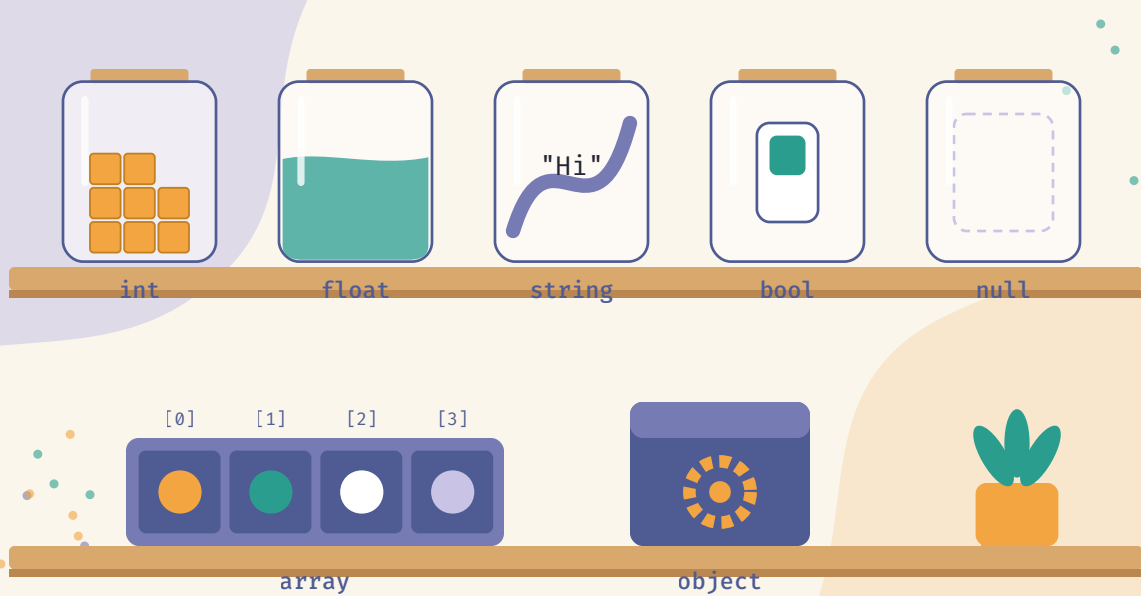




CHAPTER 3

Values and Types

Teaching PHP what things are

In Chapter 2 you installed PHP 8.5 and made it answer a request. Now we teach it to *think about data*. By the end of this chapter you will build a small command-line tool that prices a job for **Northstar Repairs**, the local handy-man business we follow through the whole book, and you will understand every value that flows through it.

What you will build

A script called `quote.php`. You give it a service, the minutes on site and the cost of parts, and it prints a tidy, VAT-inclusive estimate:

```
$ php quote.php shelf 90 1999
Labour      £72.00
Parts      £19.99
VAT         £18.40
Total      £110.39
```

Getting there means learning the building blocks every later chapter relies on:

- how PHP files are laid out: tags, statements and comments;
- the kinds of value PHP understands: integers, floats, strings, booleans, arrays, objects and `null`;
- how PHP compares and converts values, and where that goes wrong;
- how `declare(strict_types=1)` turns silent surprises into clear errors;
- two small PHP 8.5 additions, `array_first()` and the pipe operator `|>`.

NOTE How to run the examples

Every listing in this chapter is a complete file. Save it, then run it from a terminal with `php filename.php`. The output shown beneath each listing was produced by PHP 8.5.11 when this book was built, so what you see should match exactly. The files are also in the book's download, in `examples/chapter-03/`.

