PERL

Intermediate / Advanced

Advanced Regex · Context · References & Data Structures

Classic OOP · Modules & CPAN · Error Handling

Closures · Testing · One-Liners & Text Processing

Philip Osztromok

Generated with Claude

Table of Contents

Perl Intermediate/Advanced — 9 Chapters

🔬 Advanced Regular Expressions

Welcome to Perl Intermediate/Advanced. Course 1, Chapter 6 covered `m//`, `s///`, `tr///`, and basic capture groups. This chapter goes into the features that made Perl's regex engine the template PCRE was built from in the first place: lookaround assertions, named captures, and the readability-focused `/x` mode.

👀 Lookahead & Lookbehind

Lookaround assertions check that something is (or isn't) present next to a match — without actually including it in the matched text itself.

```
# positive lookahead (?=...) — match only if followed by...
my $text = "12kg 30lb 5kg";
my @matches = $text =~ /(\d+)(?=kg)/g;
say "@matches";   # 12 5 — only numbers actually followed by "kg", "kg" itself not captured
# negative lookahead (?!...) — match only if NOT followed by...
my @not_kg = $text =~ /(\d+)(?!kg)/g;

# positive lookbehind (?<=...) — match only if preceded by...
my $price = "Price: $50";
if ($price =~ /(?<=\$)(\d+)/) {
    say $1;   # 50 — the $ itself was required, but not captured
}
```

The key idea: lookaround *asserts* context without *consuming* it — the surrounding text is checked, but only what's inside the actual capturing parentheses ends up in the match. This is genuinely more precise than trying to capture-and-discard the surrounding text manually.

⚠ Lookbehind Must Be Fixed-Width

Perl's lookbehind (by default) requires the assertion to match a **fixed number of characters** — `(?<=\$)` (one character) is fine, but something like `(?<=\d+)` (a variable-length run of digits) raises `Variable length lookbehind not implemented` in many contexts. This is a real, common limitation to run into once you reach for lookbehind on anything beyond a short, fixed pattern — lookahead has no such restriction.

🏷️ Named Captures

```
my $log = "2026-07-10 ERROR Connection timeout";

if ($log =~ /(?<date>\d{4}-\d{2}-\d{2}) (?<level>\w+) (?<message>.+)/) {
    say $+{date};     # 2026-07-10
say $+{level};    # ERROR
say $+{message};  # Connection timeout
}
```

`(?<name>...)` names a capture group, retrievable afterward through the special hash `%+` as `$+{name}`. This is a genuine readability upgrade over Course 1's `$1`/`$2`/`$3` — the code reads as "the date," "the level," "the message" directly, and reordering or adding groups elsewhere in the pattern doesn't silently renumber and break every existing reference the way inserting a new plain capture group would with `$1`-style captures.

📖 The /x Extended Mode

`/x` tells Perl to ignore whitespace in the pattern (unless explicitly escaped or inside a character class) and allows `#` comments directly inside it — turning a dense, hard-to-read pattern into something closer to documented code:

```
# Course 1, Chapter 6's job-reference pattern, dense and hard to scan:
/Job Reference(?: Number)?:\s*(JO-\d+-\d+)/
# the same pattern, rewritten with /x for readability:
my $pattern = qr/
   Job \s Reference        # the label text
   (?: \s Number )?        # "Number" is optional
   :                    # the colon
   \s*                  # any whitespace left after stripping HTML tags
   ( JO- \d+ - \d+ )       # capture the actual reference number
/x;
```

Every meaningful space in the pattern that needs to match a literal space has to be written as `\s` or `\ ` (escaped) under `/x`, since plain whitespace in the pattern is now ignored — a real trade-off, but for anything beyond a short pattern, the readability gain from being able to comment each piece is substantial.

Other Useful Advanced Features

```
# greedy (default) vs non-greedy quantifiers:
"<a><b>" =~ /<(.+)>/;   # $1 = "a><b" — greedy, matches as MUCH as possible
"<a><b>" =~ /<(.+?)>/;  # $1 = "a" — non-greedy (the ?), matches as LITTLE as possible
# word boundary:
"cat category" =~ /\bcat\b/g;  # matches only the standalone "cat", not the "cat" inside "category"
```

A greedy quantifier (`+`, `*`) consumes as much as it can before backtracking to satisfy the rest of the pattern; appending `?` (`+?`, `*?`) flips it to match as little as possible instead — genuinely important the moment a pattern spans multiple delimiters, exactly the `<a><b>` case above.

🔧 A Worked Example — Validating and Parsing a Password

```
my $password = "Secr3t!Pass";

my $is_valid = $password =~ /
  ^
  (?=.*[A-Z])     # lookahead: at least one uppercase letter, anywhere
  (?=.*[0-9])     # lookahead: at least one digit, anywhere
  (?=.*[!@#\$%])  # lookahead: at least one symbol, anywhere
  .{8,}           # at least 8 characters total, actually consumed
  $
/x;

say $is_valid ? "Valid password" : "Too weak";
```

Each `(?=...)` lookahead checks a requirement independently, from the *same starting position* (lookahead doesn't advance the match position), without any of them consuming characters the others need to also see — this is a genuinely idiomatic way to validate "must contain X, and Y, and Z, in any order" with a single regex rather than several separate checks.

Lookaround: Perl vs. regex1's POSIX Regex

The site's own *Learning Regular Expressions* course (grep/sed/awk, POSIX BRE/ERE) has no lookaround support at all without reaching for GNU-specific extensions — POSIX regex simply wasn't designed with it. This is a concrete, practical instance of Course 1 Chapter 6's point that PCRE (adopted by countless other tools) grew directly out of Perl's own regex engine specifically because Perl supported features like this natively, years before they became commonplace elsewhere.

Lookahead / lookbehind

`(?=...)`/`(?!...)` and `(?<=...)`/`(?<!...)` — assert without consuming.

Named captures

`(?<name>...)`, retrieved via `$+{name}` — more maintainable than `$1`.

/x mode

Ignores whitespace, allows `#` comments — turns a dense pattern into readable code.

Greedy vs. non-greedy

`+?`/`*?` match as little as possible, instead of as much as possible.

💻 Coding Challenges

Challenge 1: Extract Prices Without the Currency Symbol

Given `my $text = "Items: $12, $8, $45";`, use a lookbehind to extract just the numbers (not the `$`) into a list, and print them.

→ Solution

Challenge 2: Named Captures for a Date

Given `my $date = "2026-07-10";`, use named captures (`?<year>`, `?<month>`, `?<day>`) to extract each part, and print them as `"Year: 2026, Month: 07, Day: 10"` using `%+`.

→ Solution

Challenge 3: Rewrite with /x

Take the regex `/^(\w+)@(\w+\.\w+)$/` (a simple email matcher) and rewrite it using `/x` mode with a comment explaining each piece, keeping its matching behavior identical.

