PERL

Fundamentals

Getting Started · Scalars & Data Types · Arrays & Lists
Hashes · Control Flow & Operators · Regular Expressions
Subroutines · File I/O & Text Processing · References

Philip Osztromok

Generated with Claude

Table of Contents

Perl Fundamentals — 9 Chapters

🐪 Getting Started

Welcome to Perl Fundamentals. Perl grew directly out of the same territory as Bash, sed, awk, and regular expressions — Larry Wall built it specifically because shell scripting and text-processing tools were hitting their limits. This chapter covers installing Perl, running a script, and the two pragmas that mark the difference between idiomatic Perl and the "write-only code" reputation the language is unfairly stuck with.

📥 Installing Perl

Most Linux and macOS systems already have Perl installed — check with:

```
perl -v
```

On Windows, Strawberry Perl is the standard distribution — a complete, self-contained installer including `cpanm` (Chapter-9-of-Course-2's package installer) out of the box.

▶️ Running a Perl Script

```
perl script.pl
```

On Linux/macOS, a script can also be made directly executable, using a **shebang line** as its first line:

```
#!/usr/bin/env perl
use strict;
use warnings;

print "Hello, World!\n";


chmod +x script.pl
./script.pl
```

`#!/usr/bin/env perl` is preferred over a hardcoded path like `#!/usr/bin/perl` — `env` finds whichever `perl` is first on the system's `PATH`, working correctly across different installations rather than assuming one fixed location.

🛡️ use strict; use warnings; — Non-Negotiable Culture

Perl is famously permissive by default — variables can be used without being declared, typos in variable names silently create a new variable rather than raising an error, and genuinely dangerous constructs are allowed without complaint. This permissiveness is exactly where Perl's "write-only code" reputation comes from. The fix, and the actual convention in every real Perl codebase, is two lines at the top of every script:

```
use strict;     # requires variables to be declared before use
use warnings;   # surfaces likely-bug situations (undefined values, etc.)
```

⚠ Without strict, a Typo Becomes a New Variable — Silently

```
# no "use strict" — this typo compiles and runs with NO error:
$user_name = "Ada";
print $usr_name;   # typo — prints nothing, since $usr_name is a brand new, empty variable
```

With `use strict;` in place, that same typo raises `Global symbol "$usr_name" requires explicit package name` at compile time — before the script ever runs — turning a silent, confusing bug into an immediate, precise error pointing at the exact line. There is essentially no legitimate reason to omit `use strict; use warnings;` from a real script.

🖨️ print and say

```
print "Hello, World!\n";   # no automatic newline — you add \n yourself
use feature 'say';
say "Hello, World!";      # say adds the newline automatically
```

`print` is the original, universal output function — but unlike `console.log` or Python's `print()`, it does *not* add a trailing newline; you write `\n` explicitly. `say` (available via `use feature 'say';`, or automatically with `use v5.10;` or later) behaves like `print` but appends the newline for you — the more ergonomic default for everyday scripts.

Comments

```
# a single-line comment — everything from # to the end of the line
=pod

This is a multi-line documentation block, using Perl's own
built-in documentation format, POD (Plain Old Documentation).
It's ignored by the interpreter but can be extracted by tools
into real formatted documentation.

=cut
```

POD (`=pod` ... `=cut`) is a distinctly Perl feature — a lightweight markup format for documentation, embeddable directly inside the same file as the code it documents, and extractable into HTML/man pages by standard tools. Ordinary `#` comments cover everything else.

Perl vs. Bash: Why Perl Exists

Perl was built specifically because Bash/sed/awk hit real limits once a script needs actual data structures, subroutines with proper scoping, or complex logic — Bash's "everything is a string" model gets unwieldy fast beyond simple pipelines. Perl keeps the same regex-heavy, text-processing instincts (Chapter 6 covers this directly) while adding real arrays, hashes, references, and subroutines on top — the reason it became the standard "glue language" for text-heavy scripting throughout the 1990s and 2000s, and still shows up constantly in legacy sysadmin and text-processing code today.

Shebang line

`#!/usr/bin/env perl` — makes a script directly executable.

use strict / use warnings

Mandatory in real code — catches typos and likely bugs at compile time.

print vs say

`print` needs an explicit `\n`; `say` adds one automatically.

Comments & POD

`#` for single lines; `=pod`/`=cut` for embedded documentation.

💻 Coding Challenges

Challenge 1: Your First Script

Write a script beginning with the correct shebang line and `use strict; use warnings;`, that prints `"Hello, Perl!"` using `say`.

→ Solution

Challenge 2: Catch the Typo

Given a short script with `use strict;` enabled that assigns to `$message` but then tries to print `$mesage` (missing an "s"), write down the exact compile-time error Perl would raise, and explain in one sentence why `use strict` is what causes an error here instead of silent wrong output.

→ Solution

Challenge 3: print vs say

Write two versions of a script that prints three separate lines — one using only `print` (correctly formatted with newlines), one using `say` — producing identical output.

→ Solution

🎯 What's Next

Next chapter: **Scalars & Basic Data Types** — the `$` sigil, strings/numbers, string interpolation, and `my`/`our`/`local` scoping.

💲 Scalars & Basic Data Types

Every Perl variable name is prefixed with a **sigil** — a symbol indicating what *kind* of variable it is, not what type of value it holds. This chapter covers the scalar sigil `$`, the two data types a scalar actually stores (strings and numbers), string interpolation, and the three ways Perl handles variable scope.

🏷️ The $ Sigil

A **scalar** — a single value, as opposed to a list of values — is always named with a leading `$`:

```
my $name = "Ada";
my $age = 28;
```

This is a genuinely distinctive design choice worth internalizing early: `$` always means "one scalar value," `@` (Chapter 3) always means "an array," `%` (Chapter 4) always means "a hash." The sigil tells you the variable's *kind* at a glance, everywhere in a script — even, as you'll see in Chapter 3, when reading a single element *out of* an array.

🔤 Scalar Values: Strings & Numbers

A scalar can hold a string, a number, or (Chapter 9) a reference — Perl doesn't require declaring which up front, and converts between string and numeric interpretations automatically depending on context.

```
my $count = 42;
my $price = 19.99;
my $big = 1_000_000;     # underscores allowed for readability, ignored by Perl
my $single = 'Raw text, no interpolation: $count';
my $double = "Interpolated: $count";
```

Single-quoted strings are **literal** — nothing inside them is interpreted except an escaped quote or backslash. Double-quoted strings are **interpolated** — variables and certain escape sequences (`\n`, `\t`) are expanded.