The shape of a PHP file

A PHP file is a plain text file, ending in `.php`, that PHP reads from top to bottom. Anything between `<?php` and `?>` is code; anything outside is sent out exactly as written. Here is the smallest useful script for Northstar Repairs:

LISTING `first-statements.php` statements, strings and comments

```
<?php
// A statement is one instruction. Each one ends with a semicolon.
$business = 'Northstar Repairs';
$callOutFee = 45;

echo 'Welcome to ' . $business . "\n";
echo "Call-out fee: £{$callOutFee}\n";

/*
 * A block comment can span several lines.
 * Use it to explain *why*, not to repeat *what*.
 */
```

OUTPUT

```
Welcome to Northstar Repairs
Call-out fee: £45
```

Three things to notice:

- 1. Every statement ends with a semicolon.** A statement is one complete instruction, such as “store this value” or “print that”. Forgetting the semicolon is the most common first error; PHP reports a “Parse error: syntax error, unexpected token” pointing at the *next* line, because that is where it realised something was missing.
- 2. Variables start with \$.** `$business` is a named box holding a value. You create it simply by assigning to it with `=`.
- 3. There is no closing `?>`.** In a file that contains only PHP, leave it off. Any stray blank line after a closing tag would be sent to the browser, which can break things like redirects later on.

Comments

PHP ignores comments; people do not. Use `//` for a note that runs to the end of the line and `/* ... */` for a longer block. (A `#` also starts a line comment, but `//` is the modern convention.) Good comments explain *why* a line exists, not *what* it does.

Mixing PHP into HTML

When PHP produces a web page, you often switch between HTML and code. The short echo tag `<?= ... ?>` prints a value and is always available, so use it freely in templates:

LISTING short-echo.php printing values into HTML safely

```
<?php
declare(strict_types=1);

$business = 'Northstar Repairs';
$tagline = 'Small jobs done properly & on time';
?>
<h1><?= htmlspecialchars($business, ENT_QUOTES, 'UTF-8') ?></h1>
<p><?= htmlspecialchars($tagline, ENT_QUOTES, 'UTF-8') ?></p>
```

OUTPUT

```
<h1>Northstar Repairs</h1>
<p>Small jobs done properly &amp; on time</p>
```

SECURITY Escape everything you print into HTML

`htmlspecialchars()` turned the `&` into `&`. That looks fussy for our own tagline, but the same habit stops a visitor typing `<script>` into a form and running code in your customers' browsers. We will wrap this call in a short helper in Chapter 13; for now, the rule is: **anything dynamic that goes into HTML goes through `htmlspecialchars()`**.

WATCH OUT Avoid the plain short open tag

You may see old code that begins with just `<?` instead of `<?php`. That form only works when the `short_open_tag` setting is switched on, so the same file can silently stop working on another server. Always write `<?php`. (The short *echo* tag `<?=` is different and is always enabled.)

The kinds of value

Every value in PHP has a **type**: a label that tells PHP what sort of thing it is and what you can do with it. You can add two integers, but not two calendars. Figure 3.1 shows the family tree we will walk through.

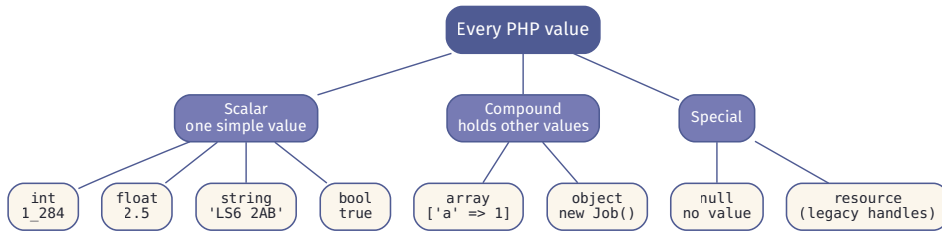


Figure 3.1 The PHP value types you will use every day. Resources are an older kind of handle that modern PHP is steadily replacing with objects.

Scalars: one simple value each

LISTING `scalars.php` the four scalar types

```

<?php
declare(strict_types=1);

$jobsThisYear = 1_284;           // int: underscores only aid reading
$hexColour    = 0x77_7B_B4;      // int written in hexadecimal
$filePerms    = 0o644;           // int in octal (0o prefix)
$flags        = 0b0101;          // int written in binary
$hoursOnSite  = 2.5;             // float
$customer     = 'Priya Shah';    // string
$isUrgent     = false;           // bool

var_dump($jobsThisYear, $hexColour, $filePerms, $flags);
var_dump($hoursOnSite, $customer, $isUrgent);