→ Solution

🎯 What's Next

Next chapter: **Context Deep Dive** — `wantarray`, and scalar vs. list context edge cases, rigorously revisited from Course 1's introduction to the topic.

🎭 Context Deep Dive

Course 1 introduced context twice — briefly with arrays (Chapter 3) and again with `wantarray` in subroutines (Chapter 7). Both were previews. Context is one of Perl's deepest, most pervasive design principles — it touches regex matches, hashes, built-in functions, and far more than just "does this go in an array or a scalar." This chapter treats it properly.

The Three Contexts: List, Scalar, Void

Every Perl expression is evaluated in exactly one of three contexts, determined entirely by what surrounds it:

Context	Triggered by	wantarray() returns
List	Assigning to an array/hash, passing as subroutine arguments, appearing inside a list	True
Scalar	Assigning to a scalar, a boolean test, a numeric/string operation	`''` (defined, but false)
Void	The result is discarded entirely — e.g. a bare subroutine call as its own statement	`undef`

Course 1, Chapter 7 showed `wantarray` distinguishing list from scalar with a simple `if`. Formally, it returns three genuinely different values — true, a defined-but-false empty string, and `undef` — letting a subroutine detect all three situations, including the case where its return value isn't being used for anything at all.

🌊 Context Propagates Through Operators, Not Just Assignment

Context isn't limited to "which variable am I assigning to" — it flows through regex matches, function calls, and more. The clearest (and most famous) example combines two contexts in one line:

```
my $text = "cat dog cat bird cat";
my $count = () = $text =~ /cat/g;
say $count;   # 3
```

This idiom (informally nicknamed for how the `= () =` looks) works by chaining two assignments: `$text =~ /cat/g` in **list context** (forced by assigning to `()`, an empty list) returns every match — all 3 occurrences of "cat." That list is then immediately assigned to `$count`, a scalar — which puts the whole assignment expression in **scalar context**, and a list assigned to a scalar evaluates to the number of elements on the right-hand side. Two contexts, chained through one expression, doing real work: counting matches without a manual loop.

🔀 Scalar Context on Different Data Types

```
my @array = (1, 2, 3);
my $n = @array;        # 3 — the count (Course 1, Chapter 3)
my %hash = (a => 1, b => 2);
my $key_count = keys %hash;   # 2 — a hash in scalar context is its key count, in modern Perl
sub get_data { return wantarray() ? (1,2,3) : "scalar result"; }
my $result = get_data();   # "scalar result" — the sub itself decides, via wantarray
```

A hash evaluated in scalar context reports its key count in current Perl versions (older Perl historically returned a bucket-usage ratio string here — worth knowing if you ever encounter genuinely old code, but not something to write yourself). A subroutine's behavior in scalar context is entirely up to that subroutine's own use of `wantarray`, generalizing Course 1, Chapter 7's example.

🎯 Forcing Context Explicitly

```
say scalar(@array);       # forces scalar context — works on anything, not just arrays
say scalar(get_data());   # forces get_data() to run in scalar context specifically
```

`scalar(...)` forces whatever's inside it into scalar context, regardless of where it appears — useful any time the surrounding code wouldn't naturally impose the context you actually want.

⚠️ Common Context Gotchas

⚠ return @array; Isn't Always "Return the Array"

```
sub get_items {
    my @items = ("a", "b", "c");
    return @items;   # context-sensitive! propagates the CALLER's context
}

my @all   = get_items();   # ("a", "b", "c")
my $count = get_items();   # 3 — NOT "c" (the last element)! return propagates scalar context into @items too
```

`return` itself is context-sensitive — it evaluates its argument in whatever context the subroutine was *called* in, then hands that back. `return @items;` called in scalar context evaluates `@items` in scalar context too (its count), not "the last element" the way you might expect from some other languages. If you specifically want a subroutine to *always* return a count regardless of how it's called, write `return scalar(@items);` explicitly.

`localtime()` is a well-known built-in example of this same principle at work: called in list context, it returns a 9-element list of individual date/time parts (seconds, minutes, hours, and so on); called in scalar context, the exact same function instead returns one formatted date string like `"Fri Jul 10 14:32:05 2026"` — genuinely different data shapes from the same call, purely from context.

🔧 A Worked Example — The Count Idiom, Fully Explained

```
my $log = "INFO INFO ERROR INFO WARN ERROR INFO";

my $error_count = () = $log =~ /ERROR/g;
say "Errors found: $error_count";   # Errors found: 2
```

Step by step: `$log =~ /ERROR/g` is evaluated first — because it's being assigned to `()`, an empty list, it runs in list context and returns every match (two "ERROR" occurrences). That two-element list is then assigned to `()` itself (a no-op destination, just there to force list context), and the *entire* `() = ...` expression, evaluated in scalar context (because it's being assigned to `$error_count`), evaluates to how many elements were on its right side: 2.

Context: Perl vs. Most Other Languages

Python and JavaScript functions return one fixed "shape" of result, always, regardless of how the caller uses it — `len(my_list)` is a completely separate call from just using `my_list`, with no ambiguity about which one you're invoking. Perl's context system has no real equivalent in either language: the exact same expression, `@array` or `my_sub()`, can mean fundamentally different things depending purely on where it's used. This is genuinely one of the more advanced, distinctive things to internalize about the language — and, once it clicks, one of the more powerful.

Three contexts

List, scalar, void — `wantarray()` distinguishes all three.

The count idiom

`my $n = () = $str =~ /pat/g;` — counts matches via chained context.

return is context-sensitive

Propagates the caller's context into the returned expression automatically.

scalar(...) forces context

Works on anything — arrays, hashes, subroutine calls.

💻 Coding Challenges

Challenge 1: Count Vowels with the Count Idiom

Given `my $word = "encyclopedia";`, use the `my $n = () = ...` idiom with a regex to count how many vowels it contains, and print the count.

→ Solution

Challenge 2: A Context-Aware Subroutine

Write a subroutine `get_names` that returns the list `("Ada", "Grace", "Alan")` in list context, and the string `"3 names"` (not just the number) in scalar context, using `wantarray`. Call it both ways and print both results.

→ Solution

Challenge 3: Fix the return Context Bug

Given a subroutine that ends with `return @results;` where the caller wants ONLY the count regardless of how it's called, rewrite the return line so it always returns the count, and explain in one sentence why the original version wouldn't reliably do that.

→ Solution

🎯 What's Next

Next chapter: **Object-Oriented Perl (Classic)** — packages, `bless`, `@ISA` inheritance, and manual method dispatch.

