV — Structured Programs** (Chapters 16–20) covers object-oriented Perl, error handling, functional techniques, and command-line tools. **Part V — Practical Perl** (Chapters 21–25) covers testing, CPAN, databases, the web, and automation. **Part VI — Projects and Next Steps** (Chapters 26–28) assembles the complete tracking tool and web service and prepares you to write, maintain, and ship Perl professionally. The appendices provide a syntax cheat sheet, a one-liner cookbook, an error-message field guide, and a glossary.

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

The Running Example: Driftmoor Parcels

Throughout the book we write software for **Driftmoor Parcels**, a fictional regional courier company with three depots: Northgate, Southmere, and Harbourside. Maren runs operations, Callum dispatches the vans, and Rosa, a data clerk, is learning Perl alongside you. It starts with a simple question — how many parcels did each depot scan today? — and grows into a pricing module, a scan-log parser, `dmtrack`, a command-line tracking tool, a SQLite database, and `dmweb`, a small JSON web service.

Versions and Conventions

Examples target **Perl 5.42**, and nearly all of them start with `use v5.36;`, so they also run on Perl 5.36 and later. Where a feature needs a newer release, we say so. Core-Perl examples were run on Perl 5.42 while writing; examples that use CPAN modules show typical output, because module versions and environments vary. Where a detail may change, a Note points you to the official documentation rather than guessing.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

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

> **WARNING**
>
> A common mistake or something that can cause bugs, data loss, or security problems.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `driftmoor/lib/Driftmoor/Pricing.pm`, tell you where the code lives. Terminal commands are shown with a `$` prompt and work the same on Windows, macOS, and Linux unless noted; one-liners need different quoting in the Windows command prompt, which we point out where it matters.

Contents

Preface

PART I — GETTING STARTED

1 Why Perl? Text, Glue, and the Perl Way

2 Installing Perl, perldoc, Editors, and Your First Script

3 Scalars: Numbers, Strings, and Operators

4 Control Flow: Conditions, Loops, and Statement Modifiers

5 Lists and Arrays

PART II — CORE LANGUAGE

6 Hashes

7 Subroutines and Signatures

8 References and Nested Data Structures

9 Context, Scope, and Variables: my, our, local, and state

10 Modules and Packages: use, require, Exporter, and @INC

PART III — TEXT PROCESSING

11 Regular Expressions I: Matching, Captures, and Substitution

12 Regular Expressions II: Named Captures, Modifiers, Lookarounds, and /x

13 Files, Directories, and I/O

14 Unicode and Encodings

15 Text Formats: CSV, JSON, and Formatted Reports

PART IV — STRUCTURED PROGRAMS

16 Object-Oriented Perl: Packages, bless, and Methods

17 Modern OO: The class Feature and Moo

18 Error Handling: die, eval, try/catch, Exception Objects, and Carp

19 Sorting, Closures, and Higher-Order Functions with List::Util

20 Command-Line Tools: @ARGV, Getopt::Long, POD, Pipelines, and One-Liners

PART V — PRACTICAL PERL

21 Testing with Test::More and prove

22 CPAN: cpanm, local::lib, cpanfile, and Choosing Good Modules

23 Databases with DBI and SQLite

24 HTTP and the Web: HTTP::Tiny, JSON APIs, and Mojolicious

25 Automation: Processes, Environment, Dates and Times, and Debugging

PART VI — PROJECTS AND NEXT STEPS

26 Project: The dmtrack Scan-Log Analyzer End to End

27 Project: A Small JSON Web Service with Mojolicious and SQLite

28 Perl Best Practices, Performance, Packaging, and Where to Go Next

APPENDICES

A Perl Syntax and Special Variables Cheat Sheet

B perl Command-Line Switches and One-Liner Cookbook

C Error and Warning Messages Field Guide

D Glossary

PART I

Getting Started

What Perl is good at and how to set it up, your first scripts and one-liners, and the fundamentals every Perl program is built from: scalars, numbers and strings, operators, control flow, lists, and arrays.

1 · Why Perl?
2 · Installing Perl and Your First Script
3 · Scalars, Numbers, and Strings
4 · Control Flow
5 · Lists and Arrays

CHAPTER 1 · PART I — GETTING STARTED

Why Perl? Text, Glue, and the Perl Way

Before you write a single line, it helps to know what kind of tool you are picking up. This chapter tells Perl's story, explains the ideas that shape the language, shows where it still earns its keep, and gives you a first small taste of real Perl code working on real-looking data.