🧵 String Interpolation

```
my $name = "Ada";
say "Hello, $name!";       # Hello, Ada!  — $name expanded directly inside the string
my $greeting = "Hi, " . $name . "!";   # concatenation with the . operator
$greeting .= " Welcome.";          # .= appends in place
```

Interpolation directly inside a double-quoted string is idiomatic Perl and generally preferred over concatenation with `.` — it reads closer to the final output and avoids a chain of `.` operators for anything with more than one or two variables.

⚠ A Literal $ Inside a Double-Quoted String Needs Escaping

```
say "Price: $50";    # tries to interpolate a variable named $5, then a literal "0" — not what you want
say "Price: \$50";   # Price: $50 — the backslash escapes $, forcing it to be literal
```

Inside double quotes, `$` always signals "interpolate a variable here" — a genuine dollar sign has to be escaped as `\$`, or the string switched to single quotes if nothing else in it needs interpolating.

🔢 Numeric Operators & String-Number Duality

```
say 10 + 5;   # 15
say 10 - 5;   # 5
say 10 * 5;   # 50
say 10 / 5;   # 2
say 2 ** 8;   # 256 — exponentiation
say 10 % 3;   # 1 — modulo
say "10" + 5;  # 15 — the STRING "10" is auto-converted to a number in numeric context
```

This last line is a real, distinctive Perl behavior: a string that "looks like" a number is automatically converted when used in a numeric operation. It's convenient for data read from files or user input (always strings) — but relying on it with genuinely non-numeric strings produces `0` and, with `use warnings` active, a visible `"abc" isn't numeric` warning rather than a silent wrong answer.

🔭 my, our, and local — Scoping

Keyword	Scope	Use case
`my`	**Lexical** — visible only within the enclosing block	The default — almost every variable you declare should be `my`
`our`	**Package-scoped** — shared across the whole package/file	A genuine global, still requires declaring under `use strict`
`local`	**Dynamic** — temporarily overrides a global's value for the current block only, then restores it	Rare; mostly for temporarily changing a special/global variable

```
our $VERSION = "1.0";   # a package-wide global
sub show_version {
   local $VERSION = "1.0-debug";   # temporarily overrides $VERSION...
inner_function();           # ...and inner_function() sees the overridden value too
}  # ...restored back to "1.0" automatically once this block ends
```

💡 When in Doubt, Use my

`local` is easy to confuse with `my` since both look like they're "declaring a new variable" — but `local` only ever works on an existing package/global variable, temporarily saving and restoring its value. Real Perl code reaches for `my` for the overwhelming majority of variables; `local` is a specialized tool for a specific, fairly rare situation.

Sigils: Perl vs. Python/JavaScript

Python and JavaScript variables carry no prefix at all — `name = "Ada"`, `let name = "Ada"` — the variable's kind is inferred purely from context and its assigned value. Perl's sigil is visible everywhere the variable is used, not just where it's declared, which is part of why Perl code can look unfamiliar at first glance to someone coming from those languages — but it also means you can tell at a glance, mid-expression, whether you're looking at a single scalar, a whole array, or a whole hash.

$ sigil

Marks a scalar — a single string, number, or reference.

Quoting

Single quotes are literal; double quotes interpolate variables and escape sequences.

String-number duality

A numeric-looking string converts automatically in numeric context.

my / our / local

Lexical (default), package-global, and temporary-override scoping.

💻 Coding Challenges

Challenge 1: A Personalized Greeting

Declare two scalars, `$name` and `$city`, and use string interpolation in a single `say` call to print `"Ada lives in London."` (using whatever values you assign).

→ Solution

Challenge 2: Single vs Double Quotes

Given `my $x = 5;`, write one `say` statement using single quotes and one using double quotes, both attempting to print the phrase `The value is $x` — show the actual output of each, and explain why they differ.

→ Solution

Challenge 3: A Simple Calculator

Declare two numeric scalars and use `say` to print their sum, difference, product, and quotient, each on its own line, each labeled (e.g. `"Sum: 15"`).

→ Solution

🎯 What's Next

Next chapter: **Arrays & Lists** — the `@` sigil, array operations, and list vs. scalar context.

📚 Arrays & Lists

This chapter introduces one of Perl's most distinctive, initially confusing ideas early on purpose: **context**. The same array can mean "all its elements" or "how many elements it has" depending entirely on how it's used — arrays are the clearest first place to see this, and it colors nearly everything later in the language.

🏷️ The @ Sigil

```
my @fruits = ("apple", "banana", "cherry");
```

`@` marks an array — "more than one scalar, in order." But watch the sigil when accessing a *single* element:

```
say $fruits[0];   # apple — note: $, not @!
```

⚠ The Sigil Changes When You Access One Element

This trips up nearly everyone coming from another language: `@fruits` is the array, but `$fruits[0]` — a single element out of it — uses `$`, not `@`. This isn't inconsistent; it's the same rule from Chapter 2 applied precisely: the sigil describes *what you're asking for*. `$fruits[0]` asks for one scalar (found inside the array called `fruits`); `@fruits` asks for the whole array. The variable's underlying name is always `fruits` — only the sigil in front changes to match what you're requesting.

🛠️ Common Array Operations

```
push(@fruits, "date");      # add to the end
my $last = pop(@fruits);      # remove and return the last element
unshift(@fruits, "apricot");  # add to the beginning
my $first = shift(@fruits);    # remove and return the first element
say scalar(@fruits);        # the number of elements — forces scalar context (see below)
say $fruits[-1];           # the LAST element — negative indices count from the end
```

🔁 Iterating Arrays

```
foreach my $fruit (@fruits) {
    say $fruit;
}

# index-based, when you need the position too:
for my $i (0..$#fruits) {
    say "$i: $fruits[$i]";
}
```

`$#fruits` is the index of the *last* element of `@fruits` (not the count — one less than that) — `0..$#fruits` is the idiomatic range covering every valid index.

🎭 List vs. Scalar Context

The same array, written identically, behaves differently depending on the context it's evaluated in:

```
my @fruits = ("apple", "banana", "cherry");

my @copy  = @fruits;   # LIST context — @copy gets all 3 elements
my $count = @fruits;   # SCALAR context — $count gets 3, the COUNT
```

Assigning to an array (`@copy = ...`) puts the right-hand side in **list context** — Perl treats `@fruits` as "all of its elements." Assigning to a scalar (`$count = ...`) puts it in **scalar context** — Perl treats the exact same `@fruits` as "how many elements does this have." Nothing about `@fruits` itself changed; only what surrounds it did.

