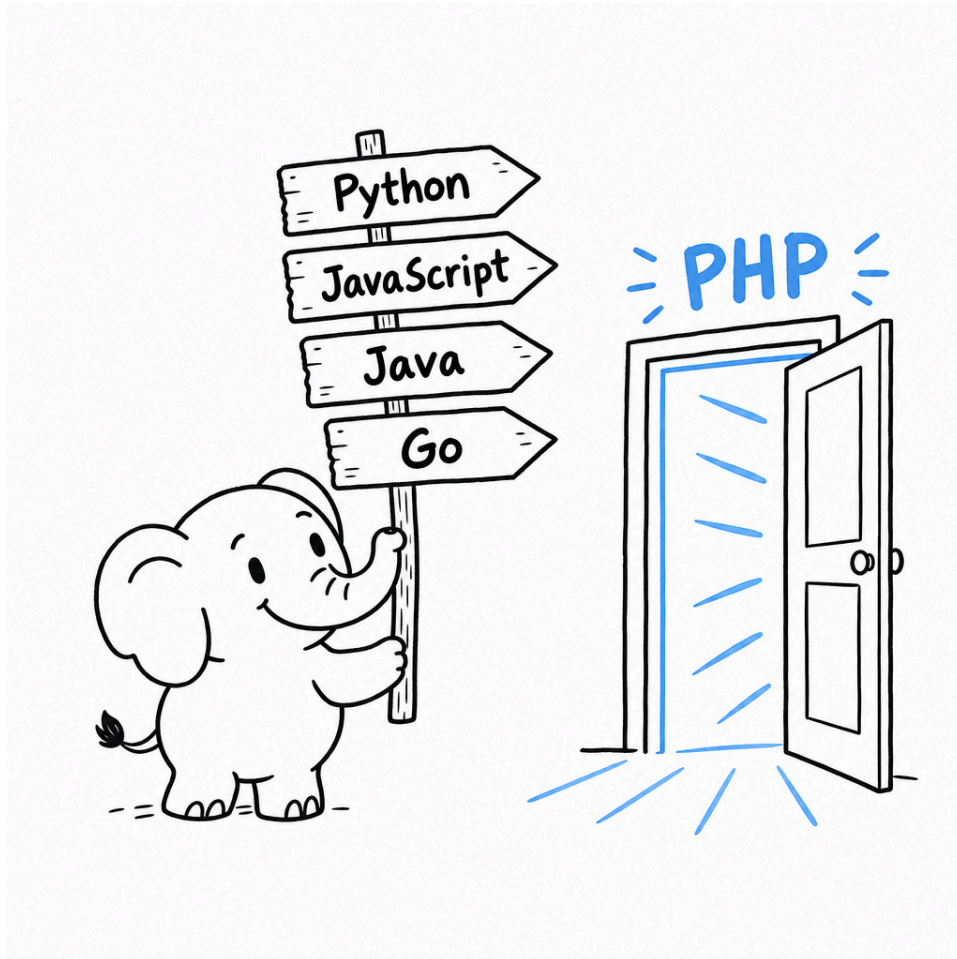




And Now, PHP

A two-to-three-hour guide to modern PHP, for developers coming from another language.



You know how to code. You have just been handed a PHP project, or you are coming back to PHP after years away. You do not need a tutorial. You need the delta: how PHP runs, how it types, how it packages code, where it will surprise you, and what today's PHP looks like once the old reputation is set aside.

This book takes two to three hours to read. Each chapter answers one question, “how does PHP do this?”, and moves on.

The PHP described here is PHP 8.5. Features younger than PHP 8.2 carry their version number so you know what your project can use.

Introduction

You have been handed a PHP codebase. Maybe you asked for it, more likely you did not. You open the first file, you see `$this->` and `->` and `::` and a function called `htmlspecialchars`, and a thought forms: is this going to be the language everyone warned me about?

The PHP people warned you about was real, and it is mostly gone. What is left is a typed, object-oriented, garbage-collected language with a package manager, a standard coding style, a serious static analysis ecosystem, a release every November, and a runtime model that is unlike anything you have used, which is the one thing you actually need to learn.

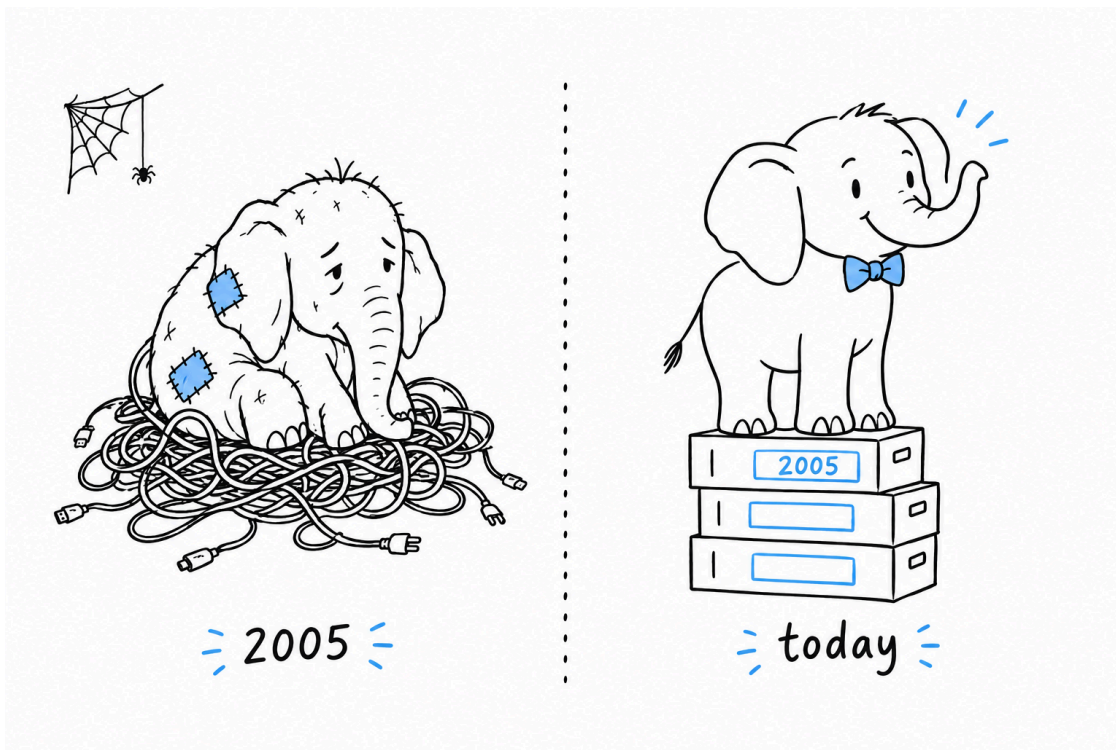
What was true

PHP started in 1995 as a set of templates with a bit of logic in them. For its first decade it was permissive to a fault. Variables appeared out of nowhere, `"abc" == 0` was true, errors were printed to the page and execution continued, database queries were built by string concatenation, and the standard library was assembled one function at a time by whoever needed it that week, which is why `strpos` sits next to `str_replace` and `array_key_exists` sits next to `in_array`.

A whole generation learned to program on that PHP, wrote a great deal of it, and much of it still runs. That is the PHP of the jokes.

What changed

PHP 7 (2015) doubled performance and added scalar type declarations. PHP 8 (2020) added a real type system with union types, `match`, named arguments, attributes, enums, `readonly`, first-class callables and a JIT compiler. PHP 8.4 added property hooks and asymmetric visibility. PHP 8.5 added a pipe operator. The comparison rules were fixed, dynamic properties were deprecated, the old `mysql_*` functions were removed, and the interpreter now throws a `TypeError` where it used to guess.



Around the language, the community built Composer, the package manager every project uses; the PHP-FIG standards, so that libraries from different authors fit together; PHPUnit and Pest for testing; PHPStan and Psalm, two static analysers that give you most of what a compiler would; and Rector, which rewrites old code to new syntax. Since 2021 the PHP Foundation has employed core developers so the language has a future that does not depend on volunteers' evenings.

What still bites

Not everything was fixed, and a book for busy people should say so up front. Strings are byte sequences, so `strlen('é')` is 2 and you want the `mb_` functions for text. The standard library still has its inconsistent names and argument orders. `==` still coerces, though far less wildly than before. Strict typing is a per-file switch you must turn on. Arrays copy when you assign them. There are no generics in the language itself. Each of these has a modern practice that neutralises it, and each is covered where you will meet it.

The reputation stayed in 2010. The language did not.

The one idea to get first

A PHP web request starts with nothing, runs your code top to bottom, sends its answer, and throws everything away. No shared memory between requests, no application object that stays alive, no event loop, no threads. The web server hands a request to PHP, PHP produces a response, PHP forgets.

If you come from Node, Java, Go or an ASGI Python server, this is the single largest difference, and most PHP idioms follow from it: why there is no connection pool by default, why “global state” is a per-request thing, why a crash affects one visitor, why scaling is a matter of adding processes, why OPcache exists, and why long-running PHP servers like FrankenPHP or RoadRunner are a topic in their own right. [How PHP Runs](#) covers it, and it is the one chapter not to skip.

How to read this book

Each chapter answers one question and stands on its own. The bold sentences carry the story: read only those and you get the delta between PHP and what you already know. Read the code blocks and you get the syntax. Read the rest when a detail matters to you.

The comparisons to Python, JavaScript, Java and a few other languages are anchors, not translations. Skip the ones for languages you do not know. Nothing depends on them.

Every example runs on a bare PHP install with `php file.php`. No framework, no library, so that the lesson is about PHP itself. Install PHP first: your package manager has it, [php.net](#) lists official builds, and the official Docker image is `php:8.5-cli`. Then keep a terminal open and run what you read.

Three chapters serve a particular reader. [Returning to PHP After Years Away](#) maps old habits to modern practice, for developers returning to the language. [Coming from Python, JavaScript, or Java](#) is a lookup table. [PHP 8.0 to 8.5 at a Glance](#) tells you what your project’s PHP version can do.

Two to three hours from now, the codebase you opened will look different. Not simpler, necessarily. Just legible.

How PHP Runs

PHP does not run your application. It runs your script, once, for one request, and then exits.

There is no server object you instantiate, no `listen()` call, no event loop. Something outside PHP (a web server, or you at a terminal) starts the interpreter, the interpreter runs a file from top to bottom, and everything it allocated is freed when the file ends.

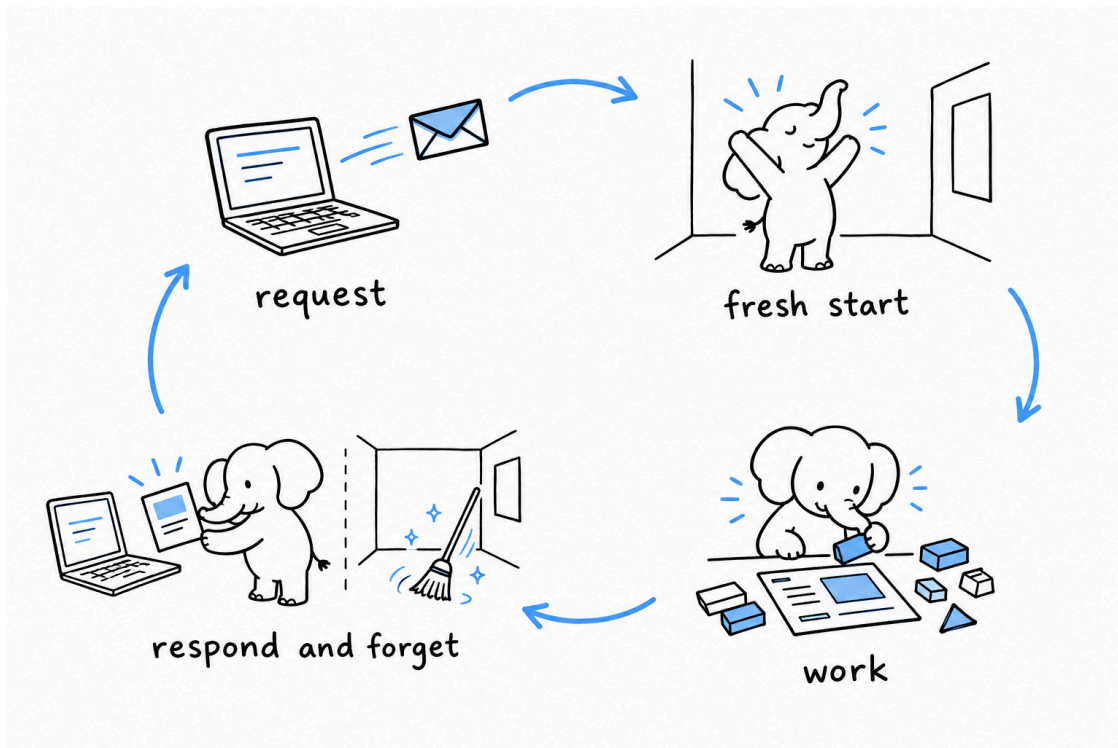
Every other chapter in this book is easier once this one has sunk in.

Two ways in

From a terminal, PHP behaves like Python or Ruby:

```
php hello.php
php -r 'echo PHP_VERSION, PHP_EOL;'
php -a           # interactive shell
php -l file.php  # syntax check only
php -S localhost:8000 # development web server, current folder as document root
```

On the web, PHP is not the server. **The web server receives the HTTP request and hands it to PHP**, most often through PHP-FPM, a pool of PHP processes waiting behind nginx, Apache or Caddy. One process takes the request, runs the script the URL maps to, writes the response, cleans up, and goes back to waiting for the next one. The pool has as many processes as you configure, which is also how PHP uses all your cores: not with threads, with processes.



The layer that connects the interpreter to the outside world is called a SAPI (server API). The command line is a SAPI. FPM is a SAPI. Apache's `mod_php` is a SAPI. The engine is the same; only the plumbing differs.

Shared nothing

This is the part that changes how you write code. **Nothing survives from one request to the next inside PHP.** A static property you set, a global you assign, a connection you open, an object you cache in an array: all of it exists for one request and is gone.