A Language Built for Getting Things Done

Every programming language is an answer to a question. C answered "how do we write an operating system without assembly?" Java answered "how do we ship the same program to many kinds of machines?" Perl answered a question that every working programmer, system administrator, and data wrangler eventually asks: *"I have a pile of text over here, and I need a slightly different pile of text over there. Why is this so tedious?"*

That question sounds modest, but it covers an enormous amount of real work. Log files, configuration files, CSV exports, reports, email, HTML, command output, data dumps from one system that must be massaged before another system will accept them — most of the information that moves between computers moves as text at some point. Perl was designed to make reading, reshaping, and writing that text quick and pleasant, and then it grew into a full general-purpose language that can also talk to databases, serve web pages, run tests, and automate whole servers.

This book teaches Perl 5 from zero. You do not need to know any Perl, and you do not need to be an experienced programmer. If you have written some Python, JavaScript, PHP, Bash, or Java, you will recognize many ideas, and we will point out the places where Perl thinks differently. If you have inherited Perl scripts at work, by the end of Part II you will be able to read most everyday Perl, and by the end of the book you will write clean, modern, tested Perl of your own.

A Short History of Perl

Perl was created by **Larry Wall**, a programmer with a background in linguistics, who released the first version in **1987**. At the time he was working on Unix systems and needed to produce reports from scattered text files. The standard Unix tools of the day — the shell, `sed`, `awk`, `grep` — could each do part of the job, but stitching them together was clumsy, and each one had its own limits and its own little syntax. Perl pulled the best ideas from those tools into a single language with real variables, real data structures, and no arbitrary limits on line length or file size.

The name is often expanded as "Practical Extraction and Report Language," which describes the original purpose well, although the expansion was attached after the name was chosen. Inside the community you will see **Perl** (capital P) used for the language and **perl** (lowercase) for the program

that runs it — the interpreter you type at the command line. We follow that convention in this book.

Perl went through several versions quickly. The famous book *Programming Perl*, nicknamed "the Camel book" because of the animal on its cover, first appeared in 1991 and helped spread the language widely. The big turning point came in **1994 with Perl 5**. Perl 5 added the features that make large programs manageable: lexical variables declared with `my`, references for building nested data structures, modules and packages for organizing code, and a simple but flexible object system. Perl 5 is the language you are learning in this book, and it has been continuously developed ever since.

Shortly after Perl 5 arrived, in 1995, the **Comprehensive Perl Archive Network (CPAN)** opened: a worldwide, mirrored collection of reusable modules that anyone can download and install. CPAN became one of Perl's defining strengths, and later language ecosystems followed similar ideas with their own package registries. You will learn to use CPAN properly in Chapter 22.

Since the early 2010s, Perl 5 has followed a steady rhythm: a new stable release roughly **once a year**, numbered with even minor versions — 5.36, 5.38, 5.40, 5.42 and so on — plus maintenance releases that fix bugs and security issues. Each yearly release adds features carefully, without breaking the enormous amount of existing code. This book targets **Perl 5.42**, released in 2025, and every complete example in it was run on Perl 5.42.2.

> **NOTE**
>
> You may hear about "Perl 6" in older articles. That project became **Raku**, a separate language with its own compiler and community; it is not a newer version of the Perl you are learning here. You may also find mentions of a future "Perl 7." It has been discussed, but the practical reality is simple: Perl 5 continues with yearly releases, and everything in this book is Perl 5.

The Perl Philosophy

Languages have personalities, and Perl's is unusually easy to describe because its community has put it into slogans. Understanding these ideas will make the language feel less surprising when you meet its quirks.

"There's more than one way to do it"

The most famous Perl motto is **TMTOWTDI**, usually pronounced "Tim Toady." Perl deliberately offers several ways to express the same idea, because natural languages work that way: "if it rains, take an umbrella" and "take an umbrella if it rains" are both correct. Perl lets you write either:

```
if ($raining) { take_umbrella() }
take_umbrella() if $raining;
```

The second form, called a *statement modifier*, reads naturally for short conditions, and you will learn it in Chapter 4. The freedom cuts both ways. In the hands of a careful programmer it produces code that reads almost like a sentence; in a hurry, it can produce code that only its writer understands. Modern Perl practice adds a second half to the motto: "there's more than one way to do it, but sometimes consistency is not a bad thing either." Throughout this book we pick one clear, modern way to do each job and mention the alternatives you will meet in other people's code.