```

OUTPUT

```

int(1284)
int(7830452)
int(420)
int(5)
float(2.5)

```

```
string(10) "Priya Shah"
bool(false)
```

- **int** is a whole number. You can write it in decimal, hexadecimal (**0x**), octal (**0o**) or binary (**0b**); PHP stores the same kind of number whichever you choose. Underscores, as in **1_284**, are ignored by PHP and simply make long numbers easier to read.
- **float** is a number with a fractional part, such as **2.5** hours.
- **string** is a piece of text: a name, a postcode, a whole web page.
- **bool** is either **true** or **false**, and answers yes/no questions such as “is this job urgent?”.

var_dump() is your best friend while learning. It prints both the **type** and the **value**, and for strings it adds the length in bytes.

Floats are approximate: do not use them for money

Floats are stored in binary, and many tidy decimal numbers, such as 0.1, cannot be stored exactly. Usually the error is far too small to notice. With money, it eventually shows:

LISTING float-money.php why prices are stored in pence

```
<?php
declare(strict_types=1);

var_dump(0.1 + 0.2 === 0.3);           // the classic surprise
printf("%.17f\n", 0.1 + 0.2);         // what is really stored

// Store money as whole pennies (an int) instead.
$partsPence  = 1_999;                  // £19.99
$labourPence = 4_500;                  // £45.00
$totalPence  = $partsPence + $labourPence;

printf("Total: £%s\n", number_format($totalPence / 100, 2));
```

OUTPUT

```
bool(false)
0.30000000000000004
Total: £64.99
```

TIP Count pennies, not pounds

Keep money as an `int` number of pence and convert to pounds only when you display it. Integer arithmetic is exact, so £19.99 + £45.00 is always £64.99. Larger systems often use a dedicated money library or the `bcmath` extension; the principle is the same.

Strings: single quotes, double quotes and joining

LISTING strings.php quoting, interpolation and length

```
<?php
declare(strict_types=1);

$name = 'Priya';
$job  = ['type' => 'shelf fitting', 'hours' => 2];

echo 'Hello, $name\n', "\n"; // single: taken literally
echo "Hello, $name\n";      // double: variables expand
echo "Job: {$job['type']} ({ $job['hours']} h)\n";
echo 'Hello, ' . $name . '!\n'; // . joins strings
echo strlen('Café'), ' bytes, ', mb_strlen('Café'), " characters\n";
```

OUTPUT

```
Hello, $name\n
Hello, Priya
Job: shelf fitting (2 h)
Hello, Priya!
5 bytes, 4 characters
```

- **Single quotes** take the text literally. `$name` and `\n` stay exactly as typed.
- **Double quotes** expand variables and escape sequences such as `\n` (new line). Wrap anything more complex than a plain variable in braces: `{ $job['type'] }`.
- The **dot operator** `.` joins strings together.
- `strlen()` counts **bytes**, not letters. “Café” is five bytes in UTF-8 because `é` needs two. When you care about characters people can see, use the `mb_` functions such as `mb_strlen()`.

`null`: deliberately nothing

`null` means “no value here”. It is different from `0`, `''` or `false`, each of which is a real value. Customers do not always give a phone number, and `null` lets us record that honestly. The **null coalescing operator** `??` supplies a fallback:

LISTING `null.php` handling missing information

```
<?php
declare(strict_types=1);

$enquiry = [
    'name' => 'Tom Okafor',
    'phone' => null,           // the customer chose not to give one
];

$phone    = $enquiry['phone'] ?? 'not supplied';
// A missing key is fine too: no warning.
$company  = $enquiry['company'] ?? 'private customer';

echo "Phone: {$phone}\n";
echo "Company: {$company}\n";

$enquiry['source'] ??= 'website'; // assign only if missing or null
echo "Source: {$enquiry['source']}\n";
```

OUTPUT

```
Phone: not supplied
Company: private customer
Source: website
```

`??` checks whether the left-hand side exists *and* is not `null`. That makes it safe for array keys that might not be there at all, such as `company` above, without PHP raising an “undefined array key” warning. `??=` assigns only when the value is missing or `null`.