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

final class Counter
{
    public static int $hits = 0;
}

Counter::$hits++;
echo Counter::$hits; // 1, on every single request, forever
```

Run that under a web server, refresh the page a hundred times, and it prints 1 a hundred times. In Node or Java the same code would count to 100. Neither is a bug. They are different models.

The consequences follow one from another:

- **A bug affects one request.** A memory leak, an infinite loop, an uncaught exception: the process handling that request dies or is recycled, and the next request gets a fresh one.
- **Scaling is horizontal by construction.** More traffic, more FPM processes, more machines. Nothing in the application needs to be thread-safe, because nothing is shared.
- **State lives outside PHP.** Sessions go to files, a database, or a key-value store. Caches go to OPcache and APCu (memory shared between the processes of one machine) or to an external store like Redis or Memcached. The database connection is opened at the start of a request and closed at the end; pooling, if you need it, happens in a pooler in front of the database, not in PHP.
- **Startup cost is paid on every request.** That is why PHP is fast at starting and why the ecosystem cares about autoloading, OPcache, and preloading.

If you find yourself designing a singleton to “keep the connection open between requests”, stop. There is no between.

The bytecode cache

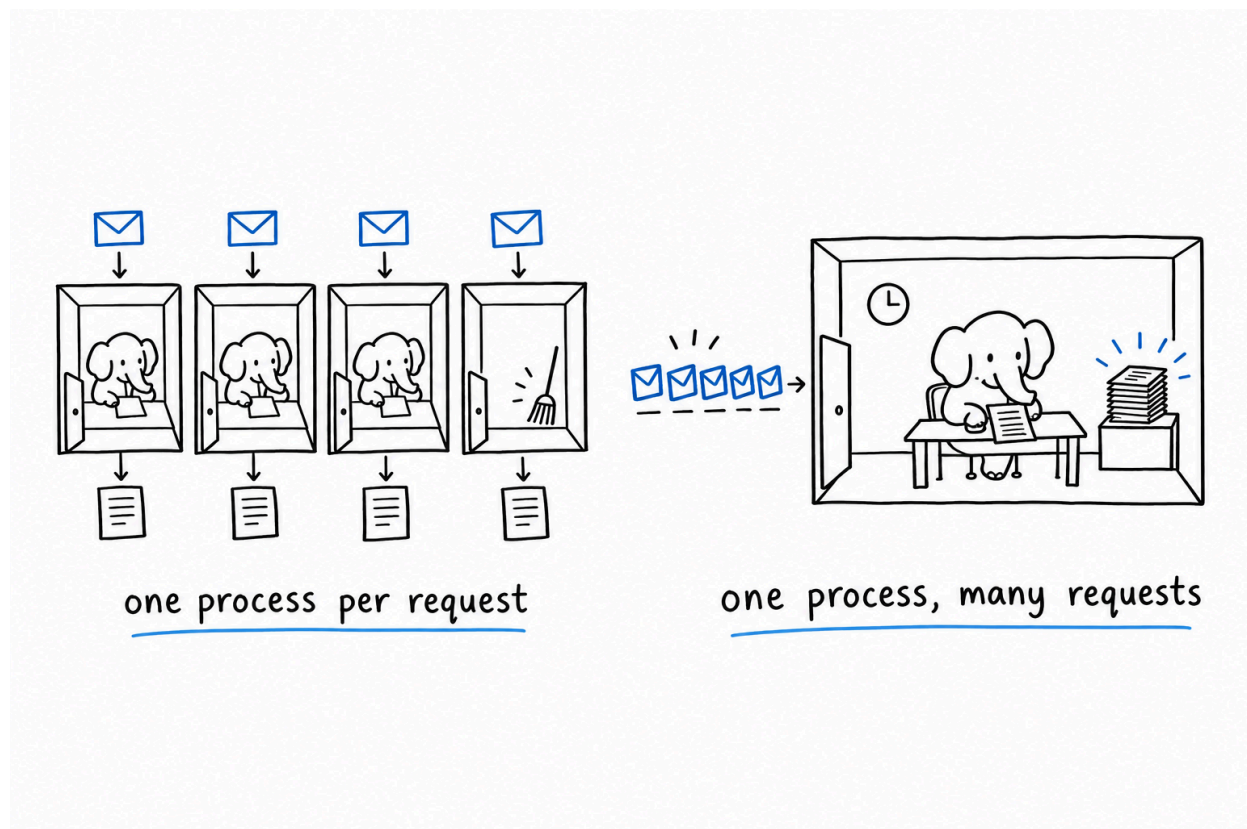
Reading and compiling every file on every request would be slow, so PHP does not. **OPcache keeps the compiled form of each file in shared memory**, and the next request reuses it. It ships with PHP and is on by default in every serious setup.

In development, OPcache checks file timestamps and recompiles what changed, so the edit-refresh loop just works, with no build step and no watcher. In production, timestamp checks are usually disabled for speed, which means **a deployment must reset the cache**: restart FPM, or call `opcache_reset()`. Forgetting this is the classic “I deployed and nothing changed” moment.

OPcache also hosts the JIT compiler (PHP 8.0). It helps CPU-bound scripts, mostly, and is not what makes a typical web application fast. Treat it as a flag to try, not as a foundation.

Long-running PHP

The shared-nothing model is the default, not a law. **Several runtimes keep your application in memory across requests**, the way a Node or Java server does: FrankenPHP in worker mode, RoadRunner, and Swoole or OpenSwoole. Your bootstrap runs once, then a loop hands you requests.



The gain is real: no bootstrap cost per request, and connections can actually be kept open. The cost is the one you already know from other languages: state now leaks unless you clean it, a memory leak grows, and a static counter really does count. Frameworks that support these runtimes reset their containers between requests for exactly this reason. Start with FPM. Move to a worker runtime when you have measured a reason to.

What is in the box

The interpreter is a C core plus extensions, some bundled and always on, some compiled in at build time, some installed separately. `php -m` lists what your build has. The ones you will notice missing on a fresh install are usually `pdo_mysql` or `pdo_pgsql` (database drivers), `intl` (Unicode collation, formatting), `mbstring` (multibyte strings), `curl`, `gd` or `imagick` (images), and `xdebug` (debugger, development only).

Your package manager provides them as separate packages (`php-intl`, `php-mbstring`, and so on). The official Docker image provides `docker-php-ext-install`. Extensions not shipped with PHP come from PECL, or from PIE, the newer Composer-style installer for extensions.

Configuration lives in `php.ini`. **The command line and FPM read different ini files**, which is why a script behaves one way in a terminal and another under the web server. `php --ini` shows which files the CLI loads; `phpinfo()` in a page shows what FPM loads. Two settings matter from day one: `memory_limit` (128 MB by default under FPM, unlimited in the CLI) and `max_execution_time` (30 seconds under FPM, unlimited in the CLI).

The trap

Coming from a long-running runtime, the first mistake is to expect memory to persist: a cache in a static array, a “connection pool” class, a counter for rate limiting. Under FPM these silently do nothing useful. The second mistake is the mirror image: after moving to a worker runtime, a request-scoped value stored in a static ends up shared between users.

Ask one question about any piece of state: does it need to outlive this request? If yes, it does not belong in PHP memory. Put it in the database, a cache store, or the session.

With the model in place, the syntax is the easy part. [Syntax](#) is exactly that.

Syntax

PHP looks like C with a `$` in front of every variable, `->` for members, and `.` for string concatenation. If you read Java, JavaScript or C#, you already read PHP. The rest of this chapter is the list of places where your fingers will type the wrong thing.

One file, top to bottom

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

$name = 'world';
$count = 3;

echo "Hello, {$name}!", PHP_EOL;
echo 'Hello, ' . $name . '! Count: ' . $count . PHP_EOL;
```

The file opens with `<?php`. Everything before that tag, even a blank line, is sent to the output as-is, because PHP began life as a template language. **A file that contains only code opens with `<?php` and never closes the tag:** a `?>` at the end followed by a stray newline would leak that newline into your HTTP response.

`declare(strict_types=1);` comes next, on its own line. It turns off automatic conversion of scalar arguments in calls made from this file. [Types](#) explains exactly what it does and does not cover. For now: put it in every file, and think of a file without it as a file with a warning sign on it.

Variables start with `$` and are never declared. Assign, and the variable exists. There is no `let`, no `var`, no type in front. Variable names are case-sensitive; function and class names are not, though nobody relies on that.

Statements end with a semicolon. `echo` takes a comma-separated list and prints it; `print` does the same with one argument and is rarely seen. `PHP_EOL` is the platform's newline.

Strings

The concatenation operator is `.`, not `+`. Writing `$a + $b` with two strings makes PHP try to add them as numbers, which is a `TypeError` for non-numeric strings under PHP 8.

Single quotes give you the bytes you typed. Double quotes interpolate variables and translate escapes like `\n`. Wrap anything beyond a plain variable in braces:

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

$user = ['name' => 'Ada'];
$items = 3;

echo "Hi {$user['name']}, you have {$items} items\n";
echo 'Hi {$user[name]}\n'; // printed literally, backslash included
```

For multi-line text, a heredoc interpolates and a nowdoc does not. The closing marker may be indented (since PHP 7.3), and that indentation is removed from every line:

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

$title = 'Report';

$html = <<<HTML
    <h1>{$title}</h1>
    <p>Generated by PHP</p>
    HTML;

$raw = <<<'TXT'
    No {$interpolation} in here.
    TXT;

echo $html, PHP_EOL, $raw, PHP_EOL;
```

Three arrows

PHP has three arrow-shaped tokens, and they never overlap.

`->` reaches into an instance: `$user->name`, `$user->save()`. `::` reaches into a class: `User::create()`, `User::MAX_AGE`, `self::$count`. `=>` pairs a key with a value inside an array literal or a match arm: `['id' => 1]`.

```

<?php
declare(strict_types=1);

final class Money
{
    public const string CURRENCY = 'EUR'; // typed constant, PHP 8.3

    public function __construct(public readonly int $cents)
    {
    }

    public static function zero(): self
    {
        return new self(0);
    }
}

$price = new Money(1999);
echo $price->cents, ' ', Money::CURRENCY, ' ', Money::zero()->cents,
PHP_EOL;
// 1999 EUR 0

```

Inside a method, the current instance is `$this`. Classes and objects get their own chapter, [Classes](#); this example only shows which arrow goes where.

Equality

Use `===`. Treat `==` as a legacy operator. The double equals converts both sides to a common type before comparing, which produced the famous absurdities of old PHP. Since PHP 8 a number compared to a non-numeric string is compared as strings, so `0 == 'foo'` is `false` and `'1' == '01'` is still `true`. Saner, but still a guessing game. Triple equals compares value and type, with no guessing.

The same rule applies to `!==` versus `!=`, and to `in_array()` and `array_search()`, which take a third `true` argument to compare strictly.

`<=>` is the spaceship: it returns `-1`, `0` or `1`, and exists for sort callbacks.

Null, in three operators

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

$config = ['debug' => null, 'owner' => null];

$debug = $config['debug'] ?? false;      // false: ?? treats null like
missing
$config['level'] ??= 'info';              // assign only if null or
missing
$length = $config['owner']?->name;        // null, no error: nullsafe
chain
$mode = $config['debug'] ? 'on' : 'off'; // plain ternary, same as
everywhere

var_dump($debug, $config['level'], $length, $mode);
```

`??` is null coalescing and also swallows “undefined key” and “undefined variable”, which makes it the idiomatic way to read an optional array entry. `??=` assigns only when the left side is null or missing. `?->` (PHP 8.0) stops a chain at the first null and yields null. The shorthand ternary `?:` exists too (`$a ?: $b` returns `$a` if truthy), and is fine with booleans, misleading with anything else.

Control flow, all at once

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

$scores = ['ada' => 92, 'linus' => 71, 'grace' => 85];

foreach ($scores as $who => $score) {
    if ($score >= 90) {
        $grade = 'A';
    } elseif ($score >= 80) {
        $grade = 'B';
    } else {
        $grade = 'C';
    }
    echo "{$who}: {$grade}", PHP_EOL;
}

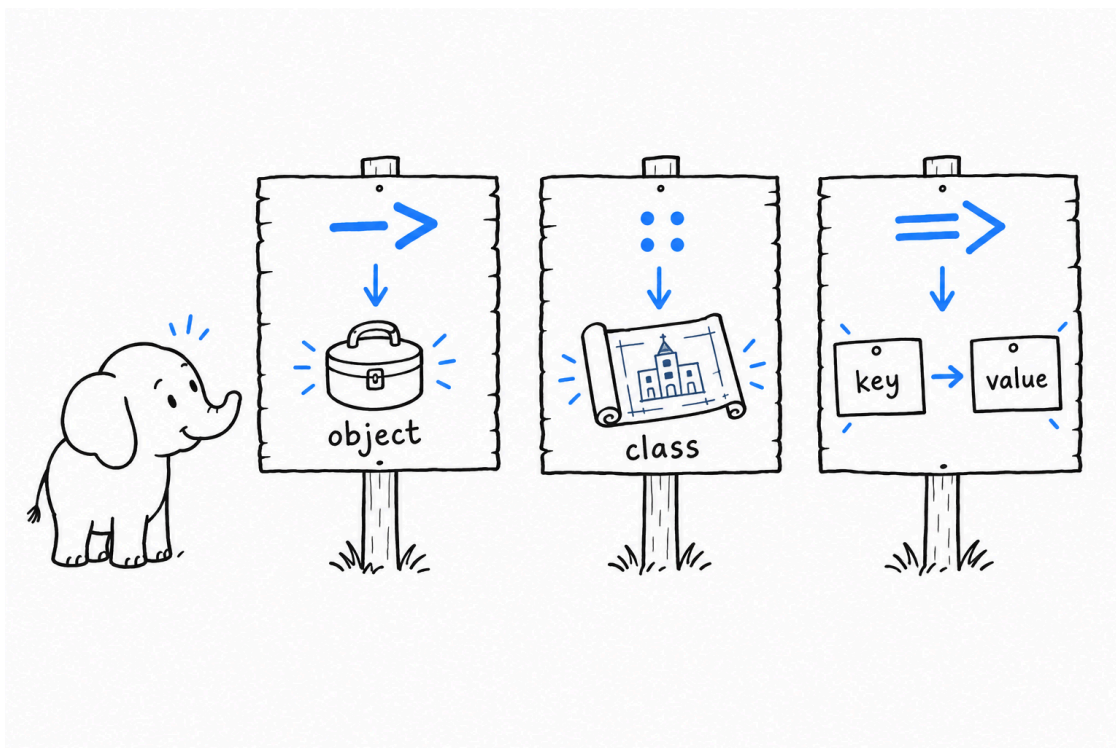
for ($i = 0; $i < 3; $i++) {
    echo $i;
}
echo PHP_EOL;

$attempts = 0;
while ($attempts < 3) {
    $attempts++;
}

$label = match (true) {
    $attempts === 0 => 'never tried',
    $attempts < 3 => 'a few tries',
    default => 'gave it everything',
};
echo $label, PHP_EOL;
```

Nothing here needs explaining except two details. `foreach` is the loop for arrays and any iterable, and it gives you the key and the value together with `as $key => $value`. `match` is an expression, compares with `===`, has no fall-through, and throws if nothing matches; `switch` still exists,

compares with `==`, falls through, and is what you find in older code. [Enums and match](#) shows what `match` is for.



Everything else on one screen

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

namespace App\Billing;           // one per file, mirrors the folder

use App\Money;                  // import a class from another namespace
use function App\format_cents;  // functions and constants can be
imported too

const TAX_RATE = 0.2;           // compile-time constant, namespaced
define('LEGACY', true);         // runtime constant, global, older style

// one-line comment
# also a one-line comment, rarer
/* block
    comment */

/**
 * Docblock: read by editors, PHPStan and Psalm, not by PHP itself.
 */
function total(int ...$cents): int
{
    return array_sum($cents);
}

$parts = [100, 250];
echo total(...$parts), PHP_EOL; // 350: ... unpacks on both sides
```

`...` unpacks an array into arguments and, in a parameter list, collects the rest into an array. Named arguments (`total(cents: 5)`) have existed since 8.0. `null` is a value, lowercase by convention, with `true` and `false`. Namespaces and `use` do exactly what you expect from Java packages or ES imports; [Namespaces, Composer, and Autoloading](#) covers how files get found.

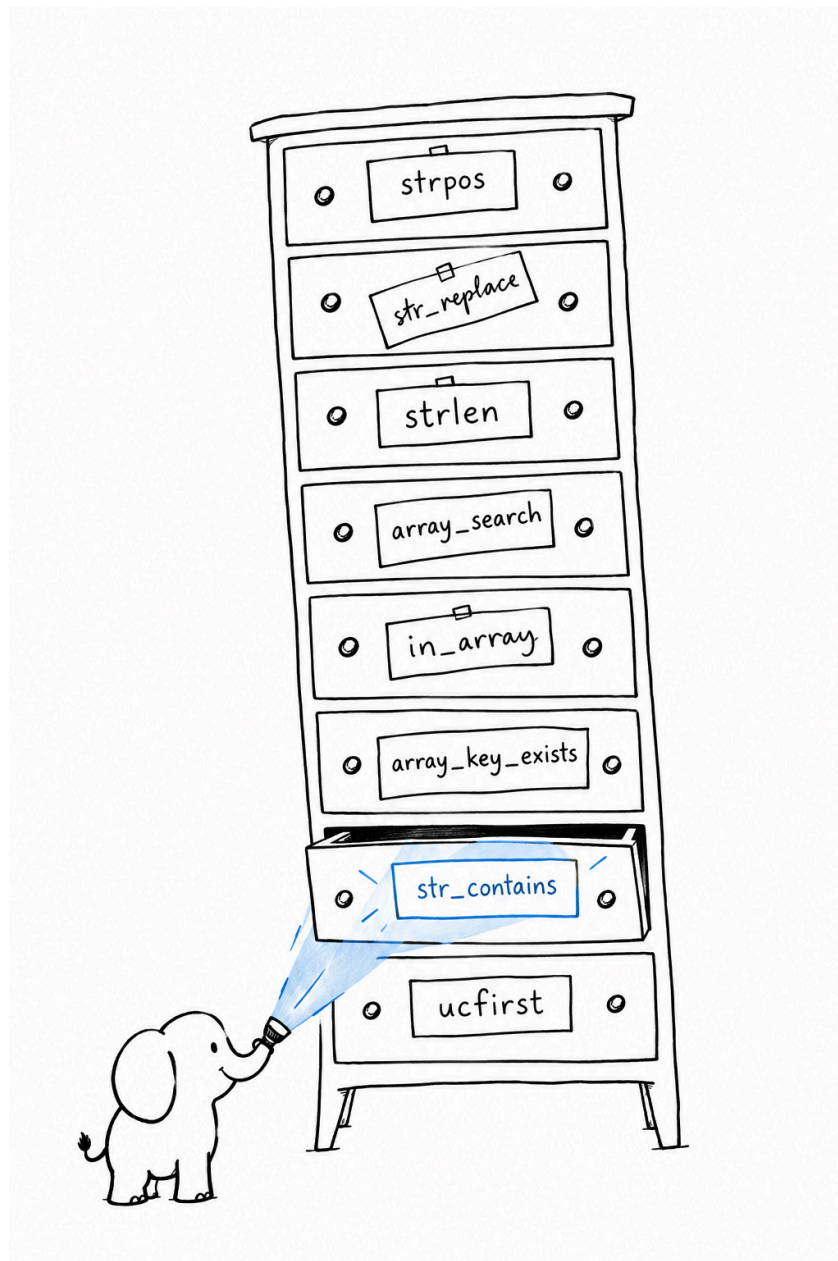
Side by side

	Python	JavaScript	Java	PHP
Concatenate	<code>a + b</code>	<code>a + b</code>	<code>a + b</code>	<code>\$a . \$b</code>
Strict equality	<code>a == b</code>	<code>a === b</code>	<code>a.equals(b)</code>	<code>\$a === \$b</code>
Null coalescing	<code>a or b</code>	<code>a ?? b</code>	<code>(none)</code>	<code>\$a ?? \$b</code>
Null-safe member	<code>(none)</code>	<code>a?.b</code>	<code>(none)</code>	<code>\$a?->b</code>
Lambda	<code>lambda x: x * 2</code>	<code>x => x * 2</code>	<code>x -> x * 2</code>	<code>fn(\$x) => \$x * 2</code>
Map literal	<code>{'k': 1}</code>	<code>{k: 1}</code>	<code>Map.of("k", 1)</code>	<code>['k' => 1]</code>
Interpolation	<code>f"{x}"</code>	<code>`\${x}`</code>	<code>(none)</code>	<code>"\${x}"</code>
Instance member	<code>obj.name</code>	<code>obj.name</code>	<code>obj.name</code>	<code>\$obj->name</code>
Static member	<code>Cls.name</code>	<code>Cls.name</code>	<code>Cls.name</code>	<code>Cls::\$name</code>
Ternary	<code>a if c else b</code>	<code>c ? a : b</code>	<code>c ? a : b</code>	<code>\$c ? \$a : \$b</code>

The trap

Two things trip every newcomer in the first hour. The first is `==`: a comparison that looks right, passes the obvious tests, and then decides that `'1e1' == '10'` and `null == false`. Type `===` until it is a reflex, and let `strict_types` catch the rest.

The second is the standard library. `strpos($haystack, $needle)` but `array_search($needle, $haystack)`. `str_replace` with underscores, `strlen` without. `array_key_exists` next to `in_array`. **The naming was never designed; it accumulated.** Newer functions are consistent (`str_contains`, `array_is_list`, `array_find`), the old ones will not change, and the fix is not memory but an editor with completion and a static analyser that flags the wrong argument order.



With the syntax out of the way, the interesting questions start. What does `int` actually enforce? [Types](#) answers that with more nuance than you might like.

Types

PHP is dynamically typed, and every type you write down is checked at runtime. Not at compile time like Java, not erased like TypeScript. A parameter declared `int` receives an `int` or the call throws a `TypeError`, every time, in production. That is the whole model; the nuance is in what “receives an `int`” means, and that nuance is a switch.

The types you can write

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

function describe(int|float $n, ?string $label, bool $verbose = false):
string
{
    return ($label ?? 'value') . ': ' . $n . ($verbose ? ' (verbose)' :
    '');
}

echo describe(3, null), PHP_EOL;           // value: 3
echo describe(2.5, 'pi-ish', true), PHP_EOL; // pi-ish: 2.5 (verbose)
```

Scalars are `int`, `float`, `string`, `bool`. Compound types are `array`, `object`, `callable`, `iterable`, and any class or interface name. On top of those, PHP has a handful of types that only make sense in one position: `void` and `never` as return types (`never` means the function throws or exits), `static` as the return type of a fluent method, `self` for the current class, `mixed` when you really accept anything, and `null`, `true`, `false` as standalone types since PHP 8.2.