"Easy things should be easy, and hard things should be possible"

The second motto explains why Perl feels so compact for small jobs. Reading a file line by line, splitting a line into fields, matching a pattern, counting things in a hash, and printing a sorted report are all one or two lines of Perl. Yet the same language scales up to modules, objects, tests, database access, and web services when the job grows. You do not have to switch tools when a five-line script turns into a five-hundred-line program; you add structure as you need it.

Context and "do what I mean"

Perl tries hard to do the sensible thing. The string `"42"` becomes the number 42 when you add to it; an array tells you how many elements it has when you use it where a single number is expected; a missing hash entry quietly becomes "nothing" instead of crashing the program. This idea, called *context*, is powerful and occasionally surprising, so Chapter 9 is devoted to it. The modern safety switches that you will turn on in every program — `strict` and `warnings` — exist to make sure "do what I mean" never silently becomes "do something you did not mean."

Where Perl Is Used Today

Perl is no longer the default choice for every new project, as it was for many web sites in the late 1990s. It remains a dependable, everyday tool in several areas, and if you work in any of them you are likely to meet it.

- **Text and log processing.** Pulling fields out of log files, summarizing them, reformatting exports, and cleaning up data are still the jobs Perl does best. Its regular expressions are built into the language rather than bolted on as a library, and the patterns you will learn in Chapters 11 and 12 influenced the regex engines of many later languages.
- **System administration and glue.** Perl sits comfortably between other programs: it runs commands, reads their output, reshapes it, and feeds it to the next step. Many Unix-like systems ship with a `perl` program installed because system tools rely on it. Even Git for Windows includes its own copy of perl, because some Git commands are written in Perl.
- **Bioinformatics and science.** Biological data such as DNA and protein sequences is stored in large text files, which made Perl a natural fit for early genomics work. The BioPerl project and many in-house research scripts remain in use.

- **Legacy and modern web applications.** Much of the early dynamic web ran on Perl CGI scripts, and many of those systems are still running and still being maintained. New web work in Perl uses modern frameworks such as Mojolicious, which you will use in Chapter 24.
- **Build and release tooling.** Perl quietly powers many build systems. GNU Automake is written in Perl, and building OpenSSL from source requires perl because its configuration script is a Perl program. Release scripts, code generators, and test harnesses in many projects are small Perl programs.

Honest Strengths and Weaknesses

No language is right for everything, and you deserve a fair picture before investing your time.

Where Perl shines

- **Text handling.** Regular expressions, string operators, `split`, `join`, `sprintf`, and line-by-line file reading are all first-class parts of the language.
- **Short programs and one-liners.** Command-line switches such as `-n`, `-p`, `-l`, and `-a` turn perl into a programmable filter you can use directly in a terminal (Chapter 20).
- **Stability.** Perl takes backward compatibility seriously. Scripts written many years ago usually still run on a current perl, and new features are switched on explicitly so old code is not disturbed.
- **CPAN and a testing culture.** A huge library of reusable modules, nearly all of which ship with automated tests. Perl's testing tools (Chapter 21) are mature and simple to use.

Where Perl is harder

- **A steeper first week.** Sigils (`$`, `@`, `%`), context, and the many built-in special variables confuse newcomers. This book introduces them slowly and explains the logic behind them.
- **Readability depends on discipline.** Because Perl allows very compact code, it also allows unreadable code. Old scripts written without `strict` and `warnings` can be hard to maintain. Good habits matter more in Perl than in stricter languages.
- **Outdated tutorials.** A lot of Perl material online predates modern features. You will find examples without `strict`, with global variables everywhere, or with hand-rolled argument handling. We show modern style and explain the older forms only so you can recognize them.
- **Smaller presence in some fields.** For machine learning, mobile apps, and browser front ends, other languages have larger ecosystems. Perl is a strong choice for text, automation, and back-end services, not for everything.

> **TIP**
>
> If you come from Python, think of Perl as a language that also values practicality, but trusts the programmer with more freedom and more shortcuts. If you come from Bash, think of Perl as the next step when a shell script grows variables, loops, and data structures that Bash handles awkwardly.