🏛️ Object-Oriented Perl (Classic)

This is the "old school" style of Perl OOP — predating the modern Moo/Moose approach Chapter 4 covers, and still genuinely common in legacy code you're likely to encounter in real work. There's no dedicated `class` keyword here: a "class" is just a package combined with a blessed reference, built entirely out of tools already covered — packages, references (Course 1, Chapter 9), and subroutines (Course 1, Chapter 7).

📦 Packages — Perl's Namespace Mechanism

```
package Animal;

# ... subroutines defined here belong to the Animal namespace ...
1;   # a .pm file must end with a true value — this is standard, if easy to forget
```

`package Animal;` declares a namespace — every subroutine defined after it, until the next `package` statement (or end of file), belongs to `Animal`. In real code, one package typically lives in its own `.pm` file named to match (`Animal.pm`), loaded elsewhere with `use Animal;` (Chapter 5 covers modules in depth).

🎯 bless — Turning a Reference into an Object

```
package Animal;

sub new {
    my $class = shift;
    my $name  = shift;

    my $self = { name => $name };   # an ordinary anonymous hashref (Course 1, Chapter 9)
bless $self, $class;            # marks that hashref as belonging to $class
return $self;
}
```

`bless` takes an ordinary reference (a hashref, almost always) and a class name, and marks that reference as an instance of that class — nothing more mysterious than that. `new` is **not a language keyword** — it's just a subroutine name, chosen entirely by convention. Nothing stops you naming a constructor something else, but every real Perl codebase calls it `new`, and deviating would only confuse readers.

📞 Methods — Just Subroutines with an Implicit First Argument

```
sub speak {
    my $self = shift;   # the object itself — always arrives as the first element of @_
return "$self->{name} makes a sound";
}

my $animal = Animal->new("Generic Animal");
say $animal->speak();   # Generic Animal makes a sound
```

`$animal->speak()` is really just syntax sugar for `speak($animal)` — Course 1, Chapter 7's `@_`-based argument passing, unchanged. The object arrives as the very first element of `@_`, which is why `my $self = shift;` is the near-universal first line of every method — exactly the same `shift` from Course 1, Chapter 3's array operations, now doing double duty as "unpack the invocant."

🧬 @ISA — Manual Inheritance

```
package Dog;
our @ISA = ('Animal');   # Dog inherits from Animal
sub speak {              # overrides Animal's speak()
my $self = shift;
   return "$self->{name} says Woof!";
}
```

`@ISA` ("is a") is a package variable listing a package's parent(s). When a method is called, Perl searches the current package first, then walks `@ISA` looking for a match — `Dog` defining its own `speak` means that version wins, but `Dog` still inherits `Animal`'s `new` automatically, since `Dog` never defines its own.

💡 use parent Is the Cleaner Modern Spelling

Directly assigning `@ISA` works, but `use parent -norequire, 'Animal';` is the more common modern form — it does the same thing, with slightly better error-checking behavior. Both are worth recognizing; `@ISA` itself is what's actually happening underneath either spelling.

⬆️ Calling the Parent's Method — SUPER::

```
package Puppy;
our @ISA = ('Dog');

sub speak {
   my $self = shift;
   my $base = $self->SUPER::speak();   # calls Dog's speak(), not Puppy's own
return "$base (but higher-pitched)";
}
```

`SUPER::method()` explicitly calls the *parent* class's version of a method rather than the current package's — useful for extending, rather than fully replacing, inherited behavior.

🔧 A Worked Example — Animal/Dog/Cat Hierarchy

```
package Animal;
sub new   { my ($class, $name) = @_; return bless { name => $name }, $class; }
sub speak { my $self = shift; return "$self->{name} makes a sound"; }

package Dog;
our @ISA = ('Animal');
sub speak { my $self = shift; return "$self->{name} says Woof!"; }

package Cat;
our @ISA = ('Animal');
sub speak { my $self = shift; return "$self->{name} says Meow!"; }

package main;
my @animals = (Dog->new("Rex"), Cat->new("Whiskers"));

foreach my $animal (@animals) {
    say $animal->speak();   # each object uses its OWN speak() automatically — polymorphism
}
```

The `foreach` loop never checks "is this a `Dog` or a `Cat`" — calling `->speak()` automatically runs whichever version belongs to that object's actual class. This is the identical polymorphism concept from the site's own Python course (`py2-2`'s `Dog`/`Cat` example) — same idea entirely, expressed through completely different mechanics: Python's built-in class system versus Perl's manual package-plus-`bless` construction.

⚠ Forgetting to shift the Class Name First

A constructor's first argument is always the class name — `Dog->new("Rex")` passes `@_ = ('Dog', 'Rex')`, not just `('Rex')`. Forgetting `my $class = shift;` as the very first line, before unpacking anything else, means every later argument is off by one — a classic, confusing bug where the constructor silently receives the wrong values with no error at all.

Classic Perl OOP vs. Python Classes

Python's `class` keyword, automatic `self` parameter, and `__init__` constructor are all built directly into the language. Classic Perl OOP has none of that: `self` (called `$self` here) is an ordinary variable you manually `shift` off `@_` yourself, `new` is a plain subroutine name rather than a reserved constructor keyword, and "class" simply means "package plus `bless`." Nothing is hidden or automatic — which is precisely why Chapter 4's Moo/Moose covers a considerably more ergonomic modern alternative, while this chapter's manual version remains essential for reading the vast amount of existing Perl code still built this way.

package

Declares a namespace — the closest thing classic Perl has to a "class."

bless

Marks an ordinary reference as belonging to a class.

Methods

`$obj->method()` is sugar for `method($obj)` — always `shift` first.

@ISA / SUPER::

Manual inheritance; `SUPER::` explicitly calls the parent's version.

💻 Coding Challenges

Challenge 1: A Basic Class

Write a package `Book` with a `new($title)` constructor and a `describe` method returning `"Title: $title"`. Create an instance and call `describe` on it.

→ Solution

Challenge 2: Inheritance with @ISA

Write a package `Shape` with an `area` method returning 0, then a subclass `Square` (using `@ISA`) with its own constructor storing a `side` and its own `area` method computing `side * side`.

→ Solution

Challenge 3: Extend with SUPER::

Given the chapter's `Animal`/`Dog` classes, write a `LoudDog` subclass (inheriting from `Dog`) whose own `speak` calls `SUPER::speak` and appends `" (LOUDLY)"` to the result.

→ Solution

🎯 What's Next

Next chapter: **Modern OOP with Moo/Moose** — a lighter, more ergonomic alternative to hand-rolled OOP, with attributes and roles.

✨ Modern OOP with Moo/Moose

Chapter 3 showed the manual `bless`/`@ISA` approach still common in legacy code. This chapter shows what most *new* Perl OOP code actually reaches for today: **Moo** and **Moose**, both of which eliminate nearly all of the boilerplate — hand-written accessors, manual constructors, manual `@ISA` assignment — the previous chapter required by hand.