They combine. `?string` is `string|null`. `int|string` is a union (PHP 8.0). `Countable&Traversable` is an intersection (PHP 8.1): the object must implement both. `(Countable&Traversable)|null` mixes the two (PHP 8.2), and that is as far as the grammar goes. No generics, no tuples, no structural types.

Types go on parameters, return values, properties, and since PHP 8.3 on class constants:

```

<?php
declare(strict_types=1);

final class Temperature
{
    public const string UNIT = 'C';

    public function __construct(
        public readonly float $degrees,
    ) {
    }
}

$t = new Temperature(21.5);
var_dump($t->degrees); // float(21.5)

```

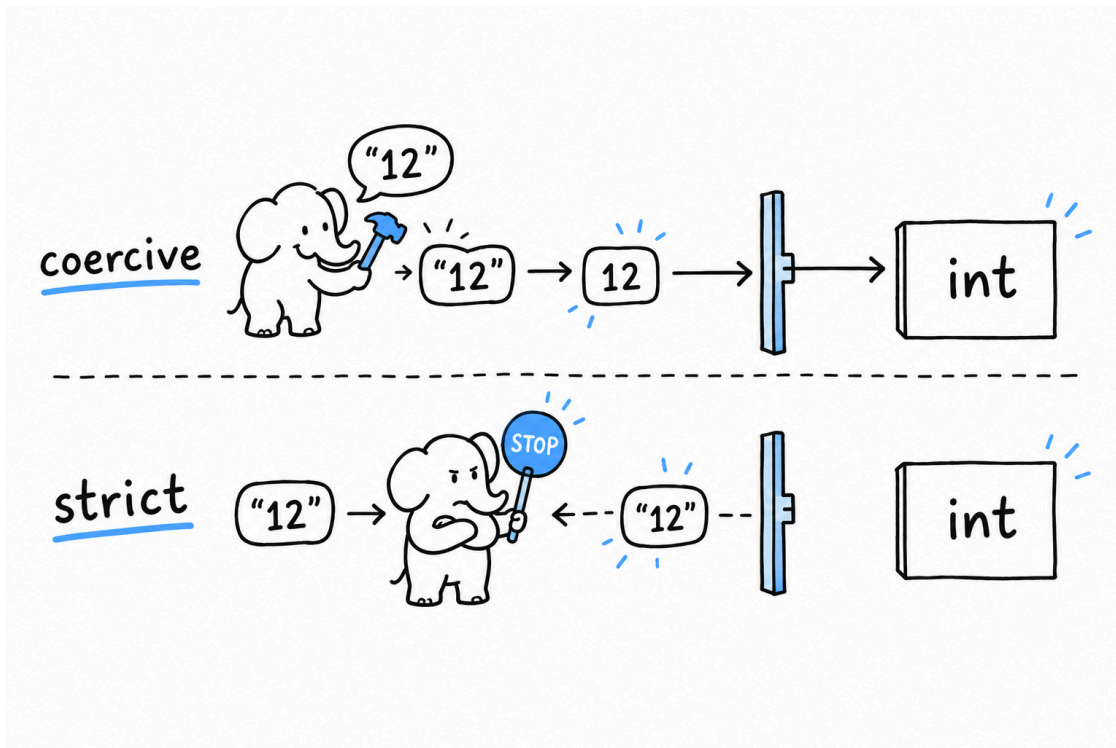
Untyped parameters and properties still exist and mean `mixed`. Modern code does not leave them out.

What `int` really accepts

Here is where PHP differs from everything else. Without `strict_types`, PHP coerces scalar arguments to the declared type when it can. Pass the string `"12"` to an `int` parameter and the function receives the integer `12`. Pass `"12abc"` and you get a `TypeError`. Pass `1.5` and you get `1` with a deprecation notice.

You pass	<code>int</code> parameter, coercive mode	<code>int</code> parameter, strict mode
<code>12</code>	<code>12</code>	<code>12</code>
<code>"12"</code>	<code>12</code>	<code>TypeError</code>
<code>"12abc"</code>	<code>TypeError</code>	<code>TypeError</code>
<code>12.0</code>	<code>12</code>	<code>TypeError</code>
<code>1.5</code>	<code>1</code> , deprecated since 8.1	<code>TypeError</code>
<code>true</code>	<code>1</code>	<code>TypeError</code>
<code>null</code>	<code>TypeError</code> (<code>?int</code> accepts it)	<code>TypeError</code> (<code>?int</code> accepts it)

Coercion only concerns scalars. An array is never turned into a string, an object never into an int. And one conversion is always allowed, even in strict mode: an `int` where a `float` is expected, because that never loses information.



The switch is per file, and per caller

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

function double(int $n): int
{
    return $n * 2;
}

echo double(21), PHP_EOL;    // 42
echo double('21'), PHP_EOL; // TypeError: must be of type int, string
given
```

Delete the `declare` line and the second call prints 42. The way the switch works surprises almost everyone on at least one point:

It is per file. There is no global setting, no `php.ini` flag, no project-wide option. Each file states its own mode, and a file without the line is in coercive mode.

It applies to calls made from the file, not to functions defined in it. If `double()` lives in a strict file and is called from a coercive file, `double('21')` coerces. The caller decides. This is deliberate: a library author cannot force strictness on code that calls it, and legacy code keeps working when it calls a modern library.

It covers return values too. A strict-mode function that declares `: int` and returns `"42"` throws.

The practical rule is short. Put `declare(strict_types=1);` at the top of every file you write, let your code-style tool enforce it, and stop thinking about it.

`strict_types` is not “PHP with types on”. PHP always checks types. The switch decides whether a string that looks like a number counts as a number.

Converting on purpose

When you want a conversion, say so:

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

var_dump((int) '42');           // int(42)
var_dump((int) '42 apples');   // int(42): a cast takes the leading
                                digits
var_dump((int) 'apples');      // int(0)
var_dump((string) 3.0);        // string(1) "3"
var_dump((bool) '0');          // bool(false): "0" is falsy, "0.0" is not
var_dump(intval('0x1A', 16)); // int(26)
```

Casts never throw and never warn; they do their best with what they get. That makes them the right tool for user input you have already validated, and the wrong tool for anything you have not. To check before converting, `is_int()`, `is_string()`, `is_numeric()` and their siblings return booleans, and `filter_var($x, FILTER_VALIDATE_INT)` returns the integer or `false`.

To see what you are holding, `var_dump()` prints the type and value. `get_debug_type()` (PHP 8.0) returns the name you would write in a declaration (`int`, `string`, `App\User`), where the older `gettype()` returns `integer` and `object`.

Null and the standard library

Your own functions reject `null` for a non-nullable parameter in both modes. **Built-in functions are more forgiving, and that forgiveness is on its way out.** `strlen(null)` returns `0` today with a deprecation notice (since PHP 8.1), and the next major version is expected to throw. Code that reads `strlen($_GET['q'])` on a missing parameter is living on borrowed time; `strlen($_GET['q'] ?? '')` is not.

The same tolerance shows up in the other direction. Where a function used to return `false` on failure, newer ones throw a `ValueError` (PHP 8.0), and the ones that still return `false` are documented that way. When the documentation says `string|false`, check for `false` with `===`, never with `if (!$result)`, because `"0"` and `""` are falsy too.

Numbers, briefly

An `int` is 64 bits on every platform you will meet. Overflow does not wrap; **an integer that overflows becomes a float**, silently:

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

var_dump(PHP_INT_MAX + 1); // float(9.223372036854776E+18)
var_dump(0.1 + 0.2 === 0.3); // bool(false), as in every IEEE 754
language
var_dump(intdiv(7, 2), 7 / 2); // int(3), float(3.5)
```

Division always produces a float unless both operands are integers and the result is exact. `intdiv()` gives integer division; `%` is the integer modulo; `fmod()` the float one. For money or anything decimal, the `bcmath` extension and its `BcMath\Number` class (PHP 8.4) do arbitrary-precision arithmetic.

Where the generics went

There are none in the language. `array` is the type of every array, whatever it contains, and a `Collection` class cannot say what it collects. **The ecosystem answered with docblocks read by static analysers**, and on a maintained project that answer is as binding as a compiler:

```

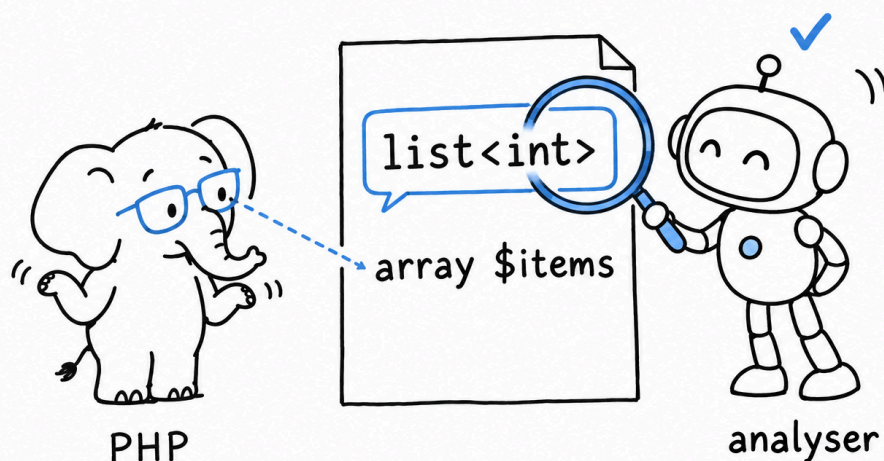
<?php
declare(strict_types=1);

/**
 * @template T
 * @param list<T> $items
 * @param callable(T): bool $keep
 * @return list<T>
 */
function keep(array $items, callable $keep): array
{
    return array_values(array_filter($items, $keep));
}

/** @var list<int> $evens */
$evens = keep([1, 2, 3, 4], fn (int $n) => $n % 2 === 0);

```

PHP reads `array` and `callable`. PHPStan and Psalm read `list<T>` and `callable(T): bool`, infer that `$evens` holds integers, and fail the build if you pass strings. `list<int>`, `array<string, User>`, `non-empty-string`, `int<1, max>`: the vocabulary is richer than the language, and the two tools agree on most of it. [Tests, Static Analysis, and Tooling](#) shows how to set one up.



The trap

The first mistake is to write `function f(int $n)` in a file without `strict_types`, pass `"12"` in a test, watch it work, and conclude that PHP checks nothing. It checked; it also converted. The second is the reverse: adding `declare(strict_types=1)` to one file and expecting the whole application to become strict. Only the calls in that file changed.

Both have the same cure. The line goes in every file, a code-style rule enforces it, and a static analyser catches at build time the cases the runtime would catch in production.

Arrays were mentioned three times in this chapter without a word about what they are. That is because they are not what your language calls an array. [Arrays](#) sets that straight.

Arrays

A PHP array is an ordered hash map, and it is the only built-in collection. It plays the role of Python's list and dict at once, of a JavaScript array and object at once, of Java's `ArrayList` and `LinkedHashMap` at once. Keys are integers or strings, values are anything, and insertion order is always preserved.

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

$list = ['apple', 'pear'];           // keys 0, 1
$map  = ['name' => 'Ada', 'born' => 1815]; // string keys
$mixed = [5 => 'five', 'six', 'x' => 'ex']; // keys 5, 6, 'x'

$list[] = 'plum';           // append, key 2
$map['died'] = 1852;        // insert, at the end

var_dump(array_is_list($list)); // true
var_dump(array_is_list($map));  // false
```

There is no separate list type. A “list” is an array whose keys happen to be 0, 1, 2 and so on, in that order, and `array_is_list()` (PHP 8.1) tells you whether that holds. The distinction matters when the array leaves PHP: `json_encode()` emits `[...]` for a list and `{...}` for anything else.

Keys are normalised

A key is either an `int` or a `string`, and PHP converts everything else on the way in. A numeric string becomes the integer it names, a float loses its decimals, a boolean becomes 0 or 1, and `null` becomes the empty string:

```

<?php
declare(strict_types=1);

$a = [];
$a['1'] = 'a';    // key 1, not '1'
$a[1.7] = 'b';    // key 1, overwrites (and a deprecation notice since
8.1)
$a[true] = 'c';   // key 1, overwrites again
$a[null] = 'd';   // key ''

var_dump($a); // [1 => 'c', '' => 'd']

```

Three writes, one key. The rule is convenient when a database returns ids as strings, and a trap when you expected `'1'` and `1` to be two entries. They never are.

Reading a key that is not there

Reading a missing key emits a warning and yields `null`. Two functions tell you whether a key exists, and they disagree about `null`:

```

<?php
declare(strict_types=1);

$user = ['name' => 'Ada', 'email' => null];

var_dump(isset($user['email']));           // false: the value is null
var_dump(array_key_exists('email', $user)); // true: the key is there
var_dump(isset($user['phone']));           // false, no warning

$phone = $user['phone'] ?? 'unknown';      // no warning, default
applied

```

`isset()` answers “is there a non-null value here?”, `array_key_exists()` answers “is the key present?”. For almost everything, `??` is what you want: it reads the key if it exists and is not `null`, and falls back otherwise, silently.

Arrays are values

This is the one to remember. **Assigning an array copies it. Passing an array to a function copies it. Returning one copies it.** The original never sees what happens to the copy.

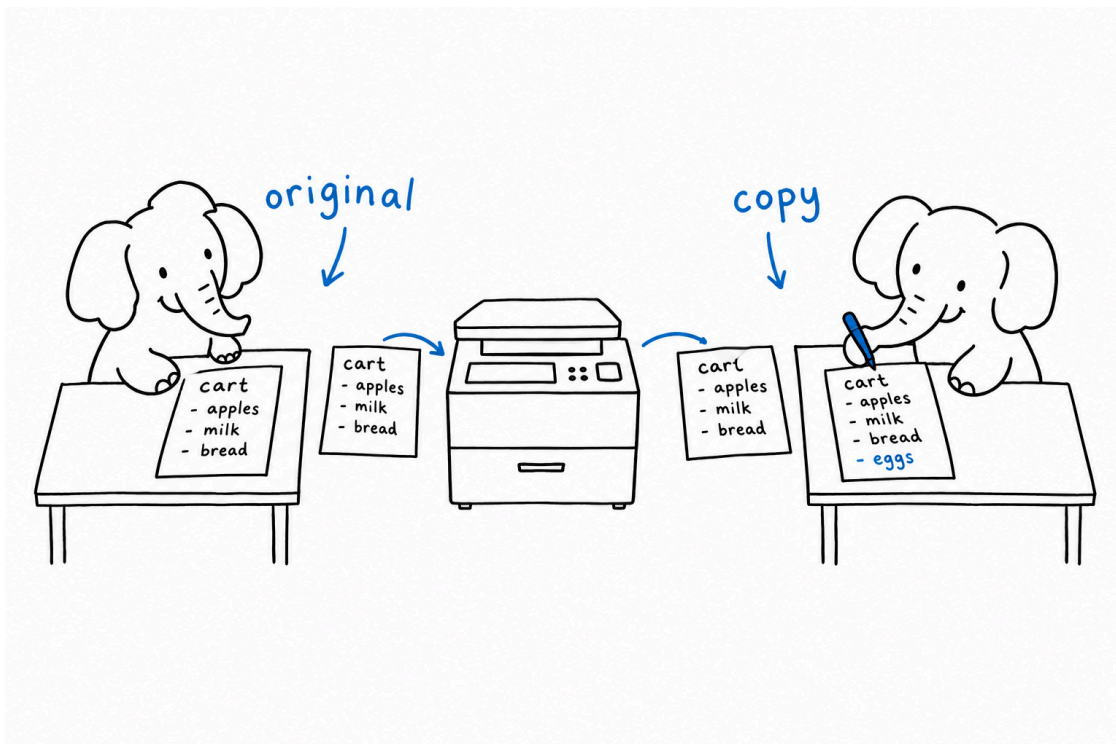
```
<?php
declare(strict_types=1);

function addItem(array $cart, string $item): array
{
    $cart[] = $item;
    return $cart;
}

$cart = ['book'];
$bigger = addItem($cart, 'pen');

var_dump(count($cart)); // 1
var_dump(count($bigger)); // 2
```

In JavaScript, Python or Java, `cart` would now hold two items, because those languages hand around a reference to one shared structure. In PHP the function got its own array, and to give you the result it must return it.