💡 This Is the Beginning of a Bigger Idea

Context isn't unique to arrays — it runs through much of Perl (Course 2's Context Deep Dive chapter covers it far more rigorously, including how your own subroutines can detect and respond to it). Arrays are simply the earliest, clearest place to encounter it: the exact same expression, two different meanings, decided entirely by what's asking for the value.

✂️ Array Slices

```
my @first_two = @fruits[0,1];   # a SLICE — note the @ sigil, since the result is a list
say "@first_two";               # apple banana
```

A slice pulls out *multiple* elements at once — and because the result is itself a list, the sigil is `@`, not `$`, even though it's written with square brackets like a single-element access.

🔀 Sorting and Reversing

```
my @words = sort @fruits;          # alphabetical (default: string comparison)
my @backwards = reverse @fruits;

my @numbers = (10, 2, 33);
say "@{[ sort @numbers ]}";        # 10 2 33 — WRONG if you wanted numeric order!
say "@{[ sort { $a <=> $b } @numbers ]}";  # 2 10 33 — correct numeric sort
```

⚠ sort Defaults to String Order, Not Numeric

This is the single most common Perl gotcha for beginners: `sort @numbers` compares elements as *strings* by default — `"10"` sorts before `"2"` because `"1"` is a "smaller" character than `"2"`, exactly like sorting words alphabetically. For actual numeric order, pass a comparator block explicitly: `sort { $a <=> $b } @numbers`. `<=>` (sometimes called the "spaceship operator") is the numeric comparison Perl needs here — the default string comparator uses `cmp` instead.

Arrays: Perl vs. Python

Perl arrays and Python lists cover similar ground — ordered, growable collections with `push`/`pop`-style operations on both sides. The real differences are Perl-specific: the sigil switches from `@` to `$` the moment you access a single element, and the same array genuinely evaluates differently depending on list vs. scalar context — Python's `len(my_list)` is an explicit function call with no equivalent ambiguity, where Perl's `my $count = @array;` relies entirely on context to mean "count."

@ vs $

`@array` is the whole array; `$array[0]` is one scalar element.

push/pop/shift/unshift

Add/remove from the end or beginning.

List vs. scalar context

The same array means "all elements" or "count," depending on what surrounds it.

sort's default is string order

Use `sort { $a <=> $b } @nums` for numeric sorting.

💻 Coding Challenges

Challenge 1: Build and Access an Array

Declare an array of 4 favorite foods. Print the first and last elements using `$array[0]` and negative indexing, and print the total count using `scalar(@array)`.

→ Solution

Challenge 2: List Context vs Scalar Context

Given `my @nums = (1, 2, 3, 4, 5);`, write one line assigning `@nums` to a new array (list context) and one line assigning it to a scalar (scalar context). Print both results and explain in one sentence why they differ.

→ Solution

Challenge 3: Numeric Sort

Given `my @scores = (85, 9, 100, 42);`, print the result of a plain `sort @scores` (demonstrating the string-sort gotcha), then print the correct numeric sort using a comparator block.

→ Solution

🎯 What's Next

Next chapter: **Hashes** — the `%` sigil, key-value operations, and iterating hashes.

🗂️ Hashes

The third and final sigil this course covers directly. A **hash** is Perl's key-value store — the same role a Python dict or JavaScript object plays — and it follows the exact same sigil-switching rule Chapter 3 established for arrays.

🏷️ The % Sigil

```
my %ages = (
    "Ada"   => 28,
    "Grace" => 85,
);
```

`=>` (the "fat comma") is functionally identical to a plain comma — it's used here purely for readability, visually pairing each key with its value. As a bonus, a bareword immediately to its left is automatically quoted, so `Ada => 28` works exactly like `"Ada" => 28`.

Accessing a single value uses curly braces and, once again, the `$` sigil — not `%`:

```
say $ages{"Ada"};   # 28 — one scalar value out of the hash, so $
```

This is the exact same rule from Chapter 3's `$fruits[0]`, just with curly braces instead of square brackets: `%ages` is the whole hash, `$ages{"Ada"}` is one scalar value pulled out of it.

🛠️ Common Hash Operations

```
$ages{"Alan"} = 41;       # add a new key, or overwrite an existing one
if (exists $ages{"Ada"}) {   # checks whether the KEY exists — even if its value is 0 or undef
say "Found Ada";
}

delete $ages{"Ada"};      # removes the key entirely
my @names  = keys %ages;    # an array of every key
my @years  = values %ages;  # an array of every value
```

⚠ exists Checks the Key, Not the Value's Truthiness

`if ($ages{"Bob"})` and `if (exists $ages{"Bob"})` are genuinely different questions. If `$ages{"Bob"}` is `0` (a valid age for a newborn, technically) or `undef`, the first check is false even though the key is really there — `exists` answers "is this key present at all," independent of whatever value it happens to hold.

🔁 Iterating Hashes

```
foreach my $name (keys %ages) {
    say "$name is $ages{$name}";
}
```

`foreach ... keys %hash` is the standard, idiomatic way to iterate — loop over the keys, look up each value as needed. Perl also has an `each` function (`while (my ($name, $age) = each %ages) { ... }`) that returns key/value pairs one at a time, but it has real gotchas if the hash is modified mid-loop — `keys`-based iteration is the safer, more common idiom in real code.

⚠ Hash Order Is Not Guaranteed

Unlike Python 3.7+ dictionaries (which guarantee insertion order), Perl hashes have **no** guaranteed order at all — `keys %ages` can return the keys in a different order between runs, or even between two calls in the same run in older Perl versions. If a specific, predictable order matters, sort explicitly: `foreach my $name (sort keys %ages)`.

🔧 A Worked Example — Word Frequency Count

```
my @words = split(' ', "the cat sat on the mat the cat ran");
my %counts;

foreach my $word (@words) {
   $counts{$word}++;    # auto-vivifies to 0 the first time, then increments
}

foreach my $word (sort keys %counts) {
   say "$word: $counts{$word}";
}
# cat: 2
# mat: 1
# on: 1
# ran: 1
# sat: 1
# the: 3
```

`$counts{$word}++` works even the very first time a given word is seen — an unset hash value auto-vivifies to `undef`, which `++` treats as `0` before incrementing, so no separate "does this key exist yet?" check is needed first.