⚖️ Moose vs. Moo — Which One?

	Moose	Moo
Feature set	The original, full-featured, built on a complete meta-object protocol	A lightweight, largely Moose-compatible subset
Startup cost	Heavier — more dependencies, slower to load	Minimal dependencies, fast startup
Typical use	When you need Moose's deeper introspection/meta-programming features	The common default for most modern code

This chapter uses Moo for its examples, since it's the more commonly reached-for modern default — the attribute/inheritance syntax shown below is nearly identical in Moose.

🏷️ Defining Attributes

```
package Animal;
use Moo;

has 'name' => (is => 'ro');   # read-only — generates a getter only
has 'age'  => (is => 'rw');   # read-write — generates a getter AND a setter
```

`has` replaces every hand-written accessor from Chapter 3's `$self->{name}` pattern — declaring an attribute automatically generates a method of the same name. `is => 'ro'` gives you `$animal->name` (read-only); `is => 'rw'` additionally lets you call `$animal->age(29)` to update it. No manual hashref key access anywhere in your own code.

The Automatic Constructor

```
my $rex = Animal->new(name => "Rex", age => 3);
say $rex->name;   # Rex — no hand-written new() needed at all
```

Moo generates `new` for you — no manually `shift`-ing the class name and building a hashref by hand, per Chapter 3's constructor pattern. Attributes accept two more useful options:

```
has 'name' => (is => 'ro', required => 1);   # new() dies if "name" isn't supplied
has 'legs' => (is => 'ro', default => sub { 4 });   # default value if not supplied
```

⚠ default Must Be a Coderef for Anything Mutable

```
has 'tricks' => (is => 'rw', default => []);         # WRONG — errors, or shared across instances
has 'tricks' => (is => 'rw', default => sub { [] });   # correct — a fresh arrayref built per instance
```

This is the exact same root cause as Course 1, Chapter 2's mutable-default-argument gotcha, resurfacing in a new form: `default => []` would build *one* arrayref when the attribute is declared, shared by every instance that doesn't supply its own — modifying it on one object would leak into every other. Wrapping it in `sub { [] }` defers building the default until each individual object is actually constructed, giving every instance its own independent, fresh reference.

🧬 Inheritance with extends

```
package Dog;
use Moo;
extends 'Animal';   # replaces Chapter 3's manual: our @ISA = ('Animal');
sub speak {
    my $self = shift;
    return $self->name . " says Woof!";
}
```

`extends 'Animal';` is the entire inheritance declaration — considerably cleaner than manually assigning `@ISA`, and `Dog` still inherits `Animal`'s attributes (`name`, its generated accessor, its constructor logic) automatically.

🎭 Roles — Composition Over Inheritance

```
package Loud;
use Moo::Role;   # a ROLE, not a class — no "new", meant to be composed in
sub speak_loudly {
    my $self = shift;
    return uc($self->speak());
}

package Dog;
use Moo;
extends 'Animal';
with 'Loud';   # composes the Loud role's methods directly into Dog
```

A **role** (`with 'Loud';`) is genuinely different from inheritance — it's a bundle of methods *composed directly into* a class, rather than inherited from a parent up a chain. This sidesteps a lot of the awkwardness multiple inheritance can create (Course 3's Vim course, and the site's own OOP-focused chapters elsewhere, cover this same composition-over-inheritance instinct from other angles) — a class can compose several unrelated roles together without any of the ambiguity a shared parent class hierarchy might introduce.

🔧 A Worked Example — Rewriting Chapter 3's Animal Hierarchy

```
package Animal;
use Moo;
has 'name' => (is => 'ro', required => 1);
sub speak { my $self = shift; return $self->name . " makes a sound"; }

package Dog;
use Moo;
extends 'Animal';
sub speak { my $self = shift; return $self->name . " says Woof!"; }

package main;
my $rex = Dog->new(name => "Rex");
say $rex->speak();   # Rex says Woof!
```

Compare this directly to Chapter 3's hand-rolled version: no manual `new`, no manual hashref-building, no manual `@ISA` assignment, no hand-written accessor for `name` — every piece that was pure boilerplate before is now generated by `use Moo;`, `has`, and `extends`, leaving only the logic that's genuinely specific to each class.

Moo/Moose Attributes vs. Python's @dataclass

The site's own Python Advanced course closes its capstone with `@dataclass` — auto-generating `__init__`, `__repr__`, and `__eq__` from type-hinted attributes. Moo's `has` aims at genuinely the same problem (eliminate boilerplate for a data-holding class) but goes further by default: accessors, a full constructor with required/default handling, and — via `extends`/`with` — inheritance and composition, all from the same declarative style. Different language, same underlying instinct: declare what an object holds, let the framework generate the mechanical code around it.

has

Declares an attribute; `is => 'ro'`/`'rw'` controls getter/setter generation.

Automatic new

No manual constructor — `required`/`default` handle validation and defaults.

extends

Clean inheritance declaration, replacing manual `@ISA` assignment.

Roles (with)

Compose methods into a class directly — genuinely different from inheritance.

💻 Coding Challenges

Challenge 1: A Moo Class

Write a Moo class `Book` with a required, read-only `title` attribute and a read-write `read` attribute defaulting to false. Create an instance without setting `read`, then update it to true using the generated setter.

→ Solution

Challenge 2: Fix the Mutable Default

Given a buggy `has 'tags' => (is => 'rw', default => []);`, rewrite it correctly per this chapter's warn-box, and explain in one sentence why the original version is dangerous.

→ Solution

Challenge 3: A Role

Write a Moo role `Describable` with a method `summary` that returns `"This is a " . ref($self)`. Compose it into a class `Widget` using `with`, and call `summary` on an instance.

→ Solution

🎯 What's Next

Next chapter: **Modules & CPAN** — `use`/`require`, writing your own module, and the CPAN ecosystem.

📦 Modules & CPAN

Course 1, Chapter 1 briefly mentioned Strawberry Perl shipping `cpanm`. This chapter covers the full picture: loading existing modules properly, writing and exporting your own, and CPAN itself — one of the oldest, largest package ecosystems in programming, predating both npm and PyPI's modern form by well over a decade.

use vs. require

```
use List::Util qw(max min sum);   # compile-time loading, imports specific symbols immediately
say max(3, 7, 2);                 # 7 — used directly, no package prefix needed
require List::Util;               # runtime loading, no automatic import
say List::Util::max(3, 7, 2);     # must fully qualify, or call ->import() manually
```

`use` is what you want for essentially all real code — it loads the module and imports its exported symbols at compile time, before the rest of the script even runs. `require` loads at runtime instead, with no automatic import — genuinely useful for the rarer case of conditionally loading a module based on logic that can't be known until the script is actually running.