Arrays: many values under one name

An **array** stores a collection. When the keys are 0, 1, 2... in order, we call it a **list**. When you choose meaningful keys yourself, it is an **associative array**: think of it as a labelled form.

LISTING `arrays.php` lists, associative arrays and two PHP 8.5 helpers

```

<?php
declare(strict_types=1);

$services = [
    'Shelf fitting',
    'Flat-pack assembly',
    'Gutter clearing',
];

$job = [
    'customer' => 'Tom Okafor',
    'service'   => 'Gutter clearing',
    'postcode' => 'LS6 2AB',
];

echo $services[0], "\n";           // lists start at position 0
echo $job['postcode'], "\n";
echo count($services), " services\n";

var_dump(array_is_list($services), array_is_list($job));

// New in PHP 8.5: first and last values without juggling keys.
echo array_first($services), ' ... ', array_last($services), "\n";
var_dump(array_first([]));         // empty array gives null

```

OUTPUT

```

Shelf fitting
LS6 2AB
3 services
bool(true)
bool(false)
Shelf fitting ... Gutter clearing
NULL

```

PHP 8.5 New in PHP 8.5: `array_first()` and `array_last()`

These return the first or last *value* of any array, whatever its keys, and return `null` for an empty array, so they pair neatly with `??`. On PHP 8.4 or earlier, use `$services[array_key_first($services)]` instead, after checking the array is not empty.

Objects, briefly

An **object** bundles data with behaviour. You have already met one: `DateTimeImmutable` represents a moment in time and knows how to format itself. We create our own classes in Chapter 9; for now, recognise objects when you see `new`.

Asking a value what it is

LISTING `inspect.php` `get_debug_type()` versus `gettype()`

```
<?php
declare(strict_types=1);

$values = [
    42, 4.2, '42', true, null, [4, 2],
    new DateTimeImmutable('2026-10-17'),
];

foreach ($values as $value) {
    echo str_pad(get_debug_type($value), 18), gettype($value), "\n";
}
```

OUTPUT

```
int            integer
float          double
string         string
bool          boolean
null          NULL
array         array
DateTimeImmutable object
```

Prefer `get_debug_type()`. It uses the same names you write in type declarations (`int`, `float`, `bool`, `null`) and gives the class name for objects. The older `gettype()` still says `integer`, `double` and `boolean` for historical reasons.

Comparing values: == versus ===

PHP has two equality operators, and choosing the right one prevents a whole family of bugs.

- **=== (identical)** is true only when both values have the **same type and the same value**.
- **== (loosely equal)** first converts the values to a common type, then compares them.

LISTING comparisons.php where loose comparison surprises people

```
<?php
declare(strict_types=1);

$checks = [
    "1 == '1'"           => 1 == '1',
    "1 === '1'"         => 1 === '1',
    "0 == 'abc'"         => 0 == 'abc',    // false since PHP 8.0
    "'1e1' == '10'"     => '1e1' == '10', // both numeric
    "null == false"     => null == false,
    "null === false"    => null === false,
    "' ' == null"       => ' ' == null,
    "'0' == false"     => '0' == false,
];

foreach ($checks as $label => $result) {
    printf("%-16s %s\n", $label, var_export($result, true));
}
```

OUTPUT

```
1 == '1'           true
1 === '1'         false
0 == 'abc'         false
'1e1' == '10'     true
null == false     true
null === false    false
' ' == null       true
'0' == false      true
```

Some of those results are reasonable; others, such as `'0' == false`, catch people out regularly. Figure 3.3 shows the full picture for common values. It was generated by running every comparison through PHP 8.5 itself, so it is exact rather than remembered.

Which pairs does == call equal? (generated on PHP 8.5.11)

true	✓		✓		✓	✓		✓			✓	
false		✓		✓			✓		✓	✓		✓
1	✓		✓			✓						
0		✓		✓			✓		✓			
-1	✓				✓			✓				
"1"	✓		✓			✓						
"0"		✓		✓			✓					
"-1"	✓				✓			✓				
null		✓		✓					✓	✓		✓
[]		✓							✓	✓		
"php"	✓										✓	
""		✓							✓			✓
	true	false	1	0	-1	"1"	"0"	"-1"	null	[]	"php"	""

Figure 3.3 Loose comparison (==) between common values. A tick means PHP treats the pair as equal. Strict comparison (===) is simple: only the diagonal.