The cost is smaller than it sounds. Under the hood PHP shares the memory and only duplicates the data at the first write, a scheme called copy-on-write. Passing a ten-thousand-element array to a function that only reads it costs nothing.

You can opt out with a reference, `&`, on the parameter:

```
function addItemInPlace(array &$cart, string $item): void
{
    $cart[] = $item;
}
```

Reserve it for the rare hot loop where the copy is measurable. A function that returns a new array is easier to read, to test and to type, and the `sort()` family, which mutates in place through references, is the legacy exception, not the model.

Objects behave the other way round: **an object variable is a handle, and copies of the handle point at the same object**, as in every language you know. When you need reference semantics for a collection, wrap it in a class. [Classes](#) covers that.

foreach, by value and by reference

`foreach` iterates over a copy, so mutating `$item` inside the loop changes nothing:

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

$prices = [10, 20, 30];

foreach ($prices as $price) {
    $price *= 2; // local copy, the array is untouched
}

foreach ($prices as &$price) {
    $price *= 2; // writes through
}
unset($price); // break the reference

var_dump($prices); // [20, 40, 60]
```

The `unset()` after a by-reference loop is not decoration. Without it, `$price` still points at the last element, and the next innocent `$price = 0;` overwrites `$prices[2]`. Most PHP developers have been bitten by this once. A `foreach` with `$key => $value` and a write to `$prices[$key]` avoids the whole question, and so does `array_map()`.

Taking arrays apart and putting them together

Destructuring works on lists and on maps:

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

[$x, $y] = [3, 4];
['id' => $id, 'name' => $name] = ['id' => 7, 'name' => 'Ada'];
[, $second] = ['skip', 'keep']; // holes are allowed

$defaults = ['color' => 'blue', 'size' => 'M'];
$order = [...$defaults, 'size' => 'L']; // string keys spread since 8.1

var_dump($order); // ['color' => 'blue', 'size' => 'L']
```

The spread with string keys behaves like JavaScript's `{...defaults, size: 'L'}`: later entries win. With integer keys, spreading rennumbers, so `[...[1, 2], ...[3]]` is `[1, 2, 3]`, not a map with duplicate keys.

The functional trio and its cousins

`array_map()`, `array_filter()` and `array_reduce()` do what their names say, with a wrinkle each:

```

<?php
declare(strict_types=1);

$orders = [
    ['id' => 1, 'total' => 40, 'paid' => true],
    ['id' => 2, 'total' => 15, 'paid' => false],
    ['id' => 3, 'total' => 90, 'paid' => true],
];

$totals = array_map(fn(array $o) => $o['total'], $orders);      //
[40, 15, 90]
$paid    = array_filter($orders, fn(array $o) => $o['paid']);  //
keys 0 and 2
$sum     = array_reduce($totals, fn(int $carry, int $t) => $carry + $t,
0); // 145

echo json_encode($paid);                                     // {"0":{"..."},"2":{"..."}} an
object!
echo json_encode(array_values($paid)); // [{...},{...}]      a
list

```

array_filter() keeps the original keys. After filtering a list, the keys have holes, `array_is_list()` says false, and `json_encode()` produces an object. `array_values()` renumbers. Watch the argument order too: the array comes first for `array_filter()` and `array_reduce()`, the callback first for `array_map()`. That inconsistency is thirty years old, and your editor's autocomplete is the cure.

PHP 8.4 added the searches you kept writing by hand: `array_find()` returns the first matching element, `array_find_key()` its key, `array_any()` and `array_all()` return booleans. PHP 8.5 added `array_first()` and `array_last()`, which return the first and last values regardless of keys, next to the older `array_key_first()` and `array_key_last()`.

```

// PHP 8.4
$firstBig = array_find($orders, fn(array $o) => $o['total'] > 50);
$allPaid  = array_all($orders, fn(array $o) => $o['paid']); // false

```

Sorting mutates in place and, since PHP 8.0, is stable. `usort()` with the spaceship operator is the idiom:

```
usort($orders, fn(array $a, array $b) => $b['total'] <=> $a['total']);
```

`sort()` and `usort()` renumber the keys; `asort()` and `uasort()` keep them; `ksort()` sorts by key. `array_column($orders, 'total', 'id')` pulls one field out of a list of rows and, given the third argument, indexes the result by another. `array_combine()`, `array_flip()`, `array_unique()`, `array_slice()` and `array_splice()` are there too, with `count()` for the length.

You will also meet `compact()` and `extract()`, which turn local variables into an array and back. Recognise them, do not write them: they defeat static analysis and your editor.

Iterating anything

`foreach` is not limited to arrays. **Anything iterable works: arrays, generators, and objects implementing `Iterator` or `IteratorAggregate`**. A function that accepts `iterable` can be fed a million-row generator without loading the million rows, which [Functions and Closures](#) picks up.

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

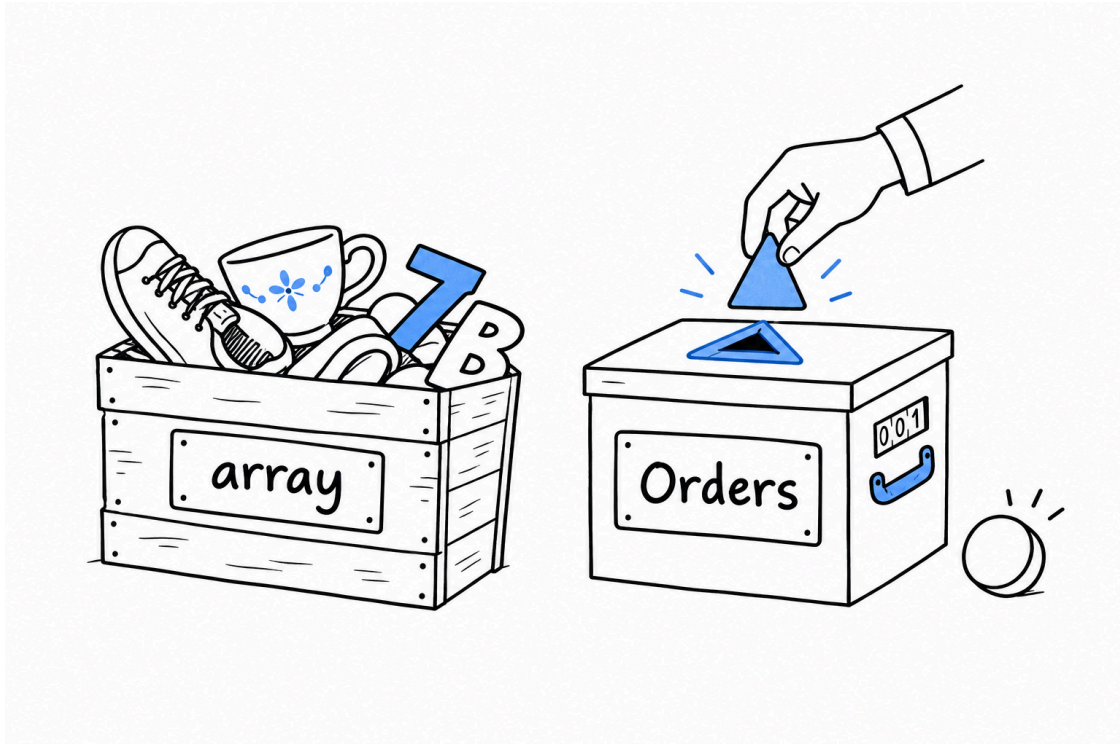
function total(iterable $amounts): int
{
    $sum = 0;
    foreach ($amounts as $amount) {
        $sum += $amount;
    }
    return $sum;
}

echo total([1, 2, 3]); // 6
```

When an array is not enough

An array cannot say what it contains. `array $orders` tells the reader nothing, and the language has no `array<Order>`. Two answers coexist. The lightweight one is a docblock, `@param list<Order> $orders`, which PHPStan and Psalm enforce as if it were a real type and your editor uses for completion. The heavier one is a small class, a `final class Orders` holding a private array, exposing exactly the operations you need, and implementing the interfaces that let

it behave like an array where useful: `Countable` for `count()`, `ArrayAccess` for `$orders[0]`, `IteratorAggregate` for `foreach`.



Two built-in classes cover cases arrays cannot. `SplObjectStorage` maps objects to data using the object itself as key. `WeakMap` (PHP 8.0) does the same without keeping the object alive, which is how caches keyed by entity avoid leaking.

Two habits to break: expecting a function to mutate the array you pass it, and forgetting that `array_filter()` leaves holes. Return the new array, and wrap in `array_values()` before encoding.

Arrays are what most PHP code passes around. Functions are what it passes them to, and PHP's closures capture differently from the ones you know. That is [Functions and Closures](#).

Functions and Closures

PHP functions look like TypeScript's with a `$` on every parameter, and closures capture by value, not by variable. That second half is where the surprises live. The first half you can read at speed.

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

function greet(string $name, string $greeting = 'Hello', bool $shout =
false): string
{
    $text = "$greeting, $name!";
    return $shout ? strtoupper($text) : $text;
}

echo greet('Ada'); // Hello, Ada!
echo greet('Ada', shout: true); // HELLO, ADA!
echo greet(greeting: 'Hi', name: 'Ada'); // Hi, Ada!
```

Parameters and return values carry types, checked at runtime as [Types](#) explained. Defaults work as everywhere. **Named arguments (PHP 8.0)** let you skip the defaults you do not care about and make a call with four booleans readable. Positional and named can mix, positional first.

Variadics, references, and the odd return types

`...` on the last parameter collects the rest into an array; `...` in a call unpacks an array into arguments, string keys included, which turns an array into named arguments:

```

<?php
declare(strict_types=1);

function sum(int ...$numbers): int
{
    return array_sum($numbers);
}

echo sum(1, 2, 3);           // 6
echo sum(...[4, 5]);        // 9

$options = ['greeting' => 'Hey', 'name' => 'Ada'];
// greet(...$options) would call greet(name: 'Ada', greeting: 'Hey')

```

A parameter declared `&$x` receives a reference: the function writes into the caller's variable. The standard library uses it for `sort()`, `preg_match()`'s `$matches` and a few others. In your own code, prefer returning the value. A function that changes its arguments is one your reader has to open to understand.

Parameter types take everything the type system offers: `?string $label = null` for an optional value, `int|string $id` for a union, `Countable&Traversable $items` for an intersection. Write the `?` explicitly; a plain `string $x = null` still works but has been deprecated since PHP 8.4. For a function that accepts a function, two declarations exist: `callable` accepts closures and also the string and array forms below; `Closure` accepts only real closure objects. New code tends to declare `Closure` and let callers convert with `(...)`, because a `Closure` can be type-checked and a string cannot.

Two return types describe functions that do not return normally. `void` means nothing comes back. `never` (PHP 8.1) means the function always throws or exits, so static analysers know that code after a `fail()` call is unreachable.

Functions are not values, but you can grab a handle

A function name is not an expression. `$f = strlen;` is a syntax error, and the historical workaround was a string: `$f = 'strlen';` works because the `callable` type accepts a function name, a `[$object, 'method']` pair, or a `'Class::method'` string. That works, and nothing checks it until it runs.

The modern way is the first-class callable syntax (PHP 8.1): the name followed by `(...)`.


```

<?php
declare(strict_types=1);

final class Mailer
{
    public function send(string $to): string
    {
        return "sent to $to";
    }
}

$length = strlen(...); // Closure wrapping strlen()
$send = (new Mailer())->send(...); // Closure bound to that
instance

echo $length('hello'); // 5
echo $send('ada@example.org');

var_dump(array_map(strtoupper(...), ['a', 'b'])); // ['A', 'B']

```

The result is a `Closure` object, PHP's only function value. It is type-safe, refactor-safe (rename the method and your editor follows), and it is what to pass to `array_map()` and friends.

`Closure::fromCallable('strlen')` does the same from a string when you have one.

A polyglot's reflex is `$this->send` without parentheses. In PHP that reads the property `send`, which does not exist. The method as a value is `$this->send(...)`.

Closures capture by value

Anonymous functions exist, and you have to say what they capture:

```

<?php
declare(strict_types=1);

$rate = 0.2;

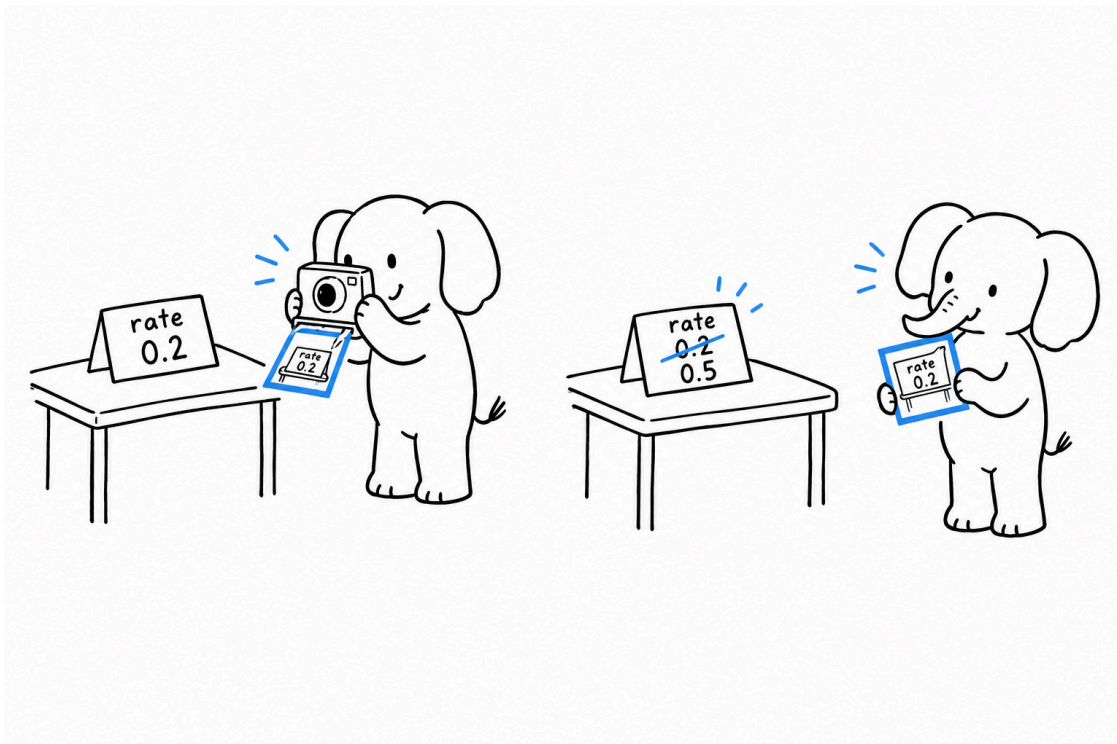
$withTax = function (float $price) use ($rate): float {
    return $price * (1 + $rate);
};

$rate = 0.5; // too late, the closure already copied 0.2

echo $withTax(100.0); // 120

```

The **use** clause copies the variables at the moment the closure is created. Nothing in the enclosing scope is visible unless listed, and later changes to the outer variable do not reach the closure. JavaScript and Python close over the variable itself and would print 150 here. PHP hands the closure a snapshot.



To capture the variable rather than its value, add **&**: `use (&$rate)`. Now the closure and the outer scope share it, in both directions. The usual need for it is an accumulator, or a recursive closure that must see itself:

```
$fact = function (int $n) use (&$fact): int {  
    return $n <= 1 ? 1 : $n * $fact($n - 1);  
};
```

Arrow functions (PHP 7.4) drop the ceremony. `fn` captures the whole enclosing scope automatically, by value, and holds a single expression:

```
$withTax = fn(float $price): float => $price * (1 + $rate);
```

No `use`, no `return`, no braces, and the same snapshot semantics. Most callbacks you write will be arrow functions. Reach for `function () use ()` when you need statements, or a by-reference capture.

Inside a class, a closure keeps `$this` automatically, which is what you expect. Mark it `static fn` or `static function` when it does not need the instance; that avoids keeping the object alive from inside a long-lived callback. `Closure::bind()` and `$closure->call($object)` rebind `$this` to another object, and are how frameworks reach private state from outside. You will rarely write them yourself.