✍️ Writing Your Own Module

```perl
# Utils.pm
package Utils;
use strict;
use warnings;

sub shout {
    my ($text) = @_;
    return uc($text) . "!";
}

1;   # required — see this chapter's own warn-box
```

This reuses Chapter 3's `package` declaration directly — a module is just a package, conventionally saved in a file matching its name (`Utils.pm` for `package Utils;`).

⚠ A .pm File Must End With a True Value

When `use` or `require` loads a file, Perl checks the file's *last evaluated expression* — if it isn't true, loading fails with `Utils.pm did not return a true value`. The trailing `1;` is a plain true value placed deliberately as the file's final statement, purely to satisfy this requirement. Forgetting it is a genuinely common first-module mistake — the module's actual code can be entirely correct and still fail to load.

📤 Exporting Symbols with Exporter

```
# Utils.pm
package Utils;
use Exporter 'import';
our @EXPORT_OK = qw(shout);   # available, but NOT imported unless explicitly requested
# calling script:
use Utils qw(shout);
say shout("hello");   # HELLO!
```

`@EXPORT_OK` lists symbols available for opt-in import — the calling script must explicitly request them (`use Utils qw(shout);`). This is the recommended practice over the older `@EXPORT`, which exports everything listed unconditionally into every script that uses the module — a real risk of silently colliding with the caller's own subroutine or variable names. Explicit opt-in keeps namespace pollution under the caller's control.

🌐 The CPAN Ecosystem

CPAN (the Comprehensive Perl Archive Network) launched in 1995 — genuinely one of the oldest centralized package repositories in programming, predating npm (2010) and PyPI's modern form by well over a decade. It's a significant part of why Perl became so practically useful early: an enormous, searchable library of community-published modules for nearly anything.

```
cpanm List::Util        # cpanm (App::cpanminus) — the modern, simple CPAN client
cpanm --installdeps .   # install everything listed in a project's cpanfile
```

A `cpanfile` declares a project's dependencies — the direct Perl parallel to Course 1, Chapter 9's Python `requirements.txt` or Node's `package.json`.

⭐ A Few Genuinely Useful CPAN Modules

Module	What it's for
`Path::Tiny`	The nicer file-handling wrapper previewed back in Course 1, Chapter 8 — read/write a whole file in one call
`JSON::PP`	JSON encoding/decoding — actually part of Perl core since 5.14, no install needed
`Try::Tiny`	A cleaner error-handling syntax, previewed ahead of Chapter 6
`DateTime`	Robust date/time handling, well beyond `localtime`'s context-sensitive quirkiness from Chapter 2

🔧 A Worked Example — A Small Module

```
# MathUtils.pm
package MathUtils;
use strict;
use warnings;
use Exporter 'import';
our @EXPORT_OK = qw(square cube);

sub square { my ($n) = @_; return $n * $n; }
sub cube   { my ($n) = @_; return $n * $n * $n; }

1;


# script.pl
use strict;
use warnings;
use feature 'say';
use MathUtils qw(square cube);

say square(4);   # 16
say cube(3);    # 27
```

CPAN vs. pip/npm

The underlying model is the same across all three — a central registry, a simple install command, community-published packages. The real difference is age and culture: CPAN's 1995 launch makes it genuinely older than npm or PyPI's current form by well over a decade, and it fostered a "there's already a well-tested module for this" culture in Perl development early — much of the reason Perl earned a reputation as a practical, get-things-done language for real work, not just an academic one.

use vs require

Compile-time with automatic import, vs. runtime with none.

The trailing 1;

Every `.pm` file must end with a true value or loading fails.

@EXPORT_OK

Opt-in exporting — safer than unconditional `@EXPORT`.

cpanm / cpanfile

The modern CPAN client, and a project's declared dependencies.

💻 Coding Challenges

Challenge 1: Write and Use a Module

Write a module `StringUtils.pm` exporting a subroutine `reverse_string` via `@EXPORT_OK`. Write a separate script that `use`s it and prints the result of reversing `"Perl"`.

→ Solution

Challenge 2: Diagnose the Missing 1;

Given a module file whose last line is a subroutine definition (with no trailing `1;`), explain in one sentence what error occurs when a script tries to `use` it, and show the corrected final lines.

→ Solution

Challenge 3: use vs require

Write two versions of loading and calling `List::Util`'s `max` function on the list `(3, 9, 4)` — one using `use` with a qw() import list, one using `require` with a fully-qualified call — and explain in one sentence the key difference between them.

→ Solution

🎯 What's Next

Next chapter: **Error Handling** — `die`/`eval`, `$@`, and modern `try`/`catch` (Perl 5.34+).

🚨 Error Handling

Course 1's `open(...) or die "..."` idiom raised errors — this chapter covers actually *catching* and handling them, from the classic `eval`/`$@` approach through Perl's modern, genuinely cleaner `try`/`catch` syntax.

💀 die — Raising an Error

```
die "Something went wrong";
# Something went wrong at script.pl line 12.   — Perl appends " at FILE line N." automatically
die "Something went wrong\n";
# Something went wrong                        — a trailing \n suppresses the automatic location text
```

If a `die` message doesn't already end in a newline, Perl appends the file and line number automatically — genuinely useful for debugging, but the reason some error messages in real scripts show a location and others don't purely depends on whether the message ends in `\n`.

🪝 eval { } — Catching an Error

```
eval {
    die "Something failed\n";
};
if ($@) {
    say "Caught: $@";   # Caught: Something failed
}
```

An `eval { }` block catches any `die` raised inside it — execution jumps straight to just after the block instead of terminating the program, and the special variable `$@` holds whatever was passed to `die` (empty string if nothing went wrong). This is the classic, pre-5.34 way Perl has always done exception handling.

⚠ $@ Can Be Clobbered Before You Check It

```
eval { die "oops\n"; };
# if something else runs an eval, or a DESTROY method fires during garbage collection here...
if ($@) { ... }   # $@ might have been silently overwritten by then!
```

`$@` is a global, and anything that runs between your `eval` and your check of `$@` — including an unrelated `eval` elsewhere, or a destructor firing during garbage collection — can silently overwrite it. Best practice: capture it into your own variable *immediately*: `my $err = $@; if ($err) { ... }`.

✨ Modern try/catch (Perl 5.34+)

```
use feature 'try';
no warnings 'experimental::try';

try {
    die "Something failed\n";
}
catch ($e) {
    say "Caught: $e";   # Caught: Something failed
}
```

This is the same error caught the same way — but `$e` is a genuinely local, unambiguous variable scoped to the `catch` block, sidestepping every `$@`-clobbering concern from the warn-box above entirely. On Perl 5.34 or later, this is the recommended approach for new code.

 die with a Reference — Structured Exceptions

```
eval {
    die { code => 404, message => "Not found" };   # dying with a HASHREF, not a plain string
};
if (my $err = $@) {
    say $err->{code};   # 404 — $@ IS that reference, not a stringified message
}
```

`die` accepts a reference just as readily as a string — `$@` (or `$e` in the modern `catch` form) becomes that exact reference, letting catching code inspect structured data (an error code, a type, extra context) instead of pattern-matching a string message. This is a natural extension of Chapter 3/4's OOP: a real codebase often defines proper exception *classes* and `die`s with a blessed object instead of a plain hashref, giving `catch` code the ability to check `ref($e)` or call methods on the caught error directly.