What Modern Perl Looks Like

Perl's commitment to backward compatibility has one consequence you must know from day one: **new features are opt-in**. A perl from 2025 will still run a script from 1995 the way that script expects. To get the modern behavior, you tell perl which version of the language your code is written for. In this book nearly every program starts like this:

```
use v5.36;
```

That single line does a lot of work. It requires perl 5.36 or newer, and it turns on a set of features and safety checks that the language designers recommend for new code. The most important ones are:

- `strict` — you must declare variables (usually with `my`) before using them, so a misspelled variable name becomes an error instead of a silent new variable.
- `warnings` — perl tells you about suspicious things, such as using a value that was never set or treating text as a number.
- `say` — a `print` that adds a newline for you.
- **Subroutine signatures** — you can name a function's parameters directly, like `sub price_cents ($service, $weight_g) { ... }`, and perl checks that the right number of arguments arrived.

Chapter 2 lists the full set and shows exactly what each part catches. To see the difference it makes, compare two versions of the same tiny program. Here is an old-style script with a typo on line 3 — `$totl` instead of `$total`:

old.pl

```
$total = 0;
foreach $w (450, 1200, 3400) {
  $totl += $w;
}
print "Total weight: $total\n";
```

```
$ perl old.pl
Total weight: 0
```

No error, no complaint, just a wrong answer. Perl happily created a brand-new variable named `$totl`. Now the modern version, with `use v5.36` at the top and variables declared with `my`:

strict.pl

```
use v5.36;
my $total = 0;
for my $w (450, 1200, 3400) {
  $totl += $w;
}
say "Total weight: $total";
```

```
$ perl strict.pl
Global symbol "$totl" requires explicit package name (did you forget to declare
"my $totl"?) at strict.pl line 4.
Execution of strict.pl aborted due to compilation errors.
```

(We wrapped the long first line to fit the page; perl prints it as one line.) Perl refuses to run the program at all and points to the exact line, even suggesting the fix. That is the whole spirit of modern Perl: keep the expressive power, but let the computer catch the boring mistakes. Notice also the shape of the message — what went wrong, then `at FILE line N.` Every perl error and warning ends that way, and Chapter 2 teaches you to read them calmly.

Here is a slightly bigger taste of modern style. It uses a hash (a lookup table), a subroutine with a signature, and a loop, to price two parcels using the rules you are about to meet. Do not worry about understanding every symbol yet; just notice that it reads fairly naturally.

modern.pl

```
use v5.36;

my %base_cents = (standard => 495, express => 895, overnight => 1495);

sub price_cents ($service, $weight_g) {
  my $extra_kg = $weight_g > 1000 ? int(($weight_g - 1000 + 999) / 1000) : 0;
  return $base_cents{$service} + 120 * $extra_kg;
}

for my $parcel (['DM1001', 'standard', 1200], ['DM1002', 'express', 450]) {
  my ($tracking, $service, $weight_g) = @$parcel;
  my $cents = price_cents($service, $weight_g);
  printf "%s %-8s %5d g  \$%d.%02d\n", $tracking, $service, $weight_g,
    int($cents / 100), $cents % 100;
}
```

```
$ perl modern.pl
DM1001 standard  1200 g  $6.15
DM1002 express    450 g  $8.95
```

Money is kept as whole **cents** (integers) and only turned into dollars when printed. That is a habit worth adopting in any language, because decimal fractions like 0.1 cannot be stored exactly in floating-point numbers, and rounding errors in money are the kind of bug nobody forgives.

Meet Driftmoor Parcels

Learning a language through disconnected toy examples is like learning to cook by reading ingredient lists. Instead, this book follows one small, realistic project from start to finish. **Driftmoor Parcels** is a fictional regional courier company with three depots:

Code	Depot
`NGT`	Northgate
`SMR`	Southmere
`HBR`	Harbourside

Three people at Driftmoor will appear throughout the book:

- **Maren**, the operations manager. She owns the problem: she needs to know where every parcel is, which depots are busy, and which deliveries failed, and she is tired of finding out by scrolling through raw files.
- **Callum**, the dispatcher. He asks the practical questions — "Which parcels are still not delivered by eleven?" "How much does an express parcel of 450 grams cost?" — and his questions often become our features.
- **Rosa**, a data clerk who is learning Perl alongside you. When something is confusing, Rosa is usually the one who asks about it, and her mistakes are the common ones we all make.

Driftmoor sells three services. Prices are kept in integer cents:

Service	Base price
`standard`	495 cents ($4.95)
`express`	895 cents ($8.95)
`overnight`	1495 cents ($14.95)

On top of the base price, there is a surcharge of **120 cents for every started kilogram over the first kilogram**. A parcel of 1000 grams or less pays no surcharge; 1001 to 2000 grams pays 120 cents extra; a 3400-gram parcel pays 360 cents extra. That "started kilogram" rule — rounding up, not down — is a small trap we will test carefully later.