Hashes: Perl vs. Python

Perl hashes and Python dicts serve the identical role — a Python developer will recognize the concept instantly. The differences are the same two threads running through this whole course: the sigil switches from `%` to `$` the moment you access one value (`$hash{key}`, vs. Python's uniform `my_dict[key]`), and — genuinely worth remembering — Perl hashes have never guaranteed any particular order, where Python's dicts have guaranteed insertion order since 3.7.

% vs $

`%hash` is the whole hash; `$hash{key}` is one scalar value.

exists vs truthiness

`exists $hash{key}` checks presence, independent of the value itself.

keys / values

Return arrays — unordered by default, sort explicitly when order matters.

Auto-vivification

`$hash{key}++` works even on a key that's never been set yet.

💻 Coding Challenges

Challenge 1: Build and Query a Hash

Declare a hash mapping 3 countries to their capital cities. Print one capital by looking it up directly, then print `"Found"` or `"Not found"` for a country that isn't in the hash, using `exists`.

→ Solution

Challenge 2: Sorted Iteration

Given a hash of 4 people mapped to their ages, print each `"Name: Age"` pair sorted alphabetically by name, using `sort keys`.

→ Solution

Challenge 3: exists vs Truthiness

Given `my %inventory = ("apples" => 5, "bananas" => 0);`, show that `if ($inventory{"bananas"})` and `if (exists $inventory{"bananas"})` behave differently, and explain why in one sentence.

→ Solution

🎯 What's Next

Next chapter: **Control Flow & Operators** — if/unless, while/until, for/foreach, and Perl's postfix conditionals.

🔀 Control Flow & Operators

Perl's control flow will look familiar coming from almost any C-family language — but two things are genuinely distinctive: postfix conditionals, which show up constantly in real Perl code, and a comparison-operator system that's split cleanly in two depending on whether you're comparing numbers or strings.

🔷 if / elsif / else

```
if ($age < 13) {
    say "child";
} elsif ($age < 20) {
    say "teenager";
} else {
    say "adult";
}
```

One firm Perl rule worth knowing up front: curly braces are *always* required, even for a single-statement body — unlike C or JavaScript, there's no bare `if (x) doSomething();` form. This isn't a style choice; leaving the braces off is a syntax error.

🚫 unless — Perl's "if not"

```
unless ($is_valid) {
    say "Invalid input";
}
# equivalent to: if (!$is_valid) { ... }
```

`unless` reads naturally for a genuinely negative condition. It's best kept simple, though — `unless (...) { ... } else { ... }` compiles fine but reads confusingly ("unless X, do this, else do that" inverts twice in your head); reach for a plain `if`/`else` the moment an `else` branch enters the picture.

➡️ Postfix Conditionals — a Distinctive Perl Style

```
say "Warning: low stock" if $stock < 5;
return unless $is_valid;
$total += $price if $in_stock;
```

Attaching `if`/`unless` to the *end* of a simple statement is genuinely idiomatic Perl — you'll see this constantly in real production code, not just as a trick. It reads almost like natural English ("return unless valid"), and for a single guarded statement it's usually clearer than a full multi-line block just to wrap one line.

⚖️ Comparison Operators — Numeric vs. String

This is a real, distinctive Perl design decision: **numeric** and **string** comparisons use entirely separate sets of operators.

Comparison	Numeric	String
Equal	`==`	`eq`
Not equal	`!=`	`ne`
Less than	`<`	`lt`
Greater than	`>`	`gt`
Less/equal	`<=`	`le`
Greater/equal	`>=`	`ge`

```
say "10" == "10.0" ? "equal" : "not equal";   # equal — compared NUMERICALLY, 10 == 10.0
say "10" eq "10.0" ? "equal" : "not equal";   # not equal — compared as STRINGS, "10" != "10.0"
```

⚠ Using == on Non-Numeric Strings Compares 0 == 0

Because `==` always forces both sides into numeric context (Chapter 2's string-number duality), comparing two genuinely non-numeric strings with `==` converts *both* to `0` first — `"apple" == "banana"` is `true`, since both sides become `0`, and (with `use warnings` active) you'll see an `isn't numeric` warning pointing at the mistake. Use `eq` for comparing text, always — reaching for `==` out of habit from another language is one of the most common early Perl bugs.

🔁 Loops — while, until, for, foreach

```
my $i = 0;
while ($i < 5) {
    say $i;
    $i++;
}

until ($i == 0) {   # until = "while not" — loops while the condition is FALSE
$i--;
}

for (my $j = 0; $j < 5; $j++) {   # classic C-style for
say $j;
}

foreach my $fruit (@fruits) {        # from Chapter 3
say $fruit;
}
```

Loop control works the same way inside any of these: `last` exits the loop entirely (like `break` elsewhere), `next` skips to the next iteration (like `continue`), and the rarely-needed `redo` restarts the current iteration from the top without re-checking the condition.

🔗 Logical Operators — Two Sets, Different Precedence

Meaning	High precedence (C-style)	Low precedence (English)
AND	`&&`	`and`
OR	`||`	`or`
NOT	`!`	`not`

```
open(my $fh, '<', $filename) or die "Can't open $filename: $!";
```

This is a genuinely common Perl idiom, and it's precisely why the low-precedence `or` exists rather than being redundant with `||`: `or`'s low precedence means the whole `open(...)` call binds tighter than the `or`, so this line reads as "(try to open the file) or (die with this message)" — using `||` here instead can bind more tightly than intended around the assignment inside `open`'s arguments, a real gotcha in exactly this pattern. As a rule of thumb: use `&&`/`||` inside expressions and conditions; reach for `and`/`or` for this control-flow-like "do this or bail out" idiom.

🔧 A Worked Example

```
my $username = "ada99";
my $age = 28;

say "Username too short" unless length($username) >= 3;
say "Adult" if $age >= 18;
say "Name check passed" if $username eq "ada99";   # eq — a string comparison
```

Comparison Operators: Perl vs. Python/JavaScript

Python and JavaScript use one `==` (or `===`) for everything, letting the runtime figure out whether it's comparing numbers or strings. Perl deliberately splits this into two explicit operator sets — more to type, but it means the comparison you write always says exactly what kind of comparison you meant, rather than relying on implicit type coercion to guess correctly. The trade-off Perl makes: explicitness over brevity, and a real trap (the warn-box above) for anyone who forgets which set to reach for.

if / unless

Braces always required; `unless` reads well alone, awkwardly with `else`.