🔧 A Worked Example — Robust File Processing

```
use feature 'try';
no warnings 'experimental::try';

sub process_file {
   my ($filename) = @_;

   try {
      open(my $fh, '<', $filename) or die "Can't open $filename: $!\n";
      while (my $line = <$fh>) {
         # ... process each line ...
      }
      close($fh);
      say "Processed $filename successfully";
   }
   catch ($e) {
      say "Failed to process $filename: $e";
   }
}
```

This combines Course 1, Chapter 8's file-handling idioms directly with modern `try`/`catch` — the `open(...) or die` pattern still triggers an error exactly as before, but now it's caught and handled gracefully by the surrounding `try`/`catch` instead of terminating the whole program.

Error Handling: Perl vs. Python/JavaScript

Perl's `die`/`eval`/`$@` conceptually parallels Python's `raise`/`except` and JavaScript's `throw`/`try`/`catch` — all three "raise an exception, catch it somewhere up the call stack." Historically, Perl's `eval`-block approach was often described as a "poor man's" version, since it reused a block form originally meant for something else and relied on a global variable (`$@`) for the error itself. The modern `try`/`catch` syntax genuinely converges Perl with the dedicated exception-handling syntax Python and JavaScript have always had — cleaner, and no more global-variable clobbering concerns.

die

Raises an error; a trailing `\n` suppresses the automatic file/line info.

eval { } / $@

The classic catch mechanism — capture `$@` immediately to avoid clobbering.

try / catch ($e)

Modern (5.34+), cleaner syntax — `$e` is safely scoped, no globals involved.

die with a reference

Structured exceptions — `$@`/`$e` becomes the reference itself.

💻 Coding Challenges

Challenge 1: Catch a Division Error

Write a subroutine `safe_divide($a, $b)` that dies with `"Cannot divide by zero\n"` if `$b` is 0. Call it inside a `try`/`catch` block with `$b` as 0, printing the caught error.

→ Solution

Challenge 2: The $@ Capture Pattern

Write an `eval { }` block that dies with a message, immediately capture `$@` into `my $err`, and print it. Explain in one sentence why capturing it immediately is safer than checking `$@` directly later.

→ Solution

Challenge 3: A Structured Exception

Write a subroutine that dies with a hashref `{ code => 400, message => "Bad input" }` when given a negative number. Catch it and print both the code and message separately.

→ Solution

🎯 What's Next

Next chapter: **Closures & Functional Techniques** — anonymous subs, closures over lexicals, and `map`/`grep`/`sort` with custom blocks.

🔒 Closures & Functional Techniques

Course 1, Chapter 7 covered named subroutines. This chapter covers *anonymous* ones — and the genuinely powerful closure mechanism they unlock — alongside `map`/`grep`/`sort`, Perl's core tools for functional-style list processing.

🕶️ Anonymous Subroutines

```
my $add = sub {
    my ($a, $b) = @_;
    return $a + $b;
};

say $add->(3, 4);   # 7 — calling a coderef uses the arrow, like Course 1, Chapter 9's dereferencing
```

`sub { ... }` with no name creates a **coderef** — a reference to a subroutine, storable in a scalar exactly like the array/hash references from Course 1, Chapter 9. Calling it uses the same `->` arrow syntax.

🔐 Closures — Capturing Lexical Variables

An anonymous sub **closes over** any lexical (`my`) variables from its surrounding scope — it doesn't just read their value once, it remembers the variable itself, even after the scope that declared it has technically ended:

```
sub make_counter {
    my $count = 0;
    return sub { return ++$count; };   # this anonymous sub closes over $count
}

my $counter1 = make_counter();
my $counter2 = make_counter();

say $counter1->();   # 1
say $counter1->();   # 2
say $counter2->();   # 1 — a completely independent $count, not shared with $counter1
```

Each call to `make_counter()` creates a genuinely fresh `$count`, and the anonymous sub returned each time closes over *that specific* variable — `$counter1` and `$counter2` maintain entirely separate, independent state, even though they were built from the exact same code.

🎯 Practical Uses of Closures

```
sub make_multiplier {
    my ($factor) = @_;
    return sub {
        my ($n) = @_;
        return $n * $factor;
    };
}

my $double = make_multiplier(2);
my $triple = make_multiplier(3);

say $double->(5);   # 10
say $triple->(5);   # 15
```

Closures are genuinely useful for: generating a family of related subroutines from one template (as above), private state without building a full class (Chapter 3/4's OOP) for something too simple to warrant it, and passing behavior around as a callback — a subroutine reference handed to another function to be invoked later.

🗺️ map — Transform Every Element

```
my @numbers = (1, 2, 3, 4);
my @squared = map { $_ * $_ } @numbers;
say "@squared";   # 1 4 9 16
```

`$_` is the implicit "topic" variable — the same default variable Chapter 6's bare `/pattern/` matches against — set to each element of `@numbers` in turn. `map` is more idiomatic than a manual `foreach`/`push` loop for a pure element-by-element transformation.

🔍 grep — Filter Elements

```
my @evens = grep { $_ % 2 == 0 } @numbers;
say "@evens";   # 2 4
```

`grep` keeps only the elements where the block evaluates true — Perl's built-in filtering function, named after the classic Unix command line tool it takes its name from.

🔀 sort with a Custom Block — Revisited

Chapter 3 (Course 1) showed `sort { $a <=> $b } @nums` for numeric ordering. The same mechanism sorts complex structures by any field:

```
my @people = (
    { name => "Ada",  age => 28 },
    { name => "Alan", age => 41 },
    { name => "Grace", age => 36 },
);

my @by_age = sort { $a->{age} <=> $b->{age} } @people;
```

Inside `sort`'s block, `$a` and `$b` hold two elements being compared — with hashrefs (Course 1, Chapter 9), accessing a specific field with `->{age}` before comparing sorts the whole structure by that field.

🔗 Chaining map/grep/sort Together

```
my @active_names_by_age =
  map  { $_->{name} }
  sort { $a->{age} <=> $b->{age} }
  grep { $_->{active} }
  @people;
```

Read from the bottom up: `grep` keeps only active people, `sort` orders what's left by age, `map` extracts just the names from the sorted result. This chained pipeline style is genuinely idiomatic Perl — filter, then order, then transform, each stage doing exactly one job.