Every time a parcel passes through a depot, a scanner writes one line to a **scan log**. A line looks like this:

```
2026-04-14T08:20:33Z NGT DM1001 SORTED bay=4
```

Reading left to right: the time in UTC (ISO 8601 format, ending in `Z`), the depot code, the parcel's tracking number, the event, and optional `key=value` details. There are five events: `RECEIVED`, `SORTED`, `LOADED`, `DELIVERED`, and `FAILED`. The full log for 14 April 2026 has fifteen scan lines plus a comment line at the top — and one of those scan lines is deliberately broken, because real data always has a broken line somewhere. Driftmoor also keeps a **manifest**, a CSV file listing each parcel's service, weight, depot, recipient, and city. You will learn to read files in Chapter 13 and meet the manifest properly when we handle CSV in Chapter 15.

By the end of the book, Driftmoor will have a tested Perl module library, a command-line tool called `dmtrack` that answers Callum's questions, a small SQLite database, and a JSON web service. Each piece is built from the concepts in its chapter, so the project grows exactly as fast as your skills do.

A First Taste: Counting Scans per Depot

Maren's first request is simple: "How many scans did each depot record?" Here is a complete Perl program that answers it for a handful of lines. The data is placed right inside the script, after a special marker, so you can run it without creating any other file.

depots.pl

```
#!/usr/bin/env perl
use v5.36;

# Count scans per depot from a few lines of the Driftmoor scan log.
my %count;
while (my $line = <DATA>) {
  my ($time, $depot) = split ' ', $line;
  $count{$depot}++;
}

for my $depot (sort keys %count) {
  say "$depot: $count{$depot}";
}

__DATA__
2026-04-14T07:58:02Z NGT DM1001 RECEIVED courier=c07
2026-04-14T08:01:45Z SMR DM1002 RECEIVED courier=c12
2026-04-14T08:05:10Z HBR DM1003 RECEIVED courier=c03
2026-04-14T08:20:33Z NGT DM1001 SORTED bay=4
2026-04-14T08:22:19Z NGT DM1004 RECEIVED courier=c07
2026-04-14T09:02:51Z SMR DM1002 LOADED van=V22
```

```
$ perl depots.pl
HBR: 1
NGT: 3
SMR: 2
```

Six lines in, three totals out. Let us walk through it once, lightly — every piece gets a full chapter later.

1. `#!/usr/bin/env perl` is the *shebang* line. On Unix-like systems it lets you run the file directly; Chapter 2 explains it.
2. `use v5.36;` switches on modern Perl, as described above.
3. `my %count;` declares a **hash**, Perl's built-in lookup table. The `%` sigil marks it as a hash. Its keys will be depot codes and its values will be counts.
4. `while (my $line = <DATA>)` reads the lines after `__DATA__` one at a time into the scalar variable `$line`. The loop stops when there are no more lines.
5. `split ' ', $line` breaks the line into fields on whitespace. We keep the first two in `$time` and `$depot` and ignore the rest.
6. `$count{$depot}++` adds one to the count for that depot. If the depot is not in the hash yet, Perl treats the missing value as zero and creates the entry — no "key not found" error and no setup code needed.
7. `for my $depot (sort keys %count)` loops over the depot codes in alphabetical order, and `say` prints each with its count.

Why the `sort`? Perl deliberately stores hash entries in an unpredictable order, which can change from one run to the next. If you print `keys %count` without sorting, you might see `SMR` first today and `HBR` first tomorrow. Sorting gives a stable report, and it is a habit you will see in every example in this book.

> **NOTE**
>
> Notice the different first characters: `$line`, `%count`, and later `@lines`. These *sigils* tell you what kind of thing a variable holds: `$` for a single value (a scalar), `@` for an ordered list (an array), and `%` for a hash. When you read `$count{$depot}`, the `$` says "one value," and the curly braces say "from the hash `%count`." Chapters 3, 5, and 6 explain this in depth.