Postfix conditionals

`statement if condition;` — genuinely idiomatic, common in real code.

== vs eq

Numeric and string comparisons use entirely separate operators.

&&/|| vs and/or

Same meaning, different precedence — `or` for the "do this or die" idiom.

💻 Coding Challenges

Challenge 1: Postfix Conditionals

Given a scalar `$temperature`, use postfix conditionals (no full `if` blocks) to print `"Freezing"` if it's at or below 0, and `"Hot"` if it's at or above 30.

→ Solution

Challenge 2: Numeric vs String Comparison

Given `my $a = "07"; my $b = "7";`, write one comparison using `==` and one using `eq`, print both results, and explain in one sentence why they disagree.

→ Solution

Challenge 3: The open-or-die Idiom

Write a line attempting to open a file that doesn't exist, using the `open(...) or die "...";` idiom from this chapter, and explain in one sentence why `or` is used here instead of `||`.

→ Solution

🎯 What's Next

Next chapter: **Regular Expressions in Perl** — `m//`, `s///`, `tr///`, and capture groups, contrasted directly with the site's own `regex1` course.

🔍 Regular Expressions in Perl

Regex isn't a library you import in Perl — it's built directly into the language's own syntax, as fundamental as `if` or `foreach`. This is a real reason Perl exists at all (Chapter 1): PCRE, the regex flavor countless other languages and tools eventually adopted, is literally modeled after Perl's own. This chapter covers matching, substitution, and transliteration — and closes by properly solving the real HTML-email-parsing problem from earlier in this course's own development.

🎯 Matching with m//

```
if ($line =~ m/error/) {
    say "Found an error";
}

# the m is optional when using / as the delimiter — this is identical:
if ($line =~ /error/) {
    say "Found an error";
}

if ($line !~ /error/) {   # !~ — negated match
say "No error here";
}
```

`=~` is the **binding operator** — it applies the regex on its right to the scalar on its left. Without it, a bare `/pattern/` implicitly matches against `$_` (Perl's default variable), which is genuinely useful in short scripts and one-liners (Course 2's closing chapter) but worth binding explicitly with `=~` in real code for clarity.

🎣 Capture Groups

```
my $line = "Order #4471 shipped on 2026-07-10";

if ($line =~ /Order #(\d+)/) {
    say $1;   # 4471 — captured groups land in $1, $2, $3... automatically
}
```

Parentheses in the pattern create **capture groups**, and — distinctly Perl — the captured text lands directly in the global variables `$1`, `$2`, and so on, immediately after a successful match. This is a different model from Python's `match.group(1)` object-based approach: no match object to hold onto, just plain scalars populated as a side effect of the match.

```
my $log = "2026-07-10 14:32:05 ERROR Connection timeout";

if ($log =~ /(\d{4}-\d{2}-\d{2}) (\d{2}:\d{2}:\d{2}) (\w+) (.+)/) {
    my ($date, $time, $level, $message) = ($1, $2, $3, $4);
    say "$level at $time: $message";
}
```

Assigning `($1, $2, $3, $4)` into named variables immediately after the match is standard practice — `$1`-style variables get overwritten by the *next* successful match anywhere in the script, so capturing them into your own named scalars right away avoids them silently changing out from under you later.

🔁 Substitution with s///

```
my $text = "I like cats";
$text =~ s/cats/dogs/;      # modifies $text in place — now "I like dogs"
my $new_text = $text =~ s/dogs/birds/r;   # /r — non-destructive, returns a NEW string, leaves $text unchange
d
```

`s///` substitutes matched text — by default modifying the variable it's applied to directly. The `/r` modifier flips this: instead of modifying in place, it returns a modified *copy*, leaving the original untouched — useful whenever you want to keep the original value around too.

🔡 Transliteration with tr///

```
my $text = "Hello World";
(my $upper = $text) =~ tr/a-z/A-Z/;   # HELLO WORLD — character-by-character replacement
my $vowel_count = ($text =~ tr/aeiouAEIOU//);   # tr with no replacement just COUNTS matches
say $vowel_count;   # 3
```

`tr///` is genuinely different from `s///` — it's not pattern matching at all, just direct character-for-character mapping (the first character in the left set maps to the first in the right set, and so on). It's also, as a side effect, the fastest way to *count* occurrences of a set of characters: called in scalar context with an empty replacement, it returns the count without modifying anything.

🚩 Common Regex Modifiers

Modifier	Effect
`/g`	Global — match/replace every occurrence, not just the first
`/i`	Case-insensitive matching
`/x`	Extended — ignores whitespace/comments in the pattern, for readability
`/m`	Multiline — `^`/`$` match at the start/end of each line, not just the whole string
`/s`	Single-line — lets `.` match a newline character too

⚠ /g Behaves Differently in List vs. Scalar Context

This is a genuinely deep Perl regex behavior, worth knowing exists even before mastering it fully (Course 2's Advanced Regular Expressions chapter goes further): in **list context**, `/pattern/g` returns *every* match (or every capture group) as a list, all at once. In **scalar context**, the exact same `/g` match instead becomes a **stateful iterator** — each call advances through the string and returns one match at a time, remembering its position between calls. Same regex, same flag, genuinely different behavior depending on context (Chapter 3's theme, again) — a real source of confusion if you're not expecting it.

🔧 Solving the Real Problem — Extracting a Job Reference from HTML Email

This is the exact HTML-in-an-email-body problem worked through earlier: a job reference number that's sometimes plain text, sometimes wrapped in HTML markup like `<strong>` tags. The robust fix — strip tags into a plain-text copy, then run one simple pattern — written properly, in context:

```
use strict;
use warnings;
use feature 'say';

my $eml_body = "Job Reference:</strong> JO-2405-98673";

(my $plain_body = $eml_body) =~ s/<[^>]+>//g;   # strip every HTML tag into a plain-text copy
if ($plain_body =~ /Job Reference(?: Number)?:\s*(JO-\d+-\d+)/) {
   my $job_ref = $1;
   say "Found: $job_ref";   # Found: JO-2405-98673
}
```

Every piece here is from this course: `(my $plain_body = $eml_body) =~ s/.../.../g` assigns and substitutes in one line — the parentheses make `=~` apply to the freshly-assigned `$plain_body`, not `$eml_body`, leaving the original untouched. `(?: Number)?` is a non-capturing group (a group used purely for structure, not one that populates `$1`) marking "Number" as optional, so the pattern matches both `"Job Reference:"` and `"Job Reference Number:"`. And `\d+` (not `\d*`) requires at least one digit in each number group — the exact fix from that earlier debugging session, now with the reasoning fully spelled out.