TIP Default to ===

Use `===` and `!==` unless you have a specific reason to want conversion. The same applies to functions that compare for you: pass `true` as the third argument to `in_array()` and `array_search()`. PHP's `match` expression always compares strictly, which is one reason we use it in `quote.php`.

NOTE PHP 8 fixed the worst case

Before PHP 8.0, `0 == 'abc'` was `true`, because the string was converted to the number 0. Since PHP 8.0 a number compared with a non-numeric string is compared *as strings*, so the result is now `false`. You will still find articles online describing the old behaviour.

Type declarations and strict types

So far PHP has trusted us to pass sensible values around. In a real application we want PHP to check. A **type declaration** states what type a function parameter must be, and what type the function returns:

LISTING `strict-types.php` catching the wrong type at the door

```
<?php
declare(strict_types=1);

function labourCost(int $minutes, int $ratePencePerHour): int
{
    return intval($minutes * $ratePencePerHour, 60);
}

echo labourCost(90, 4_800), " pence\n";

try {
    echo labourCost('90', 4_800), " pence\n"; // a string!
} catch (TypeError $e) {
    echo 'TypeError: ', $e->getMessage(), "\n";
}
```

OUTPUT

```
7200 pence
TypeError: labourCost(): Argument #1 ($minutes) must be of type int,
string given, called in strict-types.php on line 12
```

Read the signature aloud: “`labourCost` takes an `int` called `$minutes` and an `int` called `$ratePencePerHour`, and returns an `int`.” When we passed the *string* `'90'`, PHP refused with a `TypeError` that names the function, the parameter and the type it received. That is exactly the message you want at 9 p.m. when something has gone wrong.

What `strict_types` actually changes

Without the `declare(strict_types=1);` line, PHP runs in **coercive mode** and tries to convert scalar values to fit:

LISTING `coercive.php` the default mode converts when it can

```
<?php
// No strict_types declaration: PHP's default "coercive" mode.

function nights(int $count): string
{
    return "{$count} night(s)";
}

echo nights('3'), "\n";           // numeric string quietly becomes int 3
echo nights(3.0), "\n";          // whole-number float accepted

try {
    echo nights('three'), "\n";
} catch (TypeError $e) {
    echo 'TypeError: ', $e->getMessage(), "\n";
}
```

OUTPUT

```
3 night(s)
3 night(s)
TypeError: nights(): Argument #1 ($count) must be of type int, string
given, called in coercive.php on line 13
```

Coercive mode is forgiving, and forgiveness hides mistakes: if `'3'` came from a form field, you would never learn that you forgot to validate it. With strict types on, only exact matches pass, with one sensible exception: an `int` is accepted where a `float` is expected, because no information is lost.

LISTING `widening.php` the one conversion strict mode allows

```

<?php
declare(strict_types=1);

function area(float $lengthMetres, float $widthMetres): float
{
    return $lengthMetres * $widthMetres;
}

// An int is accepted for a float: the one allowed widening.
var_dump(area(3.5, 4));

try {
    var_dump(area(3.5, '4'));
} catch (TypeError $e) {
    echo get_class($e), ': ', $e->getMessage(), "\n";
}

```

OUTPUT

```

float(14)
TypeError: area(): Argument #2 ($widthMetres) must be of type float,
string given, called in widening.php on line 13

```

Figure 3.2 summarises the decision PHP makes every time a value meets a typed parameter.