The same idea as a one-liner

Perl is famous for one-liners: whole programs typed directly at the command line. Here is the depot count run against the real `scans.log` file. Do not try to decode it yet — Chapter 20 covers the switches — but look at how little typing it takes:

```
$ perl -lane 'next if /^#/; $c{$F[1]}++;
    END { print "$_ $c{$_}" for sort keys %c }' scans.log
09:30 1
HBR 3
NGT 7
SMR 4
```

Look at the first output line. `09:30` is not a depot! This is the deliberately broken line in the log, which uses a space and no seconds in its time (`2026-04-14 09:30`) instead of the proper ISO 8601 format. Because we split on whitespace, its second field turned out to be `09:30`. This is a valuable early lesson:

quick tools are only as trustworthy as their assumptions about the data. One small change — only count lines whose first field ends in `Z` — fixes it:

```
$ perl -lane 'next unless /^\S+Z /; $c{$F[1]}++;
    END { print "$_ $c{$_}" for sort keys %c }' scans.log
HBR 3
NGT 7
SMR 4
```

Fourteen valid scans: three at Harbourside, seven at Northgate, four at Southmere. The pattern `/^\S+Z /` is a *regular expression* meaning "at the start of the line, some non-space characters followed by a `Z` and a space." In Chapters 11 and 12 you will write much stricter patterns that validate every field, and in Chapter 13 you will learn to warn about broken lines instead of silently skipping them.

> **WARNING**
>
> One-liners are wonderful for exploring data, but they are not a substitute for real programs when the result matters. Once Maren starts making decisions from a report, it should come from a saved script with `use v5.36`, clear variable names, and tests. The one-liner above is a first look, not the final answer.

Counting events instead

Changing the question is usually a small change to the program. This version counts events rather than depots, and uses a regular expression to find the event word on each line:

events.pl

```
use v5.36;

my %count;
while (my $line = <DATA>) {
  next unless $line =~ /\b(RECEIVED|SORTED|LOADED|DELIVERED|FAILED)\b/;
  $count{$1}++;
}
say "$_ $count{$_}" for sort keys %count;

__DATA__
2026-04-14T07:58:02Z NGT DM1001 RECEIVED courier=c07
2026-04-14T08:20:33Z NGT DM1001 SORTED bay=4
2026-04-14T09:15:08Z NGT DM1001 LOADED van=V14
2026-04-14T10:47:29Z NGT DM1001 DELIVERED signed=Okafor
2026-04-14T11:05:12Z SMR DM1005 FAILED reason=no_access
```

```
$ perl events.pl
DELIVERED 1
FAILED 1
```

```
LOADED 1
RECEIVED 1
SORTED 1
```

Two new ideas appear. `next unless ...` skips any line that does not contain an event word — a statement modifier in action. And `$1` holds whatever the parentheses in the pattern captured, so it contains the event name that matched. The last line uses `$_`, Perl's "default variable," which a postfix `for` sets to each item in turn. Chapter 4 explains it.

How This Book Is Organized

The book has six parts. Each chapter builds on the ones before it, and each one adds something to the Driftmoor project.

Part	Chapters	What you will learn
I. Getting Started	1–5	Installing perl, scalars, operators, control flow, lists, and arrays
II. Core Language	6–10	Hashes, subroutines and signatures, references, scope, and modules
III. Text Processing	11–15	Regular expressions, files and I/O, Unicode, CSV, JSON, and reports
IV. Structured Programs	16–20	Objects, the `class` feature and Moo, errors, closures, and CLI tools
V. Practical Perl	21–25	Testing, CPAN, databases with DBI, HTTP and the web, and automation
VI. Projects	26–28	The `dmtrack` analyzer, a JSON web service, and best practices

Parts I and II teach the language itself. Here the Driftmoor examples use small amounts of data written directly into the scripts, so you can focus on the language rather than on files. In Chapter 10 the code moves into proper modules — `Driftmoor::Pricing` for prices and `Driftmoor::ScanLog` for parsing the log — which the rest of the book builds on.

Part III is where Perl's reputation was made: regular expressions, reading and writing files, handling text in many languages correctly, and turning messy data into clean reports. Part IV teaches you to structure bigger programs with objects, proper error handling, and higher-order functions, and ends with a real command-line tool. Part V covers the professional toolkit — tests, CPAN modules, databases, HTTP — and Part VI brings it all together in two complete projects and a chapter on best practices and where to go next. The appendices are quick references: a syntax cheat sheet, a one-liner cookbook, an error-message field guide, and a glossary.

How to read this book

Programming is learned with your hands, not only your eyes. We strongly suggest that you type the examples yourself rather than copying them. The typos you make along the way will teach you to read perl's error messages. Each chapter ends with a Practice section. Try the exercises before you read the Selected Solutions; even a failed attempt teaches more than reading an answer.