Perl Regex vs. regex1's grep/sed/awk Regex

The site's own *Learning Regular Expressions* course covers grep/sed/awk/Bash regex — POSIX BRE/ERE, generally the more limited of the major regex flavors (no lookaround assertions without extensions, more restricted syntax overall). Perl's regex engine is the origin of PCRE (Perl Compatible Regular Expressions) — the flavor that eventually became the de facto standard adopted by countless other languages and tools specifically *because* Perl's regex support was so far ahead: native lookahead/lookbehind, named captures, and far richer pattern syntax, all built directly into the language rather than bolted on.

m// and =~

The binding operator applies a pattern; `m` is optional with `//` delimiters.

$1, $2, $3...

Capture groups land directly in numbered global variables after a match.

s/// and /r

Substitutes in place by default; `/r` returns a non-destructive copy.

tr///

Character-by-character replacement or counting — not pattern matching.

💻 Coding Challenges

Challenge 1: Extract an Order Number

Given `my $text = "Your order ORD-88213 has shipped";`, use `m//` with a capture group to extract just `ORD-88213` into `$1`, and print it.

→ Solution

Challenge 2: A Second Reference Format

Given `my $eml_body = "<b>Reference Number</b>: JO-2601-11029";` (a different tag and slightly different label than this chapter's own example), adapt this chapter's strip-then-match approach so it still correctly extracts `JO-2601-11029`.

→ Solution

Challenge 3: Case-Insensitive Replace-All

Given `my $text = "Cat sat with the cat and a CAT";`, use `s///` with the `/g` and `/i` flags together to replace every occurrence of "cat" (regardless of case) with "dog", and print the result.

→ Solution

🎯 What's Next

Next chapter: **Subroutines** — `sub`, `@_`, argument passing, and context-sensitive return values.

🔧 Subroutines

Perl subroutines take arguments differently from most languages you may already know — no named parameters, no positional slots defined up front. Every argument arrives flattened into one array, and it's up to the subroutine itself to unpack it. This chapter also revisits Chapter 3's context theme in its sharpest form yet: a subroutine can literally behave differently depending on how it was called.

📐 Defining and Calling a Subroutine

```
sub greet {
    say "Hello!";
}

greet();   # Hello!
```

You may occasionally see the older `&greet();` call syntax in legacy code — it still works, but plain `greet();` is the standard modern form and what real code should use.

📦 @_ — Arguments Arrive as a Single Array

```
sub add {
    my ($a, $b) = @_;   # unpack @_ into named scalars — the standard idiom
return $a + $b;
}

say add(3, 4);   # 7
```

Every argument passed to a subroutine — however many there are — lands in the array `@_`, exactly like Chapter 3's arrays. `my ($a, $b) = @_;` is a list assignment (also Chapter 3) unpacking the first two elements into named scalars — this line, or a close variant of it, is the first line of nearly every real Perl subroutine.

🔢 Default Values & Variable Argument Counts

```
sub greet_user {
    my ($name, $greeting) = @_;
    $greeting //= "Hello";   # //= — assign only if $greeting is undef
say "$greeting, $name!";
}

greet_user("Ada");           # Hello, Ada!  — $greeting was never passed, so it's undef, then defaulted
greet_user("Ada", "Hi");     # Hi, Ada!
say scalar(@_);   # inside any sub, tells you exactly how many arguments were passed
```

`//=` ("defined-or assign") only assigns if the left side is currently `undef` — the idiomatic way to supply a default for a missing argument, since a genuinely passed value of `0` or `""` (falsy, but *defined*) is correctly left alone, unlike the similar-looking `||=`, which would incorrectly overwrite those too.

Return Values

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

sub square_implicit {
    my ($n) = @_;
    $n * $n;   # no explicit return — the last evaluated expression is returned automatically
}
```

Both subroutines above behave identically. Perl, like Ruby, returns the value of the *last evaluated expression* automatically if no explicit `return` is reached — genuinely convenient for short subroutines, though an explicit `return` is usually clearer once a subroutine has more than one possible exit point.

🎭 Context-Sensitive Return — wantarray

This is Chapter 3's list-vs-scalar-context theme taken to its logical extreme: a subroutine can inspect *how it was called* and return something different accordingly, using `wantarray()`.

```
sub get_scores {
    my @scores = (85, 92, 78);

    if (wantarray()) {
        return @scores;          # called in LIST context — return every score
    } else {
        return scalar(@scores);  # called in SCALAR context — return just the count
    }
}

my @all   = get_scores();  # (85, 92, 78) — list context
my $count = get_scores();  # 3            — scalar context
```

`wantarray()` returns a true value if the subroutine was called in list context, a defined-but-false value in scalar context, and `undef` in void context (called for its side effects, with the return value discarded entirely) — three genuinely distinguishable answers, not just a boolean.

🔗 Passing Arguments by Reference (Preview)

⚠ @_ Aliases the Caller's Variables — It Doesn't Copy Them

```
sub double_it {
    $_[0] *= 2;   # modifies the CALLER's original variable, not a local copy!
}

my $x = 5;
double_it($x);
say $x;   # 10 — $x itself changed, even though it wasn't explicitly passed "by reference"
```

Unlike most languages, where arguments are copied into the function by default (pass-by-value), Perl's `@_` actually **aliases** the original variables passed in — modifying an element of `@_` directly modifies the caller's own variable. In practice this is rarely done deliberately (it's a surprising, easy-to-misuse behavior); the standard idiom of immediately unpacking with `my ($a, $b) = @_;` sidesteps it entirely, since that copies the values into fresh, independent lexical variables. Chapter 9's references cover the deliberate, explicit way to achieve pass-by-reference behavior when you actually want it.

🔧 A Worked Example