Generators

A function containing `yield` returns a `Generator` without running its body. Each `foreach` step runs it to the next `yield`. This is Python's generator, nearly line for line:

```

<?php
declare(strict_types=1);

/** @return Generator<int, string> */
function lines(string $path): Generator
{
    $handle = fopen($path, 'r');
    try {
        while (($line = fgets($handle)) !== false) {
            yield rtrim($line, "\n");
        }
    } finally {
        fclose($handle);
    }
}

foreach (lines('/etc/hosts') as $number => $line) {
    echo "$number: $line", PHP_EOL;
}

```

The file is read one line at a time, however large it is, and closed when the loop ends or breaks. A generator is **iterable**, so any function accepting **iterable** takes it without knowing.

`yield $key => $value` sets explicit keys, `yield from` delegates to another generator or array, and a `return` inside a generator sets a value readable through `getReturn()` once iteration is over.

Generators run once; to iterate again, call the function again.

Pipelines

PHP 8.5 adds the pipe operator. `$x |> f(...)` calls `f($x)`, and chains read top to bottom instead of inside out:

```

// PHP 8.5
$slug = ' Hello World '
    |> trim(...)
    |> strtolower(...)
    |> (fn(string $s) => str_replace(' ', '-', $s));

echo $slug; // hello-world

```

Each stage is any callable taking one argument, which is exactly what the first-class callable syntax and arrow functions produce. Before 8.5 the same code is three nested calls or three temporary variables; both still work, and both are still common.

PHP 8.5 also brings `#[\NoDiscard]`, an attribute for functions whose return value must not be dropped. Call such a function as a bare statement and PHP emits a warning; cast the call to `(void)` to say you meant it. Libraries use it on methods returning a new immutable object, to catch the classic `$date->modify()` bug where the result is thrown away.

Legacy shapes you will recognise

`func_get_args()` and `func_num_args()` read the arguments of a function declared without parameters. They predate `...$args` and survive in old code. `call_user_func()` and `call_user_func_array()` invoke a callable; `$callable(...$args)` does the same today. `create_function()` built closures from strings and was removed in 8.0. When you meet these, the modern replacement is one line away, and Rector can make the edit for you.

Closures snapshot their `use` variables, arrow functions snapshot everything, and a method becomes a value with `(...)`. Keep those three straight and PHP callbacks hold no further surprises.

Functions carry behaviour. The data they act on is mostly objects, and PHP's object model has changed more in the last five years than in the fifteen before. [Classes](#) shows what it looks like now.

Classes

A modern PHP class is short, typed, and mostly immutable. The constructor declares the properties, the types are checked at runtime, and most of the boilerplate you remember from Java, or from PHP 5, is gone. Here is one, whole:

```

<?php
declare(strict_types=1);

namespace App\Billing;

final class Invoice
{
    private array $lines = [];

    public function __construct(
        public readonly string $number,
        public readonly \DateTimeImmutable $issuedAt,
    ) {
    }

    public function addLine(string $label, int $cents): static
    {
        $this->lines[] = ['label' => $label, 'cents' => $cents];

        return $this;
    }

    public function total(): int
    {
        return array_sum(array_column($this->lines, 'cents'));
    }
}

$invoice = new Invoice('2026-0042', new \DateTimeImmutable('2026-09-14'));
$invoice->addLine('Hosting', 1200)->addLine('Support', 800);

echo $invoice->number, ': ', $invoice->total(), PHP_EOL; // 2026-0042:
2000

```

Read it top to bottom. `namespace` puts the class in `App\Billing`, so its full name is `App\Billing\Invoice` (the [Composer chapter](#) explains how that maps to a file). `final` forbids subclassing; modern PHP code makes classes `final` by default and opens them on purpose. The constructor has no body: writing `public readonly string $number` in the parameter list declares

the property, types it, and assigns it, all in one line. This is constructor promotion (PHP 8.0), and it removed most of the ceremony from PHP classes. `readonly` (PHP 8.1) makes the property assignable once, in the constructor, and never again. `static` as a return type means “the class of the object this was called on”, which is what a fluent method wants.

Two symbols do all the member access. `->` reaches an instance member, `::` reaches a static one or a constant. `$this` is the current object, and you always write it: there is no implicit `this`.

Objects travel by handle

Arrays copy when you assign them (the [Arrays chapter](#) has the details). Objects do not. **Assigning or passing an object copies a handle to the same object**, the way Java, Python and JavaScript do it:

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

final class Cart
{
    public array $items = [];
}

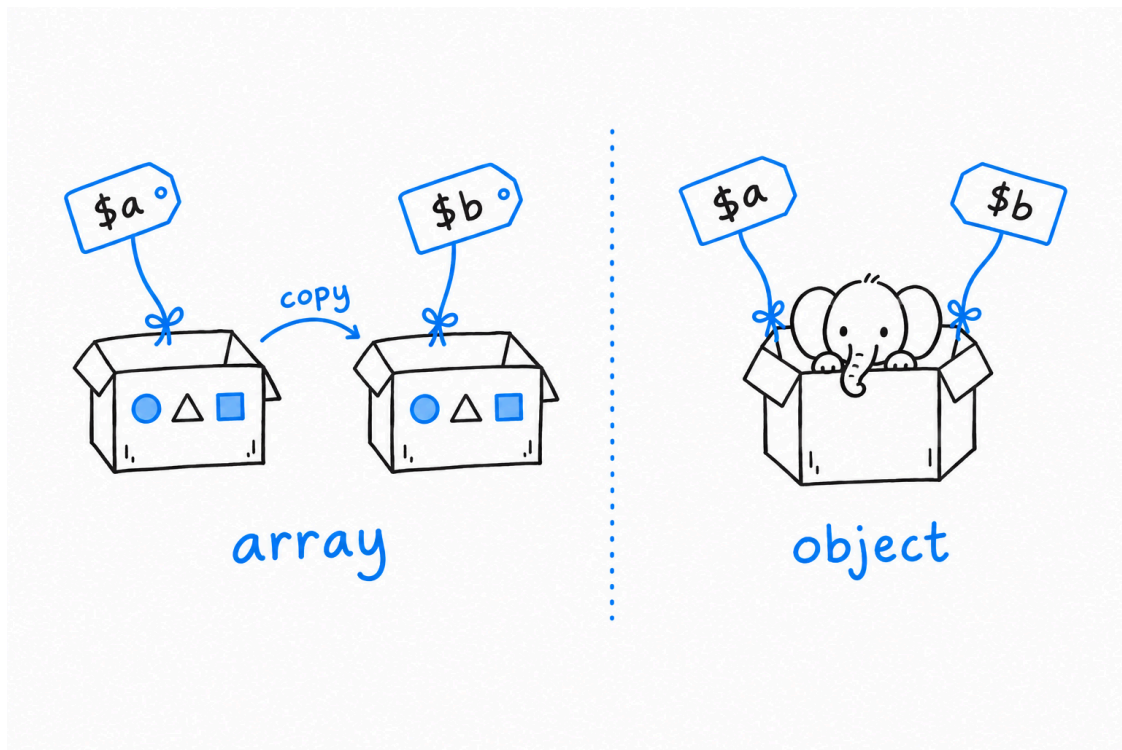
function addApple(Cart $cart): void
{
    $cart->items[] = 'apple';
}

$cart = new Cart();
addApple($cart);
echo count($cart->items), PHP_EOL; // 1, the caller sees the change
```

No `&` is involved: reference parameters (`&$x`) are a different mechanism, for variables, and you almost never need them with objects.

`clone $cart` makes a shallow copy: a new object whose properties hold the same values, so an object stored inside is shared between the two copies. Define `__clone()` if the copy needs its own inner objects.

Two comparisons exist. `==` compares state, property by property; `===` asks whether both sides are the same object. `$a === clone $a` is `false`.



Immutability without getters

The classic PHP 5 class had a private property, a getter, and sometimes a setter. Three modern features replace that pattern, and you will see all three in codebases written after 2024.

`readonly` you have met. When every property is readonly, mark the class instead (PHP 8.2):

```

<?php
declare(strict_types=1);

final readonly class Money
{
    public function __construct(
        public int $amount,
        public string $currency,
    ) {
    }

    public function add(Money $other): self
    {
        return new self($this->amount + $other->amount, $this->currency);
    }
}

```

Every property is public and nobody can change it. You get a value object with no getters at all.

Asymmetric visibility (PHP 8.4) lets the outside read a property that only the class can write:

```

// PHP 8.4
final class Counter
{
    public private(set) int $count = 0;

    public function increment(): void
    {
        $this->count++;
    }
}

$c = new Counter();
$c->increment();
echo $c->count; // 1
$c->count = 5; // Error: Cannot modify private(set) property
Counter::$count

```

That is the getter, gone. Where you need logic on read or write, property hooks (PHP 8.4) attach it to the property itself, like a C# property or Python's `@property`:

```
// PHP 8.4
final class User
{
    public string $email {
        set(string $value) {
            if (!filter_var($value, FILTER_VALIDATE_EMAIL)) {
                throw new \InvalidArgumentException("Invalid email:
$value");
            }
            $this->email = strtolower($value);
        }
    }

    public string $domain {
        get => substr($this->email, strpos($this->email, '@') + 1);
    }
}

$u = new User();
$u->email = 'Ada@example.org';
echo $u->domain; // example.org
```

To the caller these are plain properties. Inside, `set` validates and normalises, and `get` computes. A property with only a `get` hook and no backing store is a computed property.

Interfaces, abstract classes, traits

Interfaces and abstract classes work as they do in Java and C#: an interface lists method signatures and constants, an abstract class may carry implementation, a class implements many interfaces and extends one parent. `#[\Override]` (PHP 8.3) on a method makes PHP check that the parent really has it, which catches typos in overrides.

Traits are the part with no exact equivalent in most languages. **A trait is a block of methods and properties that the compiler copies into every class that uses it.** Ruby's mixins are the closest anchor; Rust's traits are not, despite the name.

```

<?php
declare(strict_types=1);

trait HasTimestamps
{
    private ?\DateTimeImmutable $createdAt = null;

    public function touch(): void
    {
        $this->createdAt ??= new \DateTimeImmutable();
    }
}

final class Article
{
    use HasTimestamps;
}

$a = new Article();
$a->touch();

```

Traits are handy for cross-cutting helpers and easy to overuse. A class that uses five traits is five files you have to read to know what it does. Prefer composition where you can.

`static` versus `self` is a one-paragraph subject. `self` names the class where the code is written; `static` names the class of the object at runtime. In a static factory method inside a parent class, `new static()` builds the subclass that was actually called; `new self()` always builds the parent.

Construction

PHP has one constructor per class and no method overloading. Two idioms fill the gap: named arguments (PHP 8.0) for optional parameters, and static factory methods for alternative ways to build:

```

<?php
declare(strict_types=1);

final readonly class Period
{
    private function __construct(
        public \DateTimeImmutable $start,
        public \DateTimeImmutable $end,
    ) {
    }

    public static function fromStrings(string $start, string $end):
self
    {
        return new self(new \DateTimeImmutable($start), new
\DateTimeImmutable($end));
    }

    public static function year(int $year): self
    {
        return self::fromStrings("$year-01-01", "$year-12-31");
    }
}

$fy = Period::year(2026);
echo $fy->end->format('Y-m-d'), PHP_EOL; // 2026-12-31

```

A private constructor plus public factories reads as well as a set of overloaded constructors, and it names each variant.

Since PHP 8.4 you can chain a call on a fresh object without wrapping it in parentheses:

```
new Period(...)->start
```

used to need `(new Period(...))->start`.

For immutable objects, “change” means “make a modified copy”. PHP 8.5 gives that its own syntax: `clone($money, ['amount' => 500])` returns a copy with the listed properties replaced. The usual visibility rules apply, and a `readonly` property can only be written from inside its class, so the call lives in a `withAmount()` method rather than at the call site. Before 8.5, that same method cloned and assigned inside `__clone()`, which PHP 8.3 permitted for readonly properties.

Strings, magic, and metadata

Any object can print itself by implementing `__toString()`. Declaring it makes the class implement `Stringable` (PHP 8.0) automatically, so you can type a parameter as `string|Stringable` and accept both.

The other double-underscore methods are hooks the engine calls: `__get` and `__set` run when code touches a property that does not exist, `__call` when it calls a method that does not exist.

Frameworks and ORMs use them to build fluent APIs and lazy models. Recognise them; do not reach for them in application code. Related: creating a property that was never declared (`$obj->foo = 1` with no `$foo` in the class) has been deprecated since PHP 8.2 and is planned to become an error in the next major version. Declare your properties.

Attributes (PHP 8.0) are structured metadata on a class, method, property or parameter, read through reflection. Java annotations and C# attributes are the direct anchor:

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

#[\Attribute(\Attribute::TARGET_METHOD)]
final readonly class Route
{
    public function __construct(public string $path) {}
}

final class HomeController
{
    #[Route('/')]
    public function index(): string
    {
        return 'Hello';
    }
}

$method = new \ReflectionMethod(HomeController::class, 'index');
foreach ($method->getAttributes(Route::class) as $attribute) {
    echo $attribute->newInstance()->path, PHP_EOL; // /
}
```

An attribute is itself a class marked `#[\Attribute]`. Nothing happens at runtime until something calls `getAttributes()`; the metadata is inert until read. Routing, validation, serialisation and test

frameworks all build on this.

`Invoice::class` yields the fully qualified name as a string, which is what you pass around instead of hardcoding `'App\Billing\Invoice'`. `$x instanceof Invoice` checks type at runtime and is `false` rather than an error when `$x` is not an object. Lazy objects (PHP 8.4) let a class be instantiated without running its constructor until a property is first touched, a tool for dependency injection containers and ORMs rather than everyday code.

There are no generics. A `Collection` class holds `mixed` as far as the engine is concerned; the `@template` docblocks described in [Types](#) give PHPStan and Psalm what the engine lacks.

The trap

A function that receives an object and “just tweaks it a bit” tweaks the caller’s object too. If you meant a local modification, `clone` first, or design the class as readonly and return a new instance.

And `readonly` is shallow. It freezes the property slot, not what the slot points to. A `readonly array $items` cannot be reassigned, but a `readonly Cart $cart` still lets anyone holding `$cart` push items into it. Immutability of the whole graph is a design decision, not a keyword.

With classes in hand, the question is how PHP represents a fixed set of choices. [Enums and match](#) answers it.

Enums and match

A PHP enum is a class with a fixed set of instances, and `match` is the expression that picks one branch per instance. Together they replace the class-constants-and-`switch` pattern that older PHP code is full of. If you know Java enums, you know most of this already; if you come from Python's `Enum` or from TypeScript's string literal unions, the shape will look familiar and the checks will be stricter.

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

enum Status: string
{
    case Draft = 'draft';
    case Published = 'published';
    case Archived = 'archived';

    public function label(): string
    {
        return match ($this) {
            self::Draft => 'Draft',
            self::Published => 'Live',
            self::Archived => 'Archived',
        };
    }

    public function canEdit(): bool
    {
        return $this !== self::Archived;
    }
}

$status = Status::from('published'); // Status::Published
echo $status->label(), PHP_EOL;      // Live
echo $status->value, PHP_EOL;        // published
var_dump($status->canEdit());         // bool(true)
var_dump(Status::tryFrom('deleted')); // NULL
```

Cases, values, and methods

An enum (PHP 8.1) declares its `cases` and nothing else can be one. `Status::Draft` is an object, the only object of its kind, so **two references to the same case are the same object and `===` is the comparison to use**. Enums cannot be instantiated with `new`, cannot be extended, and cannot hold state: no properties, no per-instance data. What they can have is methods, constants, static methods, and interfaces to implement.

The `: string` after the name makes this a backed enum: each case carries a scalar (`string` or `int`) reachable through `->value`. `from()` turns a scalar back into a case and throws a `ValueError` when nothing matches; `tryFrom()` returns `null` instead. That pair is how an enum crosses a boundary: a database column, a JSON field, a query string. A pure enum, declared without a backing type, has cases and no `->value`; use it when the choice never leaves your code.

`cases()` returns every case in declaration order, which is what a dropdown or a validation rule wants:

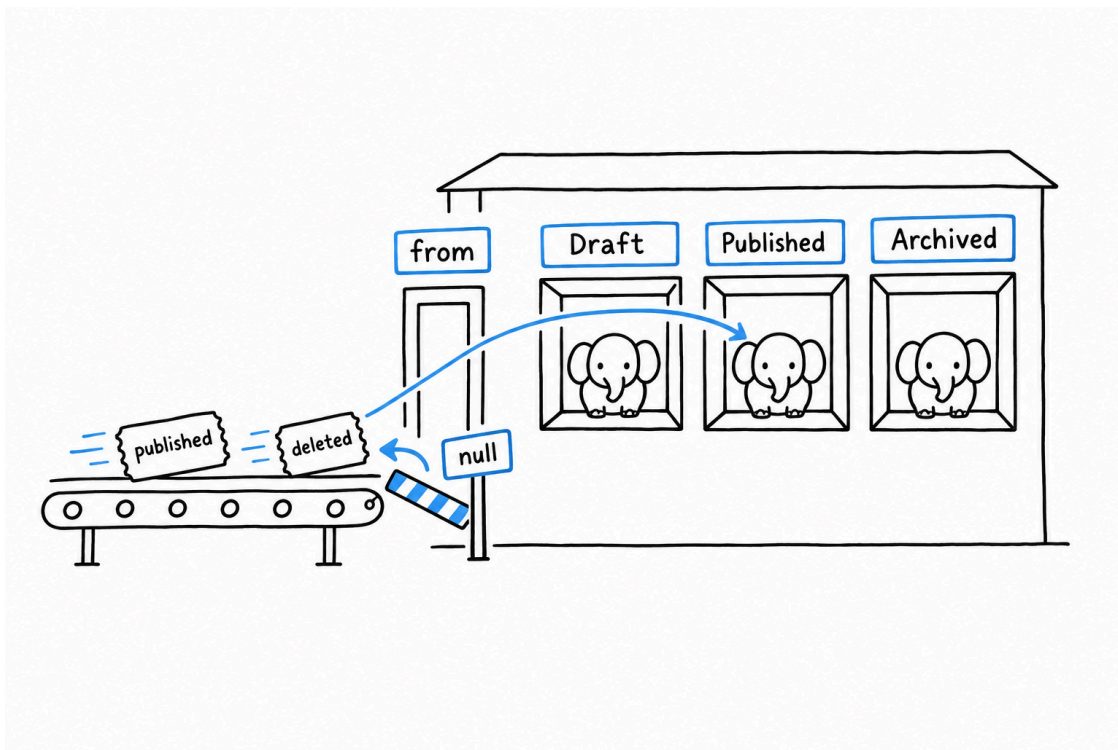
```
$allowed = array_map(fn (Status $s) => $s->value, Status::cases());  
// ['draft', 'published', 'archived']
```

Every case also has `->name` (`'Draft'`), useful for logging.

Because `$this` inside an enum method is a case, behaviour that depends on the case belongs on the enum, not in `if` chains scattered across the code. `label()` and `canEdit()` above are the pattern: the enum knows what each of its cases means.

Since PHP 8.2 enum cases are allowed in constant expressions, so they can be default parameter values, class constants, and attribute arguments:

```
public function __construct(private Status $status = Status::Draft).
```



match

`match` (PHP 8.0) looks like `switch` and behaves like an expression in Rust or a `when` in Kotlin. The rules that follow are what set it apart.

It returns a value. `$label = match ($status) { ... };` and `return match (...)` are the normal uses. There is no `break`, because there is no fallthrough: exactly one arm runs.

It compares with `===`. `match ('1') { 1 => 'int', '1' => 'string' }` picks the second arm. `switch` would have picked the first.

It must be exhaustive. If no arm matches and there is no `default`, PHP throws `UnhandledMatchError`. Add a case to an enum, forget to update a `match` on it, and the first time that case reaches the `match` you get an exception naming the exact line, rather than a silent `null`.

One arm can list several values, separated by commas:

```

<?php
declare(strict_types=1);

enum Status: string
{
    case Draft = 'draft';
    case Published = 'published';
    case Archived = 'archived';
}

function isVisible(Status $status): bool
{
    return match ($status) {
        Status::Published => true,
        Status::Draft, Status::Archived => false,
    };
}

```

Static analysers understand this: PHPStan and Psalm both report a `match` over an enum that leaves a case unhandled, before it ever runs.

The subject of a `match` does not have to be an enum. Any value works, and the `match (true)` idiom turns it into a condition ladder that yields a value:

```

$size = match (true) {
    $bytes < 1024 => 'small',
    $bytes < 1024 * 1024 => 'medium',
    default => 'large',
};

```

Each arm is compared with `===` against `true`, so each arm is a boolean expression. It reads better than a nested ternary and cannot fall through.

`switch` still exists, with loose comparison, `fallthrough`, and `break`. You will meet it in older code. In new code, `match` is the right default, and `switch` is for the rare case where you actually want several labels to share a block of statements.

Nulls and throws as expressions

Two operators pair naturally with `match` and enums.

The nullsafe operator `?->` (PHP 8.0) short-circuits a chain when the left side is `null`:

`$order?->customer?->email` is `null` if any link is `null`, instead of an error. Combined with `??` it gives a default in one line: `$email = $order?->customer?->email ?? 'nobody@example.org';`.

`throw` is an expression (PHP 8.0), so it can sit on the right of `??`, in a ternary, or in a `match` arm:

```
$status = Status::tryFrom($input) ?? throw new
\InvalidArgumentException("Unknown status: $input");

$handler = match ($status) {
    Status::Draft => $this->saveDraft(...),
    Status::Published => $this->publish(...),
    Status::Archived => throw new \LogicException('Archived items are
read-only'),
};
```

The second example also shows the pattern for dispatching: a `match` that returns a callable, then `$handler($item)`.

Persistence and behaviour in one place

Here is the full shape, the way it appears in an application: a backed enum for storage, methods for behaviour, `match` where the branches live.

```

<?php
declare(strict_types=1);

interface HasColor
{
    public function color(): string;
}

enum Priority: int implements HasColor
{
    case Low = 1;
    case Normal = 2;
    case High = 3;

    public const DEFAULT = self::Normal;

    public static function fromLabel(string $label): self
    {
        return match (strtolower($label)) {
            'low' => self::Low,
            'normal', 'medium' => self::Normal,
            'high', 'urgent' => self::High,
            default => throw new \ValueError("Unknown priority:
$label"),
        };
    }

    public function color(): string
    {
        return match ($this) {
            self::Low => 'grey',
            self::Normal => 'blue',
            self::High => 'red',
        };
    }

    public function escalate(): self
    {
        return match ($this) {
            self::Low => self::Normal,

```

```

        self::Normal, self::High => self::High,
    };
}
}

$p = Priority::fromLabel('medium');
echo $p->color(), PHP_EOL;           // blue
echo $p->escalate()->name, PHP_EOL;   // High
echo Priority::DEFAULT->value, PHP_EOL; // 2

```

The integer goes in the database. The enum goes everywhere else. There is no `PRIORITY_HIGH = 3` constant to keep in sync with a lookup table of colours somewhere, because the case and its behaviour live in one file.

Compared to the older pattern, class constants plus string flags plus a `switch`, an enum gives you a real type to put in a signature (`function assign(Priority $p)`), which the engine checks, and a closed set the analyser can reason about.

The trap

Two habits from other languages, and from old PHP, cause the same bug.

Reaching for `switch`: it compares loosely, so `switch (Status::Draft)` with `case 'draft':` never matches (an enum is not equal to its value, under either comparison), and with a plain string subject a `case 0:` matches more than you expect.

Comparing an enum to its backing value: `$status == 'published'` is always `false`. The enum is an object; the string is what it stores. Compare cases to cases (`$status === Status::Published`), or convert first (`$status->value === 'published'`).

The next question a polyglot asks is what happens when something goes wrong. [Errors and Exceptions](#) covers a model that has both.

Errors and Exceptions

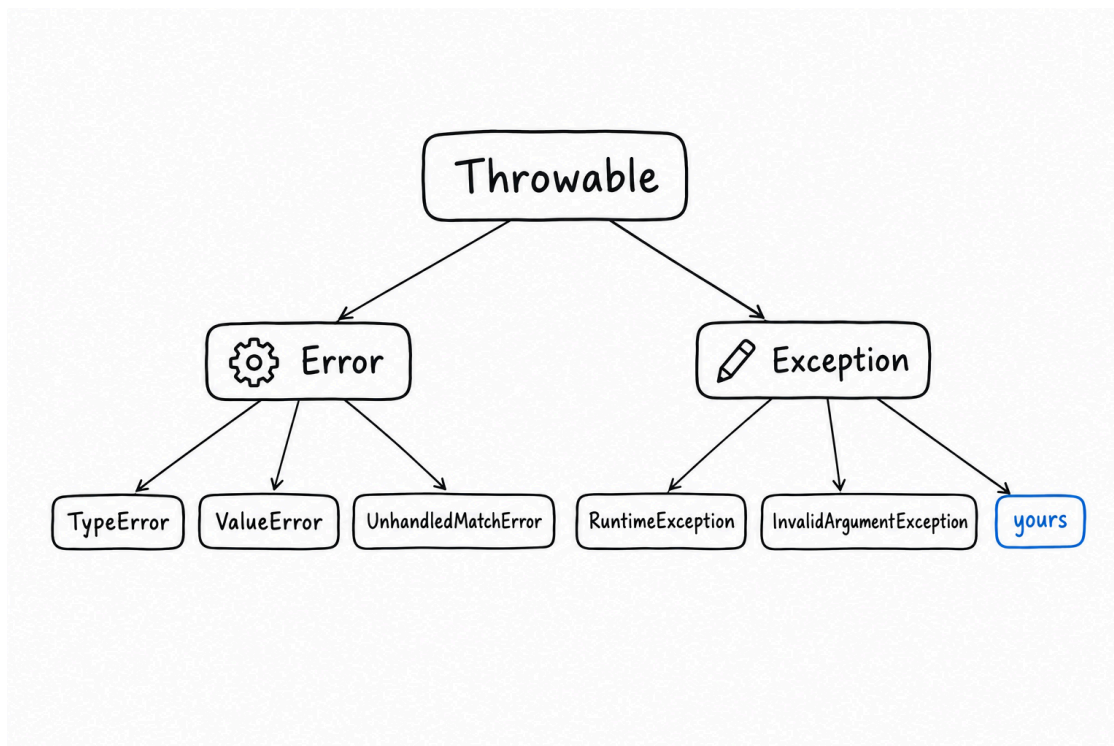
PHP has exceptions, and they work the way Python's or Java's do. It also has an older mechanism, **engine errors**, that predates exceptions and still exists. Modern practice is to route everything through the first one. This chapter shows the exception model, then the three lines of configuration that make the old mechanism behave.

Two hierarchies, one root

Everything you can `throw` and `catch` implements `Throwable`. Under it sit two families.

`Error` is what the engine throws when your code is wrong: `TypeError` for a bad argument, `ValueError` for a right type with an impossible value, `ArgumentCountError`, `DivisionByZeroError`, `UnhandledMatchError` when a `match` finds no arm. You do not throw these yourself, and you rarely catch them, because they mean a bug rather than a condition.

`Exception` is yours. PHP ships a small set in the SPL, and the names are the whole documentation: `InvalidArgumentException`, `RuntimeException`, `LogicException`, `DomainException`, `OutOfRangeException`, `UnexpectedValueException`. Extend one of these rather than `Exception` directly, and callers get a meaningful family to catch.



The syntax has no surprises:


```

<?php
declare(strict_types=1);

function parsePort(string $raw): int
{
    if (!ctype_digit($raw)) {
        throw new InvalidArgumentException("Not a port: $raw");
    }

    return (int) $raw;
}

try {
    $port = parsePort('80a');
} catch (InvalidArgumentException|ValueError $e) {
    echo 'Bad input: ', $e->getMessage(), PHP_EOL;
} finally {
    echo 'Done.', PHP_EOL;
}

```

| catches several types in one block (PHP 7.1). `finally` runs whether or not something was thrown. Catch `Throwable` when you truly mean everything, for instance at the top of a worker loop.

No checked exceptions, no error values

A PHP function signals failure by throwing or by returning `null`. Nothing forces the caller to handle either, and nothing in the signature lists what may be thrown. If you come from Java, there is no `throws` clause and no compile-time check; a `@throws` docblock is a courtesy read by your IDE and by PHPStan or Psalm, not by the engine. If you come from Go or Rust, there is no error return value and no `Result` type in the language. A handful of libraries offer one, but idiomatic PHP does not use them.

The nullable return type plus `??` is the idiom for “maybe”:

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

function findUser(int $id): ?array
{
    return $id === 1 ? ['name' => 'Ada'] : null;
}

$name = findUser(2)['name'] ?? 'anonymous';
echo $name, PHP_EOL; // anonymous
```

The rule of thumb: return `null` when absence is normal, throw when it is not. And since `throw` is an expression (PHP 8.0), the two combine in one line:

```
$user = findUser($id) ?? throw new RuntimeException("No user $id");
```

Custom exceptions carry data

A custom exception is a class, so it can hold typed context and offer a named constructor that builds the message for you:

```

<?php
declare(strict_types=1);

final class InsufficientFunds extends DomainException
{
    public function __construct(
        public readonly int $requested,
        public readonly int $available,
        ?Throwable $previous = null,
    ) {
        parent::__construct(
            "Requested $requested, only $available available",
            previous: $previous,
        );
    }

    public static function forWithdrawal(int $requested, int
$available): self
    {
        return new self($requested, $available);
    }
}

try {
    throw InsufficientFunds::forWithdrawal(100, 40);
} catch (InsufficientFunds $e) {
    echo $e->available, PHP_EOL; // 40
}

```

The `previous` argument is how you chain. **Catch a low-level exception, wrap it in one that means something to your caller, and pass the original as `previous`.** `getPrevious()` walks the chain back, and every logger prints it, so nothing is lost.

```

try {
    $pdo->query($sql);
} catch (PDOException $e) {
    throw new RuntimeException('Order lookup failed', previous: $e);
}

```

The other mechanism

Before exceptions existed, PHP reported problems by emitting an error of a given level (notice, warning, fatal) and, for anything short of fatal, carrying on. Most of that machinery has been folded into `Error` exceptions over the years: division by zero throws, a wrong argument type throws, calling a method on `null` throws. **A few conditions still emit a warning and continue:** reading an undefined variable, an undefined array key, an undefined property, and every deprecation notice.

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

$config = [];
echo $config['debug']; // Warning: Undefined array key "debug"
echo 'still running', PHP_EOL;
```

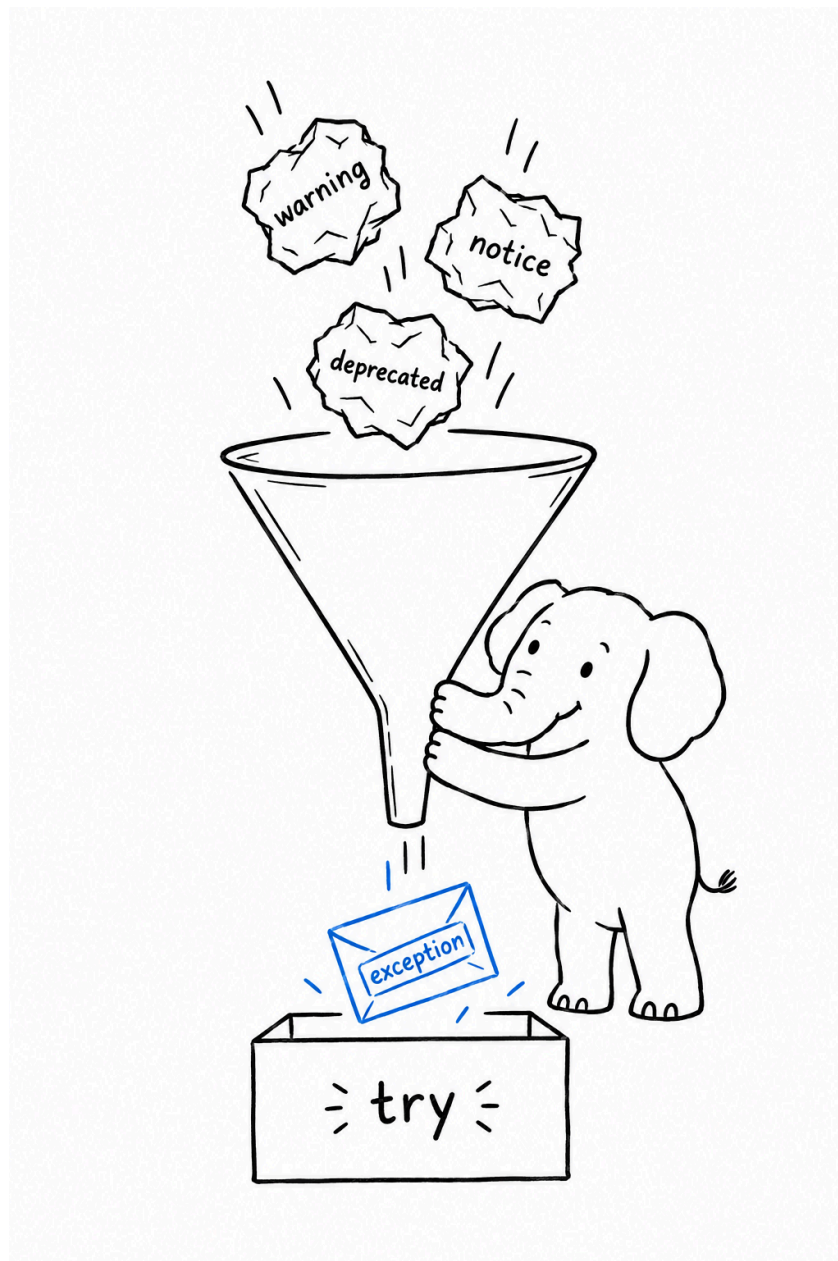
That “still running” is what a polyglot does not expect. The fix is one handler, installed at bootstrap, that turns every engine error into an exception. Every framework does exactly this:

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

error_reporting(E_ALL);

set_error_handler(function (int $severity, string $message, string
$file, int $line): bool {
    throw new \ErrorException($message, 0, $severity, $file, $line);
});

$config = [];
echo $config['debug']; // \ErrorException: Undefined array key "debug"
```



`ErrorException` is a built-in exception that remembers the severity. From that point on there is only one failure path, and `try` catches all of it.

Three settings complete the picture. `error_reporting(E_ALL)` makes sure nothing is filtered out. `display_errors` is `On` in development and `Off` in production, where errors go to the log instead: an uncaught exception on a public page must never print a stack trace. `set_exception_handler()` receives whatever reaches the top without being caught, which is where you log it and render a generic error page. PHP 8.5 adds `get_error_handler()` and `get_exception_handler()` so that libraries can inspect what is installed before wrapping it.