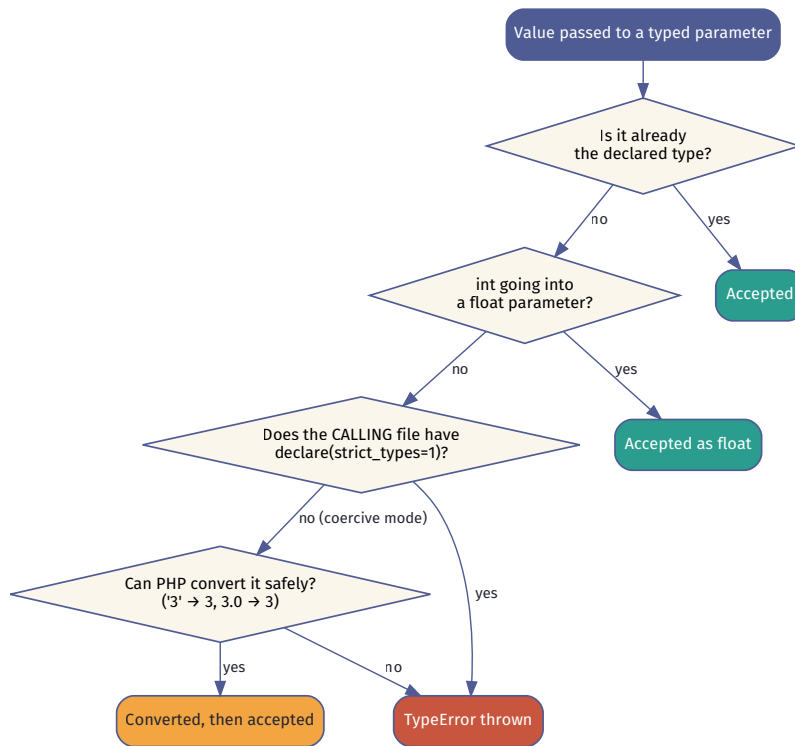


Figure 3.2 What happens when a value is passed to a typed parameter. Note that the setting that matters belongs to the file making the call.

WATCH OUT `strict_types` is per file, and must come first

`declare(strict_types=1);` must be the very first statement in the file, straight after `<?php`. It governs calls *made from* that file (and the `return` values of functions written in it), not calls into it from other files. Put it at the top of every PHP file you write: every listing in this book does, except the one demonstrating coercive mode.

NOTE Strict types are not validation

A `TypeError` tells you a programmer passed the wrong *kind* of value. It cannot tell you that 900 minutes is an unlikely shelf-fitting job, or that a postcode is malformed. Checking that values make sense for the business is still your job, and Chapter 13 is devoted to it.

Converting values on purpose

Sometimes you *want* a conversion: text from the command line, a form or a file always arrives as a string. A **cast** converts explicitly, by writing the target type in brackets.

LISTING `casts.php` what casts do with awkward input

```
<?php
declare(strict_types=1);

var_dump((int) '42');           // int(42)
var_dump((int) '42 hours');     // int(42): the rest is dropped
var_dump((int) 'about 42');     // int(0): no leading digits
var_dump((bool) '0');           // bool(false)
var_dump((bool) '0.0');         // bool(true)!
var_dump((string) false);       // string(0) ""

// Validate rather than cast when the value comes from a person:
var_dump(filter_var('42', FILTER_VALIDATE_INT));
var_dump(filter_var('42 hours', FILTER_VALIDATE_INT));
```

OUTPUT

```
int(42)
int(42)
int(0)
bool(false)
bool(true)
string(0) ""
int(42)
bool(false)
```

Look at `(int) '42 hours'`. A cast never fails: it takes the leading digits and quietly throws away the rest, and `'about 42'` becomes `0`. That is fine when you already know the string is a clean number, and dangerous when a person typed it. For input, **validate** instead: `filter_var()` with `FILTER_VALIDATE_INT` returns the integer or `false`, so bad input cannot slip through disguised as zero.

PHP 8.5 Deprecated in PHP 8.5: the long cast names

The aliases `(integer)`, `(double)`, `(boolean)` and `(binary)` now raise a deprecation notice. Use the short forms `(int)`, `(float)`, `(bool)` and `(string)`, which have always worked.