```
sub format_price {
    my ($amount, $currency) = @_;
    $currency //= "USD";

    return sprintf("%.2f %s", $amount, $currency);
}

say format_price(19.9);         # 19.90 USD
say format_price(19.9, "EUR");  # 19.90 EUR
```

Subroutines: Perl vs. Python

Python's `def f(a, b=1, *args, **kwargs)` bakes named parameters, defaults, and keyword arguments directly into the language's own function-definition syntax. Perl has none of that built in — every subroutine just receives one flat `@_`, and named-parameter-style calling is a convention built on top (commonly, passing a hash: `configure(name => "Ada", timeout => 30)`, then unpacking with `my %args = @_;` inside). More manual, but also more uniform — every subroutine's calling convention is the same flat array underneath, regardless of how the caller chooses to structure the arguments they pass.

@_

Every argument, flattened into one array — unpack with `my ($a, $b) = @_;`.

Implicit return

The last evaluated expression returns automatically without an explicit `return`.

wantarray

Detects list vs. scalar vs. void calling context, inside the subroutine itself.

@_ aliasing

Modifying `$_[0]` directly changes the caller's variable — unpack first to avoid it.

💻 Coding Challenges

Challenge 1: A Subroutine with a Default

Write a subroutine `power` that takes a number and an optional exponent (defaulting to 2 if not given, using `//=`), and returns the number raised to that power.

→ Solution

Challenge 2: Context-Sensitive Return

Write a subroutine `get_letters` that returns the list `('a', 'b', 'c')` in list context, and the count (3) in scalar context, using `wantarray`. Call it both ways and print both results.

→ Solution

Challenge 3: Demonstrate @_ Aliasing

Write a subroutine `reset_to_zero` that sets `$_[0] = 0`, call it on a variable holding 99, and show that the original variable changed. Then rewrite the subroutine to safely unpack its argument first (`my ($n) = @_;`) and show the original variable is no longer affected.

→ Solution

🎯 What's Next

Next chapter: **File I/O & Text Processing** — `open`/`close`, the `while (<FH>)` idiom, and Perl's original "practical extraction and report" purpose.

File I/O & Text Processing

"Perl" originally stood for the **P**ractical **E**xtraction and **R**eport **L**anguage — and this chapter is where that name pays off directly. Reading files line by line, pulling structured data out of raw text, and generating a report from it is the exact job Perl was built for, and it's very likely close to what you already reach for it for in real work.

📂 Opening and Closing a File

```
open(my $fh, '<', $filename) or die "Can't open $filename: $!";
# ... read from $fh ...
close($fh);
```

This is the modern, **3-argument form** of `open` — filehandle, mode, filename, each as separate arguments. The modes: `'<'` for reading, `'>'` for writing (truncates the file first — Course 4-equivalent caution applies), `'>>'` for appending. Always use the 3-argument form; the older 2-argument form (`open($fh, "< $filename")`) mixes the mode and filename into one string, which is a genuine security and correctness hazard if `$filename` ever contains unexpected characters.

⚠ Always Check open()'s Return Value

Chapter 5's `open(...) or die "...";` idiom isn't optional politeness — without it, a script that fails to open a file (wrong path, missing permissions) continues running anyway, silently doing nothing useful with an unopened filehandle. This is a genuinely common source of "my script ran with no errors but produced no output" confusion. Every `open` call should be followed by `or die` (or equivalent error handling — Course 2's Error Handling chapter covers alternatives).

🔁 Reading Line by Line — the while (<FH>) Idiom

```
open(my $fh, '<', 'access.log') or die "Can't open: $!";

while (my $line = <$fh>) {
    chomp($line);   # removes the trailing newline — $line includes it otherwise
say $line;
}

close($fh);
```

`<$fh>` reads one line at a time from the filehandle — this is the single most iconic Perl idiom, appearing in essentially every script that ever touches a file. Each line returned *includes* its trailing newline character; `chomp` removes exactly that one trailing newline (and only if present), which is why it's almost always the first thing done to a freshly-read line.

📥 Reading an Entire File at Once

```
open(my $fh, '<', $filename) or die "Can't open: $!";
local $/;              # undefine the input record separator — "slurp mode"
my $content = <$fh>;   # reads the ENTIRE file into one scalar
close($fh);
```

`$/` is the special variable controlling where a line "ends" — normally a newline. Temporarily undefining it (`local $/;`, using Chapter 2's `local` to restore it automatically afterward) makes `<$fh>` read the whole file in one call instead of one line at a time — the classic "slurp" idiom. Course 2's CPAN chapter covers `Path::Tiny`, a module that wraps this pattern into a single tidy method call.

✍️ Writing to a File

```
open(my $fh, '>', 'report.txt') or die "Can't open: $!";
print $fh "Report generated.\n";   # NO comma between $fh and the text!
close($fh);
```

⚠ No Comma Between the Filehandle and What You're Printing

`print $fh "text";` is correct — `print $fh, "text";` is a real, easy-to-make mistake. With the comma, Perl treats `$fh` as just another item in the list being printed to standard output, printing the filehandle's internal stringification followed by "text" — not writing to the file at all. This asymmetric-looking syntax (a filehandle directly followed by the thing to print, no separator) is one of Perl's odder corners, but it's exactly how `print` is told which filehandle to target.

🔧 A Worked Example — Processing a Log File

```
use strict;
use warnings;
use feature 'say';

sub count_errors {
    my ($log_file) = @_;
    my %error_counts;

    open(my $fh, '<', $log_file) or die "Can't open $log_file: $!";
    while (my $line = <$fh>) {
        if ($line =~ /(ERROR|WARN|INFO)/) {
            $error_counts{$1}++;
        }
    }
    close($fh);

    return %error_counts;
}

my %counts = count_errors('app.log');