⚠ Loop Variables and Closures — Less of a Footgun in Perl Than Elsewhere

In some languages, creating closures inside a loop is a classic trap — every closure ends up sharing the same loop variable, all seeing its final value. Perl's `foreach my $x (@list) { ... }` genuinely creates a fresh lexical `$x` on every single iteration, so a closure built inside the loop body correctly captures that iteration's own independent variable — the same trap other languages have doesn't naturally occur here, as long as you declare the loop variable with `my` (the standard form) rather than reusing an outer variable across iterations.

🔧 A Worked Example — Multiplier Factory + Data Pipeline

```
my $to_percent = make_multiplier(100);

my @ratios = (0.25, 0.5, 0.75);
my @percentages = map { $to_percent->($_) } @ratios;
say "@percentages";   # 25 50 75
```

This combines both halves of the chapter directly: `$to_percent` is a closure built from Chapter 7's factory pattern, and `map` applies it across a whole list in one line — functional composition, Perl-style.

Closures: Perl vs. JavaScript vs. Python

Perl and JavaScript closures behave almost identically — both freely capture *and mutate* the captured variable, exactly like this chapter's counter example. Python closures are more restrictive by default: a nested function can *read* an enclosing variable, but reassigning it requires the explicit `nonlocal` keyword — without it, Python treats the assignment as creating a new local variable instead. Perl needs no such keyword; a closure captures the variable itself, not just its value, the same way JavaScript does.

Anonymous subs

`sub { ... }` — a coderef, called with `->()`.

Closures

Capture the actual lexical variable, not just its value — independent per creation.

map / grep

Transform every element / keep only matching elements, using `$_`.

sort with $a/$b

Custom comparators sort by any field, including inside complex structures.

💻 Coding Challenges

Challenge 1: An Independent Counter Pair

Using this chapter's `make_counter`, create two separate counters, call the first one three times and the second one once, and print both counters' current values to prove they're independent.

→ Solution

Challenge 2: Filter and Transform

Given `my @numbers = (1..10);`, use `grep` to keep only numbers greater than 5, then `map` to double each remaining number, printing the final list.

→ Solution

Challenge 3: Sort an Array of Hashrefs by Name

Given this chapter's `@people` array, use `sort` to order it alphabetically by name (using `cmp`, Course 1, Chapter 5's string comparison operator), then `map` to print just the sorted names.

→ Solution

🎯 What's Next

Next chapter: **Testing with Test::More** — `ok`/`is`/`like` assertions, `prove`, and Perl's real testing culture.

🧪 Testing with Test::More

Perl has one of the oldest, most established testing cultures in programming — TAP (Test Anything Protocol), the output format `Test::More` produces, originated in Perl and was later adopted as a language-agnostic standard used by testing tools well beyond Perl itself. This chapter covers writing and running real tests.

✅ The Basic Assertions

```
use Test::More;

ok(1 + 1 == 2, 'basic math works');         # true/false — the most basic assertion
is(2 + 2, 4, 'addition is correct');        # equality — reports BOTH values on failure
isnt(2 + 2, 5, 'addition is not wrong');    # inequality
like("Hello World", qr/World/, 'contains World');  # regex match — reuses Chapter 1's m// and qr//
```

`is()` is preferred over `ok($got == $expected, ...)` for equality checks specifically because a failure reports *both* the actual and expected values automatically — genuinely more useful for diagnosing what went wrong than a bare pass/fail from `ok` alone.

📝 A First Test File

```
# t/basic.t
use strict;
use warnings;
use Test::More;

is(1 + 1, 2, 'one plus one is two');
ok("perl" eq "perl", 'strings match');

done_testing();   # modern style — no need to declare a plan count up front
```

Test files conventionally live in a `t/` directory and end in `.t`. `done_testing();` at the end is the modern convention, replacing the older style of declaring `use Test::More tests => 5;` and having to keep that count manually in sync with the actual number of assertions.

▶️ Running Tests with prove

```
prove t/basic.t     # run one test file
prove -r t/         # recursively run every .t file in the t/ directory

t/basic.t .. ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs
```

`prove` (installed alongside Perl itself) is the standard test runner — it executes every matched `.t` file and reports a clean pass/fail summary.

🧬 Testing Data Structures

```
my $got      = { name => "Ada", age => 28 };
my $expected = { name => "Ada", age => 28 };

is_deeply($got, $expected, 'hashref contents match');   # PASSES — compares contents, not identity
```

⚠ is() on References Checks Identity, Not Contents

`is($got, $expected, ...)` on two references — even two hashrefs holding identical data — compares whether they're the exact *same* reference in memory (Course 1, Chapter 9's reference-vs-value distinction), not whether their contents match. Two separately-built hashrefs with identical keys and values will **fail** a plain `is()` check. `is_deeply()` is the correct tool for comparing the actual contents of nested arrays/hashes, recursively.

🔧 A Worked Example — Testing a Real Subroutine

```
# t/multiplier.t
use strict;
use warnings;
use Test::More;

sub make_multiplier {   # reused from Chapter 7
my ($factor) = @_;
   return sub { my ($n) = @_; return $n * $factor; };
}

my $double = make_multiplier(2);

is($double->(5), 10, 'double(5) is 10');
is($double->(0), 0, 'double(0) is 0');
like("Result: " . $double->(3), qr/\d+/, 'result contains a number');

done_testing();
```

This exercises a closure (Chapter 7) directly through `Test::More`'s assertions — `is()` for the expected numeric results, `like()` confirming the output's shape via regex rather than an exact string match.

TAP: Perl's Testing Format, Adopted Widely

TAP (Test Anything Protocol) — the simple, line-based pass/fail output `Test::More` produces — originated in Perl's testing culture and was later formalized as a language-agnostic standard, with TAP producers and consumers built for many other languages beyond Perl. This is a genuine parallel to CPAN's own historical head start (Chapter 5): Perl's testing tooling matured early enough to influence testing practices well outside the language itself, much like PCRE's regex engine did.

ok / is / isnt / like

The core assertions — `is()` for equality reports both values on failure.

done_testing()

Modern style — no upfront test-count plan to keep in sync.

prove

The standard test runner — `prove -r t/` runs every test file.

is_deeply

Compares reference contents recursively — `is()` alone only checks identity.

💻 Coding Challenges

Challenge 1: A Basic Test File

Write a test file `t/strings.t` with at least 3 assertions covering string concatenation, a regex match with `like`, and one intentionally-failing assertion (to see what a failure looks like) — then remove the failing one and confirm all tests pass with `prove`.

→ Solution

Challenge 2: Test a Subroutine From Course 1

Write a test file testing Course 1, Chapter 7's `power` subroutine (base and optional exponent, defaulting to 2) with at least 3 `is()` assertions covering both the default and an explicit exponent.