What cannot be caught

Fatal errors end the request: running out of memory, exceeding `max_execution_time`, declaring the same class twice. No `catch` sees them. (A parse error in an included file, on the other hand, is a

`ParseError` you can catch since PHP 7.) If you must react, `register_shutdown_function()` runs after the script stops, and `error_get_last()` tells you whether the stop was clean. Since PHP 8.5, a fatal error prints a backtrace, so an out-of-memory in production finally points at a line.

Two operators to recognise

You will meet `@` in older code: `@file_get_contents($url)`. **It silences any warning the expression emits.** Your error handler is still called, but `error_reporting()` returns a reduced mask inside it, which is how a handler can tell that `@` was used. Treat the operator as a smell. The one defensible use is around a function that warns and returns `false` on failure, immediately followed by a check on that return value. Even there, a `try` around an exception-throwing alternative reads better.

`assert()` is the other one. It is a development-time check, stripped from production when `zend.assertions` is set to `-1` in `php.ini`, which is the recommended production value. Use it for invariants that document intent, never for input validation.

The trap

Two mistakes, both silent. The first is catching `Exception` in a top-level handler and believing you caught everything. A `TypeError` is an `Error`, not an `Exception`, and it walks straight past that block. Catch `Throwable` at the boundary.

The second is the empty catch:

```
try {  
    $cache->delete($key);  
} catch (Throwable) {  
}
```

The variable can be omitted (PHP 8.0), which makes the block honest about ignoring the exception, and there are cases where ignoring is right. But an empty catch around anything that matters is how a bug hides for a year. Log it, at least.

Everything that goes wrong should reach you as one exception, in one place. PHP will do that, once you ask.

That handler, and every class you throw, live in files that PHP has to find. [Namespaces](#), [Composer](#), and [Autoloading](#) explains how it finds them.

Namespaces, Composer, and Autoloading

PHP has no module system. It has three smaller things, `require`, namespaces, and an autoload hook, and Composer assembles them into a package manager that behaves like npm, pip, Maven or Cargo. You will never write a `require` for one of your own classes again, but it helps to know what Composer does on your behalf.

The three primitives

`require 'file.php';` reads and runs a file, once per call. That was how code was shared in 2005, and it is still how the whole thing bootstraps: a single `require` of `vendor/autoload.php` at the top of your entry point.

A namespace is a prefix on a class name. `namespace App\Billing;` at the top of a file makes every class declared in it `App\Billing\Something`. Nothing else: no hierarchy, no visibility, no folder. `App\Billing` is not “inside” `App`; they are two strings that happen to share a prefix.

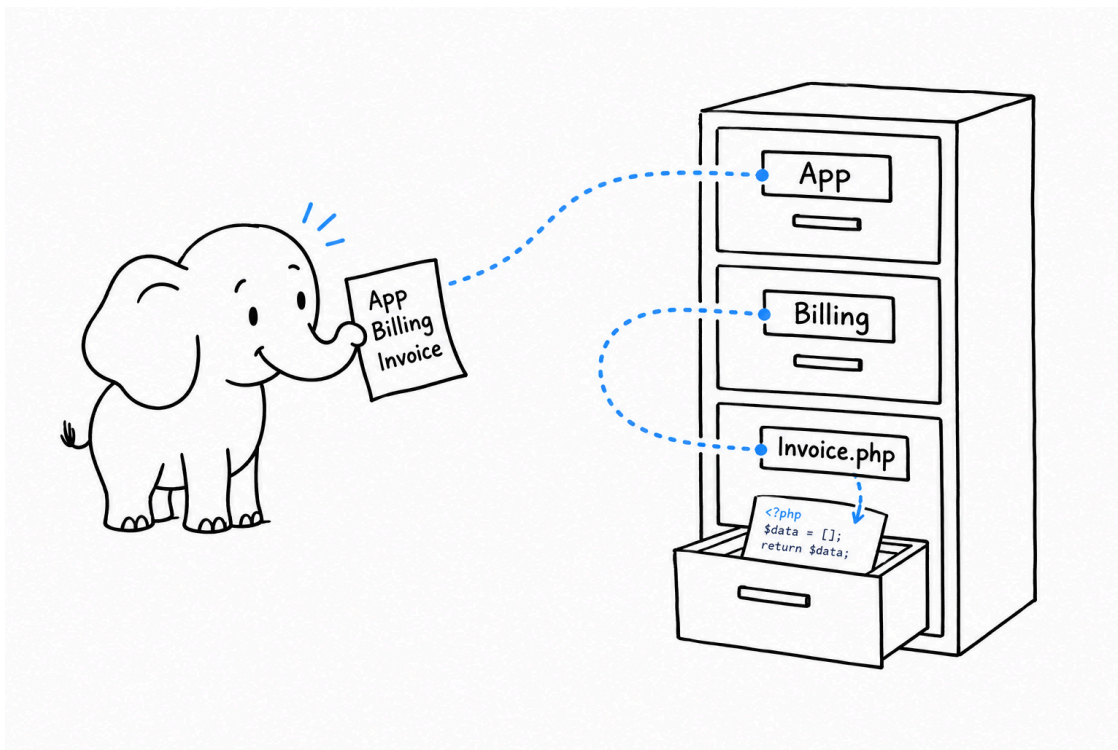
The autoload hook is the piece that makes the first two useful. **When PHP meets a class it has not seen, it calls a function you registered, passing the class name, and that function is expected to `require` the right file.** After that PHP retries.

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

spl_autoload_register(function (string $class): void {
    $file = __DIR__ . '/src/' . str_replace('\\', '/', $class) .
    '.php';
    if (is_file($file)) {
        require $file;
    }
});

$invoice = new App\Billing\Invoice(); // loads
src/App/Billing/Invoice.php
```

That is the entire mechanism. Composer writes a better version of that closure, with a cache, and hands it to you.



PSR-4: name to path

The rule the closure above follows has a name, PSR-4, and Composer implements it from a block in `composer.json`:

```
{
  "name": "acme/shop",
  "type": "project",
  "require": {
    "php": "^8.4",
    "ext-intl": "*"
  },
  "require-dev": {
    "phpunit/phpunit": "^12.0"
  },
  "autoload": {
    "psr-4": { "App\\": "src/" }
  },
  "autoload-dev": {
    "psr-4": { "App\\Tests\\": "tests/" }
  }
}
```


`App\Billing\Invoice` lives in `src/Billing/Invoice.php`. One class per file, file name equal to class name, folders equal to namespace segments after the prefix. The layout that results is the same on every modern PHP project you will open:

```
shop/
├─ composer.json
├─ composer.lock
├─ public/
│   └─ index.php          # require __DIR__ . '/../vendor/autoload.php';
├─ src/
│   └─ Billing/
│       └─ Invoice.php    # namespace App\Billing;
├─ tests/
│   └─ Billing/
│       └─ InvoiceTest.php
└─ vendor/                # generated, not committed
```

Add a class under `src/`, and it is found on the next request with no command to run. PSR-4 resolves by path at call time. The older `classmap` strategy scans folders into a generated array and needs `composer dump-autoload` after every new file; you will meet it in legacy projects.

Composer, in the vocabulary you know

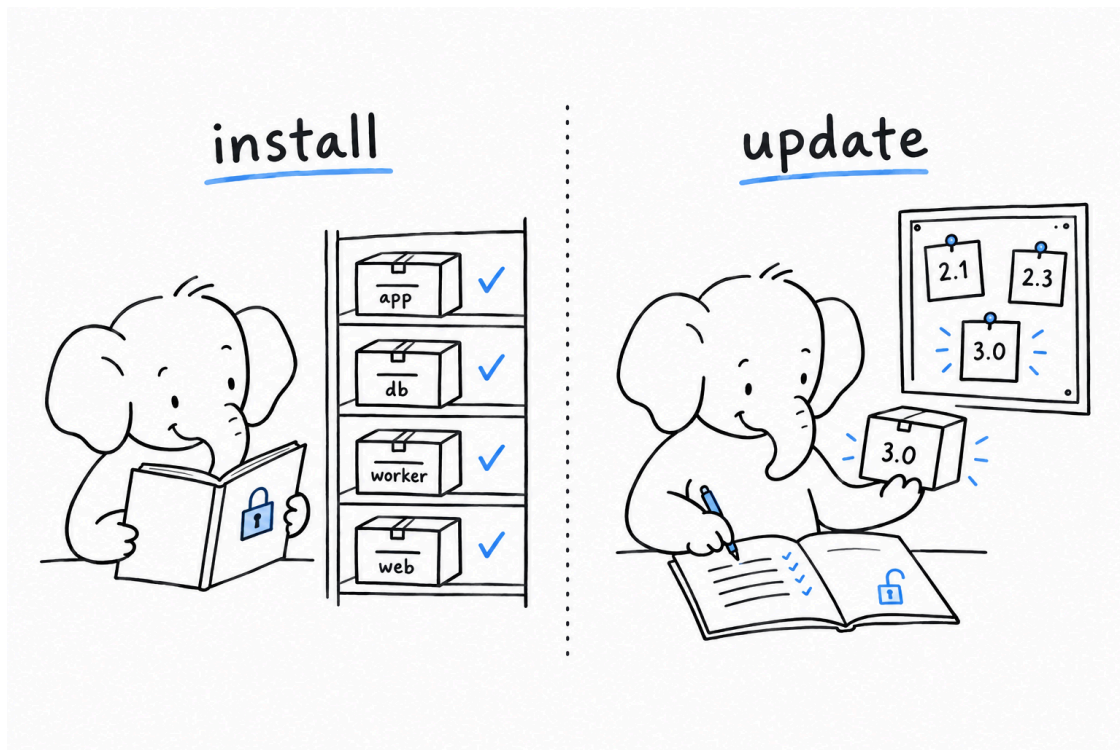
`composer.json` is your `package.json`. `composer.lock` is the lockfile. `vendor/` is `node_modules`, generated and ignored by Git. Packagist is the registry.

```
composer init                                # interactive composer.json
composer require monolog/monolog            # add and install, updates the
lock
composer require --dev phpstan/phpstan      # development-only dependency
composer install                             # reproduce exactly what the
lock says
composer update                             # resolve anew, rewrite the lock
composer update monolog/monolog             # ... for one package only
composer show                               # what is installed, with
versions
composer outdated                           # what has a newer release
composer audit                              # known vulnerabilities in the
lock
composer dump-autoload -o                   # regenerate the autoloader,
optimised
```

install obeys the lock; **update** rewrites it. In CI and in production you run `install`, and you get the exact versions your colleague tested. Commit the lock for an application. For a library, most authors do not, so that the library is tested against whatever its users resolve; the debate is the same one you have had in every other ecosystem.

Version constraints follow semver, and `^8.4` means “8.4 or any later 8.x”. The `php` entry in `require` is a constraint too; with `config.platform.php` you can pin the version Composer resolves against, so a developer on PHP 8.5 does not pull a package your production 8.4 cannot run.

Scripts live under a `scripts` key and run with `composer run name` or as lifecycle hooks (`post-install-cmd`), the way npm scripts do. Extensions are not packages: `ext-intl` in `require` only checks that the extension is present, and installing it is the job of your package manager, PECL, or PIE. Tools you would install globally elsewhere (linters, analysers) go in `require-dev` instead, so every developer and CI runs the same version; `composer global require` exists and is best left alone. A monorepo declares its inner packages as `path` repositories, and Composer symlinks them.



Living with namespaces

Inside a file, a `use` statement imports a name so you can write it short. Aliases resolve collisions, and functions and constants can be imported too:

```

<?php
declare(strict_types=1);

namespace App\Billing;

use App\Customer\Customer;
use DateTimeImmutable as Date;
use function App\Support\money;
use const App\Support\CURRENCY;

final class Invoice
{
    public function __construct(
        public readonly Customer $customer,
        public readonly Date $issuedOn,
    ) {
    }

    public function total(): string
    {
        return money(1999, CURRENCY);
    }
}

echo Invoice::class, PHP_EOL; // App\Billing\Invoice

```

`Invoice::class` gives the fully qualified name as a string, which is what you pass to a container, a mock, or `instanceof` checks that take a string. A leading backslash, `\DateTimeImmutable`, names a class from the global namespace without a `use`.

Functions fall back to the global namespace; classes do not. Calling `strlen()` inside `App\Billing` looks for `App\Billing\strlen` first, then `strlen`. Calling `new DateTimeImmutable()` without a `use` or a leading backslash fails. You will see `\strlen()` in some libraries: the backslash skips the lookup and lets OPcache inline a few built-ins. It is a micro-optimisation, not a convention you need to adopt.

The standards worth knowing

The PHP-FIG is the group where framework and library authors agree on interfaces, published as PSRs. **A PSR is a contract, not a library**: implementations come from many vendors, and you can swap one for another because your code only sees the interface. The ones you will meet in the first week:

- **PSR-4** autoloading, above.
- **PER Coding Style**, the successor of PSR-12: brace placement, indentation, naming. PHP-CS-Fixer or PHP_CodeSniffer enforces it.
- **PSR-3** `LoggerInterface`. Every library logs through it; you plug in the logger you like.
- **PSR-7**, **PSR-15** and **PSR-17**: HTTP request and response objects, middleware, and their factories. The language's own `$_GET` and `header()` are covered in [A Web Request, Without a Framework](#); PSR-7 is the object model libraries share on top.
- **PSR-11** `ContainerInterface`, so a library can ask any container for a service.
- **PSR-14** event dispatching, **PSR-6** and **PSR-16** caching.

A library that types its constructor against `Psr\Log\LoggerInterface` works in every framework listed in this book. That interoperability is why the ecosystem has fewer forks and rewrites than its size would suggest.

The trap

Never edit a file in `vendor/`. The next `composer install` on any machine erases the change, and nobody will know why production differs from your laptop. Fork the package, or override the class through your own autoload entry, or send the patch upstream.

The second trap is quieter. You add a class, PHP says it does not exist, and you spend twenty minutes on a typo that is not there. Check the namespace against the folder: `App\Billing\Invoice` must be `src/Billing/Invoice.php`, letter for letter, case included on Linux. If the project uses `classmap` rather than `psr-4`, run `composer dump-autoload` and move on.

Composer is not a build step. There is nothing to compile, bundle or transpile.
`composer install`, then run the code.

With classes found and packages installed, what remains is the standard library you will lean on every day. [Strings](#), [Numbers](#), [Dates](#), and [JSON](#) is the tour.

Strings, Numbers, Dates, and JSON

PHP ships with a large standard library, and most of it is functions, not methods. You do not call `$s.upper()`. You call `strtoupper($s)`. There is no string class, no number class, no receiver: values go in as arguments, results come out as return values. Once you accept that, the library is wide, fast, and documented function by function on php.net, where each page has the signature, the changelog by version, and examples worth reading.

The names are inconsistent. `strlen` sits next to `str_replace`, `array_key_exists` takes the key first and `in_array` takes the needle first, `strpos` returns `false` on failure, and so does `array_search`, but `preg_match` returns 0. This is the residue of thirty years of growth, and it is not going away. Your editor's autocomplete and the php.net page are the fix, not memorisation. After a week you stop noticing.

Strings are bytes

A PHP string is a sequence of bytes. Not code points, not grapheme clusters, bytes. `strlen('é')` is 2. `strrev('héllo')` scrambles the accent. `substr` can cut a multibyte character in half.

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

$word = 'café';

echo strlen($word), PHP_EOL;      // 5
echo mb_strlen($word), PHP_EOL;   // 4
echo strtoupper($word), PHP_EOL;  // CAFÉ
echo mb_strtoupper($word), PHP_EOL; // CAFÉ
```

For any text a human will read, use the **mb_ family**: `mb_strlen`, `mb_substr`, `mb_strtoupper`, `mb_str_pad`, and since PHP 8.4 `mb_trim`, `mb_ucfirst` and `mb_lcfirst`. They assume UTF-8 by default. The plain functions stay useful for what they were made for: binary data, ASCII protocols, hashes, and anything where a byte really is the unit.

For anything locale-aware, the `intl` extension wraps ICU: `Normalizer` to canonicalise composed and decomposed forms, `Collator` to sort “é” next to “e” in the user’s language, `NumberFormatter` for currencies and plurals, `IntlDateFormatter` for dates spelled out the way a French or Japanese reader expects.