LISTING deprecated-casts.php the notice PHP 8.5 gives

```
<?php
declare(strict_types=1);

$hours = (integer) '3';    // old alias: deprecated in PHP 8.5
var_dump($hours);
```

OUTPUT

```
Deprecated: Non-canonical cast (integer) is deprecated, use the (int) cast
instead in deprecated-casts.php on line 4
int(3)
```

Tidying text with the pipe operator

PHP 8.5 adds the **pipe operator** `|>`, which passes the value on its left into the function on its right. A chain of pipes reads top to bottom, in the order things happen, instead of inside-out:

LISTING pipe.php cleaning up a customer's name

```
<?php
declare(strict_types=1);

$typed = "    priya    SHAH    ";

// PHP 8.5 pipe operator: the value flows left to right.
$name = $typed
    |> trim(...)
    |> (fn(string $s): string => preg_replace('/\s+/', ' ', $s))
    |> strtolower(...)
    |> ucwords(...);

var_dump($name);
```

OUTPUT

```
string(10) "Priya Shah"
```

`trim(...)` is a **first-class callable**: the function itself, ready to be called later. The middle step is a short arrow function, which must be wrapped in brackets inside a pipe chain. If you need to support PHP 8.4, write the same steps as ordinary nested calls.

Putting it together: quote.php

Here is the complete estimating tool. It uses almost everything in this chapter: strict types, typed functions, integers for money, `null` for “not found”, strict `match`, validation with `filter_var()` and a clear error path.

LISTING `quote.php` Northstar Repairs' job estimator

```
<?php
declare(strict_types=1);

/**
 * Northstar Repairs: estimate a job from the command line.
 * Usage: php quote.php <service> <minutes> [parts-in-pence]
 */

const VAT_RATE_PERCENT = 20; // UK standard rate (check it)

function hourlyRatePence(string $service): ?int
{
    return match ($service) {
        'shelf'      => 4_800,
        'flatpack'   => 4_200,
        'gutter'     => 5_500,
        default      => null, // unknown service: no rate
    };
}

function labourPence(int $minutes, int $ratePence): int
{
    $billable = max(60, $minutes); // one-hour minimum charge
    return intdiv($billable * $ratePence, 60);
}

function formatPounds(int $pence): string
```

```

{
    return '£' . number_format($pence / 100, 2);
}

$service = $argv[1] ?? '';
$minutes = filter_var($argv[2] ?? '', FILTER_VALIDATE_INT,
    ['options' => ['min_range' => 1]]);
$parts    = filter_var($argv[3] ?? '0', FILTER_VALIDATE_INT,
    ['options' => ['min_range' => 0]]);
$rate     = hourlyRatePence($service);

if ($rate === null || $minutes === false || $parts === false) {
    fwrite(STDERR, "Usage: php quote.php shelf|flatpack|gutter"
        . " <minutes> [parts-in-pence]\n");
    exit(1);
}

$labour = labourPence($minutes, $rate);
$net    = $labour + $parts;
// Round VAT half-up to the nearest penny.
$vat    = intval($net * VAT_RATE_PERCENT + 50, 100);
$total  = $net + $vat;

printf("%-8s %10s\n", 'Labour', formatPounds($labour));
printf("%-8s %10s\n", 'Parts', formatPounds($parts));
printf("%-8s %10s\n", 'VAT', formatPounds($vat));
printf("%-8s %10s\n", 'Total', formatPounds($total));

```

Try it with a valid job, then with mistakes:

```

$ php quote.php shelf 90 1999
Labour      £72.00
Parts       £19.99
VAT         £18.40
Total       £110.39

$ php quote.php gutter 30
Labour      £55.00
Parts       £0.00
VAT         £11.00
Total       £66.00

$ php quote.php ladder 30
Usage: php quote.php shelf|flatpack|gutter <minutes> [parts-in-pence]

```

Walk through the important lines:

- `const VAT_RATE_PERCENT = 20;` gives a meaningful name to a value that might change. 20% is the UK standard rate at the time of writing; real invoicing rules (reduced rates, VAT registration thresholds) are a matter for the business and its accountant.
- `?int` as a return type means “an `int` or `null`”. `hourlyRatePence()` returns `null` for a service we do not offer, and the caller checks with `=== null`.
- `match` compares the service name strictly and returns the matching rate. Unlike `switch`, it cannot accidentally “fall through” to the next case.
- `$argv` is the list of words typed after `php` on the command line, *always as strings*. That is why `filter_var()` validates the minutes and pence before any arithmetic happens. The gutter job is 30 minutes but is billed as a full hour, because of the minimum charge in `labourPence()`.
- `intdiv()` divides integers and discards any remainder, so labour and VAT stay in whole pence. Adding 50 before dividing by 100 rounds the VAT half-up to the nearest penny.
- **Errors go to `STDERR` with exit code 1**, so other scripts (and you) can tell success from failure.