→ Solution

Challenge 3: is vs is_deeply

Given two separately-built arrayrefs both containing `(1, 2, 3)`, write one assertion using `is()` (predict and note that it fails) and one using `is_deeply()` (predict and note that it passes), and explain in one sentence why they disagree.

→ Solution

🎯 What's Next

Final chapter: **Perl One-Liners & Practical Text Processing** — `perl -e`/`-pe`/`-ne`, closing the loop back to the Bash/sed/awk cluster.

⚡ Perl One-Liners & Practical Text Processing

The final chapter — and where this whole course's opening framing pays off directly. Course 1, Chapter 1 introduced Perl as "what you reach for once shell text-processing hits its ceiling," positioned against the site's own Bash/SED/AWK/`regex1` cluster. This chapter shows Perl competing on sed's and awk's own turf: the command-line one-liner.

▶️ perl -e — Running Code Without a Script File

```
perl -e 'print "Hello\n";'
perl -e 'print 2**10, "\n";' # 1024 — a quick calculation, no .pl file needed at all
```

`-e` runs the given code directly, no script file required — genuinely useful for a quick calculation or test without leaving the terminal.

🔁 perl -n — Implicit Loop Over Input Lines

```
perl -ne 'print if /error/;' access.log
# equivalent to: grep 'error' access.log
```

`-n` wraps your code in an implicit `while (<>) { ... }` loop — one iteration per input line, without writing that loop yourself. `<>` is the "diamond operator" — the generic version of Course 1, Chapter 8's `<$fh>`, reading from whatever files are named on the command line, or standard input if none are given. Inside the loop, each line is available as `$_`, the same default variable from Chapter 6's bare regex matches and Chapter 7's `map`/`grep`.

🖨️ perl -p — Implicit Loop with Automatic Printing

```
perl -pe 's/foo/bar/;' file.txt
# equivalent to: sed 's/foo/bar/' file.txt
```

`-p` behaves like `-n`, but automatically prints `$_` at the end of every iteration — the go-to flag for line-by-line transformation, directly replicating what `sed 's/pattern/replacement/'` does, using Chapter 1's own `s///`.

💾 perl -i — In-Place Editing

```
perl -i.bak -pe 's/foo/bar/;' file.txt   # modifies file.txt directly, keeping file.txt.bak as a backup
```

⚠ -i With No Backup Suffix Is Irreversible

`perl -i -pe '...' file.txt` (no suffix after `-i`) overwrites the file directly, with **no undo** — a typo'd pattern can silently and permanently destroy real data. Always test the exact same command without `-i` first (it prints the transformed output to the terminal without touching the file), or run with `-i.bak` to keep an automatic backup, before ever running it for real against something you can't afford to lose.

🚩 Useful Command-Line Flags Combined

Flag	Effect
`-e`	Run code given directly on the command line
`-n`	Implicit line-reading loop, no automatic printing
`-p`	Implicit line-reading loop, WITH automatic printing
`-i[.suffix]`	Edit files in place, optionally keeping a backup
`-l`	Automatically handles line endings (chomps input, adds newline to output)
`-a`	Autosplits each line into `@F` by whitespace — genuinely awk-like field splitting
`-F`	Sets a custom field separator for `-a` (e.g. `-F,` for CSV)

```
perl -F, -lane 'print $F[1];' data.csv
# -a splits each line into @F by the -F, delimiter — prints the 2nd column of a CSV
```

⚔️ A Direct Comparison — Perl vs. sed vs. awk

Task	sed	awk	Perl
Replace text	`sed 's/foo/bar/' f`	`awk '{gsub(/foo/,"bar")}1' f`	`perl -pe 's/foo/bar/' f`
Print matching lines	`sed -n '/err/p' f`	`awk '/err/' f`	`perl -ne 'print if /err/' f`
Print 2nd column	(awkward in sed)	`awk -F, '{print $2}' f`	`perl -F, -lane 'print $F[1]' f`

Perl's one-liner syntax is more verbose than sed's for the simplest substitutions, but scales considerably better the moment a task needs anything sed/awk don't handle natively — real subroutines, complex data structures, or CPAN modules (Chapter 5) — without switching tools entirely.

🔧 A Worked Example — Practical One-Liners

```
# count lines matching a pattern:
perl -ne '$c++ if /ERROR/; END { print "$c\n" }' app.log

# extract email addresses from a file:
perl -ne 'print "$1\n" while /([\w.+-]+@[\w-]+\.[\w.-]+)/g;' contacts.txt

# number every line:
perl -ne 'print "$.: $_";' file.txt   # $. is the current input line number
# sum a column of numbers:
perl -lne '$sum += $_; END { print $sum }' numbers.txt
```

Every one of these reuses tools from earlier in this course directly — Chapter 6's regex captures, Chapter 4's/Chapter 2's context-driven accumulation, and the special `END { }` block (runs once, after all input has been processed) for a final summary.

One-Liner vs. Full Script vs. sed/awk — When to Reach for Which

A one-liner is right for a genuinely quick, one-off task at the terminal — something you'd otherwise open an editor for and immediately throw away. The moment a task needs to be run more than once, tested (Chapter 8), or exceeds a few lines of real logic, a proper script (everything else in this course) is the better tool — readable, maintainable, version-controllable. sed and awk remain excellent for the simplest cases they were purpose-built for; Perl one-liners are worth reaching for the moment a task needs slightly more power than either comfortably provides, without needing to abandon the one-liner workflow entirely.

-e

Run code directly, no script file.

-n / -p

Implicit line loop, with or without automatic printing.

-i[.bak]

Edit files in place — always use a backup suffix until confident.

-a / -F

Autosplit fields — Perl's genuinely awk-like column-processing mode.

💻 Coding Challenges

Challenge 1: Replace Text with -pe

Write a `perl -pe` one-liner that replaces every occurrence of "cat" with "dog" (case-insensitive) in a file, first WITHOUT `-i` to preview the change, then WITH `-i.bak` to apply it safely.

→ Solution

Challenge 2: Count and Print with -ne

Write a `perl -ne` one-liner that counts how many lines in a file contain the word "WARN", printing just the final count using an `END` block.

→ Solution

Challenge 3: Extract a CSV Column

Given a CSV file with columns `name,age,city`, write a `perl -F -lane` one-liner that prints just the `city` column (the 3rd field) for every row.

→ Solution

🎓 Course Complete!

That's all 9 chapters of **Perl Intermediate/Advanced** — advanced regex, context internals, classic and modern OOP, modules and CPAN, error handling, closures, testing, and one-liners — and with it, the complete Perl track: Fundamentals and Intermediate/Advanced, 18 chapters combining a genuinely practical, work-relevant language with the site's own broader Bash/SED/AWK/`regex1` text-processing cluster it was framed against from the very first chapter.