open(my $out, '>', 'summary.txt') or die "Can't open: $!";
foreach my $level (sort keys %counts) {
    print $out "$level: $counts{$level}\n";
}
close($out);
```

This is a genuine, small end-to-end script combining nearly everything covered so far: a subroutine (Chapter 7) that reads a file line by line, uses regex capture groups (Chapter 6) to classify each line, tallies results in a hash (Chapter 4), and returns it context-appropriately — followed by writing a sorted summary report to a new file. This is exactly the shape of real, everyday Perl text-processing work.

File Handling: Perl vs. Python

Python's `with open(...) as f:` context manager guarantees the file closes automatically when the block ends. Perl's lexical filehandles (`my $fh`) have a similar safety net built in, less obviously: since `$fh` is just an ordinary lexical scalar, the file it holds open closes automatically once `$fh` itself goes out of scope — no special syntax required. That said, an explicit `close($fh);` remains standard practice in real code, since it makes the exact moment a resource is released clear to a reader, rather than relying on scope boundaries alone.

3-argument open

`open($fh, '<', $filename) or die ...;` — always check the result.

while (<$fh>)

Reads one line at a time; `chomp` strips the trailing newline.

Slurp mode

`local $/;` reads an entire file into one scalar in one call.

print $fh "..."

No comma between the filehandle and what's being printed.

💻 Coding Challenges

Challenge 1: Write Then Read

Write a script that opens a file for writing, writes 3 lines to it using `print $fh`, closes it, then reopens the same file for reading and prints each line (with `chomp` applied) using the `while (<$fh>)` idiom.

→ Solution

Challenge 2: Count Lines Matching a Pattern

Given a text file with several lines, write a subroutine `count_matching(filename, pattern)` that returns how many lines match a given regex pattern, using the `while (<$fh>)` idiom and `=~`.

→ Solution

Challenge 3: Fix the Missing or die

Given `open(my $fh, '<', 'does_not_exist.txt');` with no error handling, explain in one sentence what happens when this line runs against a genuinely missing file, then rewrite it correctly using this chapter's `or die` idiom.

→ Solution

🎯 What's Next

Next chapter: **References** — scalar/array/hash references, anonymous data structures, and arrow notation.

🔗 References

The final chapter of Perl Fundamentals — and a genuine bridge chapter. Everything so far has worked with flat scalars, arrays, and hashes. References are what let you build real nested data (an array of hashes, a hash of arrays — the shapes JSON and config files actually take), and they're the exact foundation Course 2's classic object-oriented Perl (`bless`, Chapter 3) is built directly on top of.

❓ What Is a Reference?

A **reference** is a scalar that points *to* another piece of data — an array, a hash, another scalar, even a subroutine — rather than holding that data directly. It's a similar idea to a pointer in C, but considerably safer: no pointer arithmetic, no manual memory management.

```
my @fruits = ("apple", "banana");
my $array_ref = \@fruits;   # \ creates a reference TO @fruits
my %ages = ("Ada" => 28);
my $hash_ref = \%ages;    # a reference to %ages
```

The backslash operator `\` creates a reference — `$array_ref` is a single scalar value that *points to* `@fruits`, it is not itself an array.

⚠ A Reference Is Not the Thing It Refers To

```
say $array_ref;   # ARRAY(0x55d1f8a2c3e0) — the reference's own string form, NOT the array's contents!
```

Printing a reference directly shows an internal identifier, not the data it points to — a common early confusion. To see the actual contents, you have to **dereference** it first, covered next.

👉 Dereferencing

```
# the classic syntax — wrap in the appropriate sigil and braces:
say join(", ", @{$array_ref});   # apple, banana
say ${$scalar_ref};

# the modern, idiomatic arrow syntax — used constantly in real code:
say $array_ref->[0];      # apple — one element, via ->
say $hash_ref->{"Ada"};    # 28
```

`->` is by far the more common style in real Perl — `$array_ref->[0]` reads naturally as "the reference, then index 0 of what it points to," and it's what you'll see in essentially every real codebase.

🕶 Anonymous Data Structures

```
my $fruits_ref = ["apple", "banana", "cherry"];   # [] creates an ANONYMOUS array ref directly
my $person_ref = { name => "Ada", age => 28 };   # {} creates an ANONYMOUS hash ref directly
```

Square brackets and curly braces used this way build a reference to a brand-new array or hash *with no named* `@array` *or* `%hash` *ever existing at all* — this is the standard way real code constructs nested structures, rather than declaring a separate named variable for every intermediate piece.

🏗️ Building Nested Data Structures

This is the actual payoff — combining anonymous references to build shapes that map directly onto real-world data, JSON responses, or config files:

```
# an array of hashes:
my @people = (
   { name => "Ada",  age => 28 },
   { name => "Alan", age => 41 },
);

say $people[0]{name};   # Ada — the arrow between [0] and {name} is optional here
# a hash of arrays:
my %teams = (
   red  => ["Ada", "Grace"],
   blue => ["Alan"],
);

say $teams{red}[0];    # Ada
```

➡️ Arrow Notation — When You Can Drop the Arrow

Perl allows omitting `->` *between two consecutive subscripts* — `$people[0]{name}` and `$people[0]->{name}` are exactly identical. The very first arrow (immediately after the reference variable) is the one that genuinely matters when it's not already implied by context; every subsequent one in a chain is optional stylistic sugar. Many real codebases keep every arrow for consistency and readability regardless — both styles are correct.

🔧 A Worked Example — Extending Chapter 8's Log Processor

```
my @entries = (
   { level => "ERROR", message => "Connection timeout" },
   { level => "WARN",  message => "Retrying request" },
   { level => "ERROR", message => "Disk full" },
);

foreach my $entry (@entries) {
   say "[$entry->{level}] $entry->{message}";
}
```

Where Chapter 8's log processor tallied plain counts into a flat hash, this version keeps each log entry as its own structured hashref inside an array — genuinely closer to how a real log parser or API response is shaped, and directly extensible (add a `timestamp` key, a nested `{context => {...}}` hashref, and so on) without restructuring anything already written.

References: Perl vs. Python

In Python, a variable holding a list or dict is *already* a reference under the hood — `b = a` for two lists means both names point to the same underlying list, with no explicit "reference" syntax anywhere. Perl makes this distinction fully explicit and visible: a plain `@array` is real data with value-copy semantics on assignment, while `\@array` is deliberately, visibly a reference. More to type, but it means you can always tell from the code itself, without knowing a type's internal behavior, whether you're holding data or a pointer to it.

\ creates a reference

`\@array`, `\%hash`, `\$scalar` — a scalar pointing to the original.

-> dereferences

`$ref->[0]`, `$ref->{key}` — the idiomatic modern syntax.

[ ] and { } build anonymous refs

No named variable required — the standard way to build nested structures.

Chained arrows are optional

Only the first arrow ever matters; later ones in a chain are sugar.

💻 Coding Challenges

Challenge 1: Build and Dereference

Create an array of 3 numbers, take a reference to it with `\`, then use the arrow syntax to print the second element and the total count (`scalar(@$array_ref)`).

→ Solution

Challenge 2: An Array of Hashes

Build an array of 3 anonymous hashrefs, each representing a product with `name` and `price` keys. Loop over the array and print each product formatted as `"Widget: $19.99"`.

→ Solution

Challenge 3: A Hash of Arrays

Build a hash mapping 2 department names to an anonymous array of employee names in each. Print every employee in the "Engineering" department using arrow-based dereferencing.

→ Solution

🎓 Course Complete!

That's all 9 chapters of **Perl Fundamentals** — getting started, scalars, arrays, hashes, control flow, regular expressions, subroutines, file I/O, and references. Next up: **Perl Intermediate/Advanced** (`perl2`), starting with Advanced Regular Expressions.