TIP Read the error before changing the code

When a script misbehaves, run it again with `var_dump()` on the value you are unsure of. Nine times out of ten, the type or value is not what you assumed, and the dump tells you so immediately.

Common mistakes and how to diagnose them

Symptom	Likely cause	Fix
Parse error: syntax error, unexpected token ... on a line that looks fine	Missing <code>;</code> at the end of the previous line	Check the line above the one reported
Warning: Undefined variable <code>\$nmae</code>	A typo in a variable name	Copy names rather than retyping; a good editor will highlight these
A price is off by a penny	Money held in a float	Store pence as <code>int</code> ; format only for display
A check passes when it should not	<code>==</code> converting values	Switch to <code>===</code> ; pass <code>true</code> to <code>in_array()</code>
TypeError ... must be of type <code>int</code> , string given	A string from input reached a typed function	Validate with <code>filter_var()</code> first, then pass the result
Text shows <code>&amp;amp;</code>	Escaped twice	Escape once, at the moment you print into HTML

Exercises

- Add a service.** Give `quote.php` a `paint` service at £38 an hour. Run it for 150 minutes with £24.50 of parts. Work out the total by hand first, then check.
- Make it fail loudly.** Remove `declare(strict_types=1);` from `strict-types.php` and run it again. What changed, and why is the strict version safer?
- Predict, then run.** Before running them, write down the result of `'10' == '1e1'`, `100 == '1e2'` and `'abc' == 0`. Check your answers with `var_dump()`.
- Use the pipe.** Write a pipe chain that turns `" 1s6 2ab "` into `"LS6 2AB"`.
- Stretch.** Change `quote.php` so the parts argument can be written in pounds, such as `19.99`, but is still stored as pence. Which function would you use to validate it, and how will you avoid float rounding errors? (Hint: treat it as text and split on the decimal point.)

Checklist

- ☐ I can explain the difference between `int`, `float`, `string`, `bool`, `array` and `null`.
- ☐ I store money as whole pence.
- ☐ I use `===` unless I have a reason not to.
- ☐ Every PHP file I write starts with `declare(strict_types=1);`.
- ☐ I validate input with `filter_var()` instead of casting it.
- ☐ `quote.php` runs, prints a correct estimate, and fails clearly on bad input.

Next up, Chapter 4: Variables, Constants and Operators. We will look closely at how variables are scoped, give the business settings proper constants, and teach `quote.php` to apply discounts with arithmetic you can trust.

Sources

Facts in this chapter were checked against the official PHP documentation on 17 October 2026:

- PHP Manual, *Types*: php.net/manual/en/language.types.php
- PHP Manual, *Type declarations* and *Strict typing*: php.net/manual/en/language.types.declarations.php
- PHP Manual, *Comparison operators* and *PHP type comparison tables*: php.net/manual/en/language.operators.comparison.php, php.net/manual/en/types.comparisons.php
- PHP Manual, *Integers* (octal `0o` prefix, PHP 8.1): php.net/manual/en/language.types.integer.php
- PHP Manual, *Floating point numbers* (precision warning): php.net/manual/en/language.types.float.php
- PHP RFC, *Saner string to number comparisons* (PHP 8.0): wiki.php.net/rfc/string_to_number_comparison
- PHP 8.5 release announcement (pipe operator, `array_first()` / `array_last()`, cast deprecations): php.net/releases/8.5/

- PHP Manual, *Basic syntax: PHP tags* (`short_open_tag` , omitting `?>`): php.net/manual/en/language.basic-syntax.phptags.php
- HMRC, *VAT rates*: gov.uk/guidance/rates-of-vat-on-different-goods-and-services