The everyday helpers are what you expect from any language, with PHP names:

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

$path = '/var/log/app.log';

var_dump(str_contains($path, 'log'));      // true (PHP 8.0)
var_dump(str_starts_with($path, '/var'));  // true
var_dump(str_ends_with($path, '.log'));    // true

echo implode(', ', ['a', 'b', 'c']), PHP_EOL;    // a, b, c
print_r(explode('/', trim($path, '/')));         // ['var', 'log',
'app.log']
echo str_pad('7', 3, '0', STR_PAD_LEFT), PHP_EOL; // 007
echo ucfirst('php'), PHP_EOL;                   // Php
echo sprintf('%05.2f|%-6s|%03d', 3.14159, 'ok', 7), PHP_EOL; //
03.14|ok    |007
echo number_format(1234567.891, 2), PHP_EOL;     // 1,234,567.89
```

`sprintf` is C's, `printf` prints instead of returning, and `number_format` is what you reach for before you discover `NumberFormatter`.

Two syntaxes handle multi-line text. **Heredoc interpolates**, **nowdoc does not**, and both strip the indentation of the closing marker:

```

<?php
declare(strict_types=1);

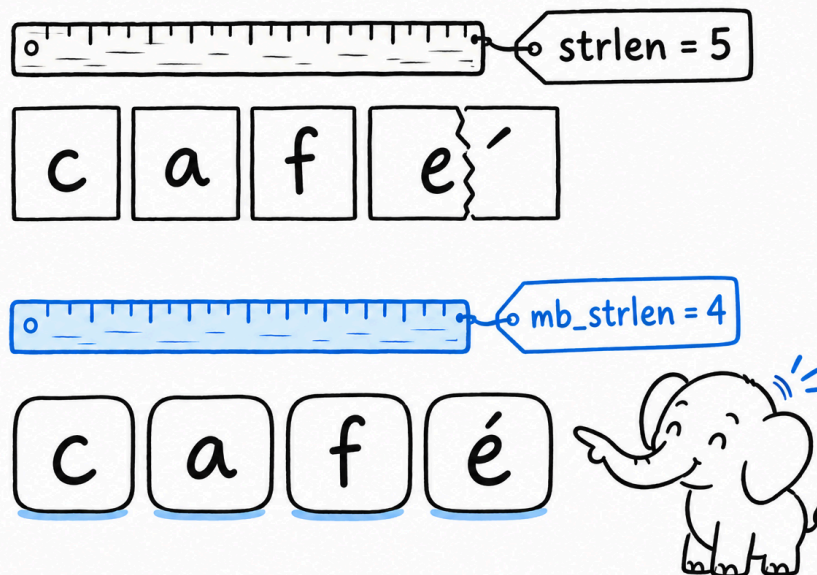
$name = 'Ada';

$greeting = <<<TXT
    Hello, {$name}.
    Welcome back.
    TXT;

$raw = <<<'TXT'
    Hello, {$name}. This stays literal.
    TXT;

echo $greeting, PHP_EOL, $raw, PHP_EOL;

```



Regular expressions

PHP uses PCRE, the same dialect as Perl and close to what Python's `re` and JavaScript accept. The pattern is a string with delimiters, usually `/` or `~`, followed by modifiers. **Add the `u` modifier whenever the text is UTF-8**, or `.` matches single bytes and `\w` ignores accented letters.


```

<?php
declare(strict_types=1);

$line = 'Order #4521 shipped to Zoé on 2026-03-14';

if (preg_match('/#(?<id>\d+).*on (?<date>\d{4}-\d{2}-\d{2})/u', $line,
$m)) {
    echo $m['id'], ' ', $m['date'], PHP_EOL; // 4521 2026-03-14
}

echo preg_replace('/\s+/u', ' ', "too  many\n\nspaces"), PHP_EOL;

echo preg_replace_callback(
    '/\d+/',
    fn (array $m): string => (string) ($m[0] * 2),
    'a1 b22 c333',
), PHP_EOL; // a2 b44 c666

```

`preg_match` returns 1, 0, or `false` on a malformed pattern. Named groups land in the match array under their name. `preg_split`, `preg_quote` and `preg_match_all` round out the family.

Numbers

`int` is a signed 64-bit integer, and when it overflows PHP silently switches to `float`. No exception, no wrap-around, a float. That is fine for a counter and wrong for an ID or a money amount.

```

<?php
declare(strict_types=1);

var_dump(PHP_INT_MAX + 1);    // float(9.223372036854776E+18)
var_dump(intdiv(7, 2));      // int(3)
var_dump(7 / 2);             // float(3.5), division always yields
float unless exact
var_dump(7 % 2);             // int(1)
var_dump(2 ** 10);           // int(1024)
var_dump(fdiv(1, 0));        // float(INF), where 1 / 0 throws
DivisionByZeroError

var_dump(0.1 + 0.2 === 0.3); // false
var_dump(abs(0.1 + 0.2 - 0.3) < PHP_FLOAT_EPSILON); // true
var_dump(round(2.5), round(3.5), round(-2.5));      // 3, 4, -3:
half away from zero

```

Floats are IEEE 754 doubles with the usual caveats. `round` defaults to rounding half away from zero and takes a mode constant for bankers' rounding. **Money is either integers in the smallest unit (cents) or arbitrary precision:** `bccadd`, `bcmul` and friends work on strings, and PHP 8.4 wraps them in `BcMath\Number`, an immutable object with operator support.

```

// PHP 8.4
$price = new BcMath\Number('19.99');
$total = $price * 3;
echo $total, PHP_EOL; // 59.97

```

For randomness, `rand()` and `mt_rand()` are fast and predictable. **Anything security-related uses `random_int`, `random_bytes`, or the `Random\Randomizer` class** (PHP 8.2), which are backed by the operating system's CSPRNG.

```

<?php
declare(strict_types=1);

echo random_int(1, 6), PHP_EOL;           // a fair die
echo bin2hex(random_bytes(16)), PHP_EOL;  // 32 hex chars, a fine
token

$r = new Random\Randomizer();
echo $r->getInt(1, 100), PHP_EOL;
print_r($r->shuffleArray([1, 2, 3, 4]));
echo $r->getBytesFromString('abcdef0123456789', 8), PHP_EOL; // PHP 8.3

```

Dates

Use **`DateTimeImmutable`** and never **`DateTime`**. Both exist, they share an interface, and the mutable one is a trap: `$date->modify('+1 day')` changes `$date` in place and returns it, so every reference to that object shifts with it. The immutable version returns a new object and leaves the original alone, the way you would expect from Java's `java.time` or Python's `datetime`.

```

<?php
declare(strict_types=1);

date_default_timezone_set('UTC');

$start = new DateTimeImmutable('2026-03-14 09:30', new
DateTimeZone('Europe/Paris'));
$end = $start->modify('+2 weeks')->setTime(18, 0);

echo $start->format(DateTimeInterface::ATOM), PHP_EOL; // 2026-03-
14T09:30:00+01:00
echo $end->format('D, d M Y H:i'), PHP_EOL;           // Sat, 28 Mar
2026 18:00

$diff = $start->diff($end);
echo $diff->days, ' days, ', $diff->h, ' hours', PHP_EOL; // 14 days, 8
hours

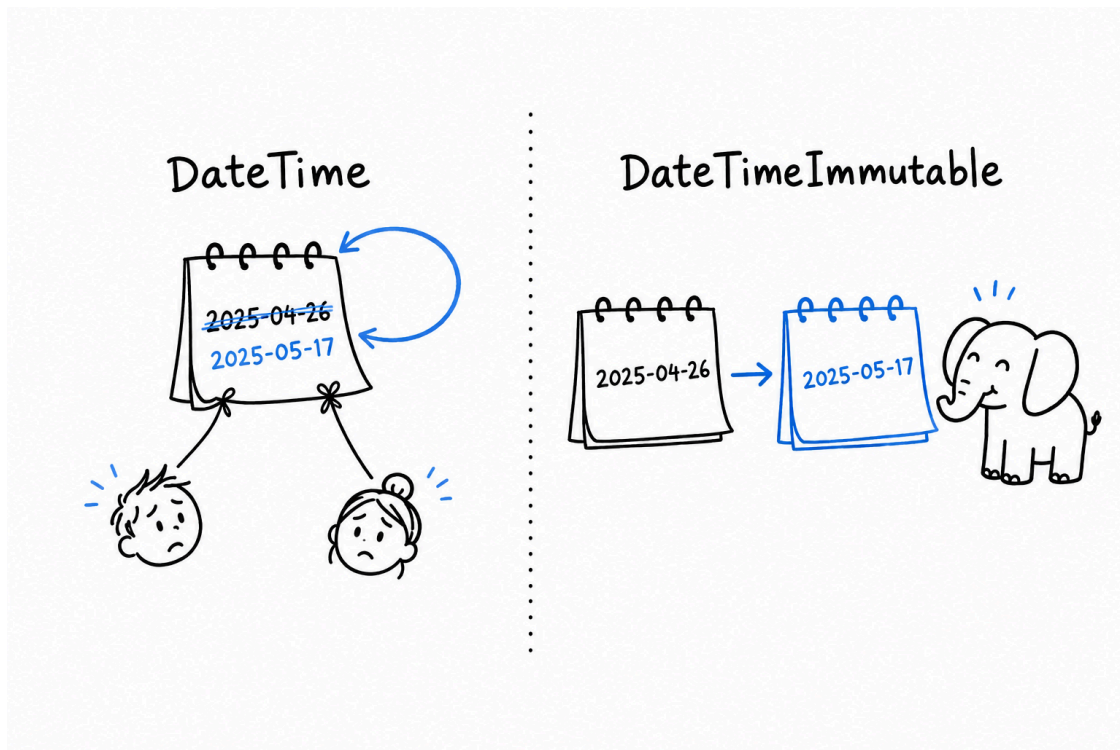
$parsed = DateTimeImmutable::createFromFormat('d/m/Y', '01/07/2026');
echo $parsed->getTimestamp(), PHP_EOL;

$inTokyo = $start->setTimezone(new DateTimeZone('Asia/Tokyo'));
echo $inTokyo->format('H:i T'), PHP_EOL; // 17:30 JST

foreach (new DatePeriod($start, new DateInterval('P1D'), 3) as $day) {
    echo $day->format('l'), PHP_EOL; // Saturday, Sunday, Monday,
Tuesday
}

```

The parser behind `new DateTimeImmutable(...)` and `modify()` accepts English phrases ('next monday', 'first day of last month') and every common ISO layout. `DateInterval` uses ISO 8601 durations (P1Y2M3DT4H). The default timezone comes from `php.ini`; setting it to UTC at the top of your entry point and converting at the edges is the usual practice. Format codes are PHP's own (Y-m-d H:i:s), not `strftime`'s, which was deprecated in 8.1.



JSON

`json_encode` and `json_decode` are built in and fast. Pass `JSON_THROW_ON_ERROR` every time, otherwise a failure returns `false` or `null` and you find out three functions later.

```

<?php
declare(strict_types=1);

$payload = ['id' => 4521, 'tags' => ['php', 'json'], 'total' => 59.97,
'note' => null];

$json = json_encode($payload, JSON_THROW_ON_ERROR | JSON_PRETTY_PRINT |
JSON_UNESCAPED_UNICODE);
echo $json, PHP_EOL;

$array = json_decode($json, true, flags: JSON_THROW_ON_ERROR); //
    nested arrays
$obj = json_decode($json, flags: JSON_THROW_ON_ERROR); //
    stdClass objects
echo $array['tags'][0], ' ', $obj->tags[1], PHP_EOL; //
    php json

var_dump(json_validate('{"ok": true}')); // true (PHP 8.3), without
    building the tree

```

The second argument of `json_decode` decides between associative arrays and `stdClass`. Arrays are what most code wants. Objects of your own classes serialise through the `JsonSerializable` interface: implement `jsonSerialize(): mixed` and return the array shape you want. The reverse direction, JSON to typed object, is not in the language; libraries and frameworks provide it.

One pitfall deserves its own sentence, and [Arrays](#) explains why: **an array whose keys are not 0, 1, 2... encodes as a JSON object, not a list.** `array_filter` leaves holes, `json_encode` sees holes, your API returns `{"0": ..., "2": ...}`. Wrap it in `array_values()` first.

Large integers survive a round trip only while they fit in 64 bits; beyond that, decode with `JSON_BIGINT_AS_STRING`. Floats print with up to 17 significant digits by default (`serialize_precision`), so `0.1` stays `0.1`.

Files and streams

`file_get_contents` and `file_put_contents` read or write a whole file in one call, and they accept URLs and stream wrappers as well as paths. For line-by-line work there is the C-style `fopen` / `fgets` / `fclose` trio, or `SplFileObject`, which is iterable:

```

<?php
declare(strict_types=1);

$path = __DIR__ . '/notes.txt';
file_put_contents($path, "one\ntwo\nthree\n");

foreach (new SplFileObject($path) as $n => $line) {
    if ($line !== '') {
        echo $n, ': ', rtrim($line), PHP_EOL;
    }
}

$stdin = fopen('php://stdin', 'r');
$buffer = fopen('php://memory', 'r+');
fwrite($buffer, 'scratch');
rewind($buffer);
echo fread($buffer, 100), PHP_EOL;

```

`__DIR__` is the folder of the current file, and relative paths otherwise resolve against the working directory, which under a web server is rarely what you think. The `php://` wrappers expose standard streams, memory buffers and temp files through the same functions as real files, and `file_get_contents('https://...')` works when `allow_url_fopen` is on, though for real HTTP work you want `curl` or a PSR-18 client.

Validation, hashing, URIs

`filter_var` validates and sanitises scalars against a set of built-in filters, with no dependency:

```

<?php
declare(strict_types=1);

var_dump(filter_var('ada@example.org', FILTER_VALIDATE_EMAIL)); // the
string, or false
var_dump(filter_var('42', FILTER_VALIDATE_INT));                //
int(42)
var_dump(filter_var('yes', FILTER_VALIDATE_BOOL,
FILTER_NULL_ON_FAILURE)); // true

$hash = password_hash('correct horse battery staple',
PASSWORD_DEFAULT);
var_dump(password_verify('correct horse battery staple', $hash)); //
true
echo hash('sha256', 'payload'), PHP_EOL;

```

`password_hash` picks the algorithm, generates the salt, and encodes everything into one string; `password_verify` reads it back. That is the whole password story, and `PASSWORD_DEFAULT` moves to stronger algorithms across versions without you changing code. `hash` covers everything else, from `sha256` to `xxh3`, and `hash_hmac` signs.

Since PHP 8.5, URLs are parsed by a real parser rather than the loose `parse_url`:

```

// PHP 8.5
$uri = new Uri\Rfc3986\Uri('https://example.org:8443/docs/intro?
lang=fr#top');
echo $uri->getHost(), ' ', $uri->getPort(), ' ', $uri->getPath(),
PHP_EOL;
// example.org 8443 /docs/intro

```

`Uri\WhatWg\Uri` in the same extension applies the browser rules instead of the RFC ones, for when you need to agree with what an `<a href>` will do.

That is the standard library you will touch in a normal week. The next question is what PHP gives you when the input is not a string in a variable but an HTTP request. [A Web Request, Without a Framework](#) answers it with no library at all.

A Web Request, Without a Framework

PHP can serve a web page with no library, no server code, and no configuration, because handling an HTTP request is what the language was built for. The request is already parsed when your script starts. The response is whatever you print. A framework adds structure on top of that; it does not add the capability.

Seeing the raw layer once makes every framework legible, because they all sit on exactly these primitives.

The front controller

Point PHP's development server at a single file and every URL goes through it:

```
php -S localhost:8000 public/index.php
```

That file is the front controller. In production the web server does the same thing with a rewrite rule (or FrankenPHP and RoadRunner do it for you, as [How PHP Runs](#) describes). With the development server, one detail matters: **if the script returns `false`, the server serves the requested file from disk instead**, which is how static assets get through.

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

$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

if ($path !== '/' && is_file(__DIR__ . $path)) {
    return false; // let the built-in server send the CSS or image
}

echo 'Every other URL lands here: ', htmlspecialchars($path,
ENT_QUOTES);
```

Reading the request

The request lives in superglobals, arrays that PHP fills before the first line of your code runs.

`$_GET` holds the query string, `$_POST` the fields of a submitted form, `$_COOKIE` the cookies, `$_FILES` the uploads, and `$_SERVER` everything else: `REQUEST_METHOD`, `REQUEST_URI`, and each HTTP header as `HTTP_` plus its upper-cased name, so `Accept-Language` becomes `$_SERVER['HTTP_ACCEPT_LANGUAGE']`.

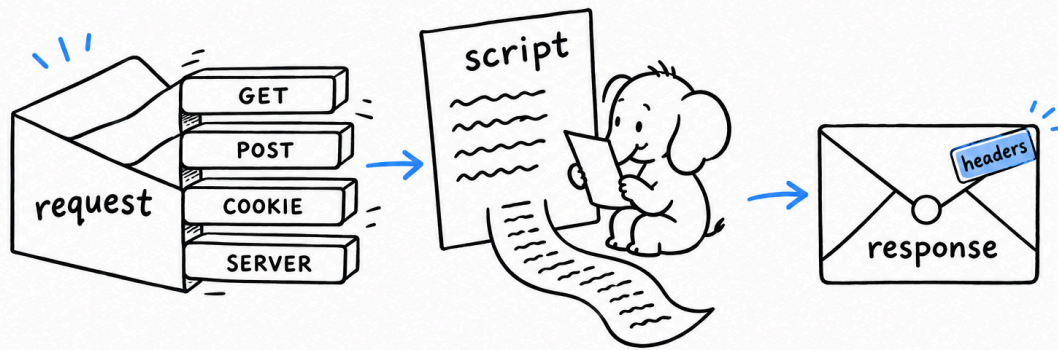
```
<?php
declare(strict_types=1);

$method = $_SERVER['REQUEST_METHOD'];
$page = filter_input(INPUT_GET, 'page', FILTER_VALIDATE_INT) ?: 1;
$name = trim($_POST['name'] ?? '');
$lang = $_SERVER['HTTP_ACCEPT_LANGUAGE'] ?? 'en';

$body = json_decode(file_get_contents('php://input'), true, flags:
JSON_THROW_ON_ERROR);
```

`$_POST` is only filled for `application/x-www-form-urlencoded` and `multipart/form-data` bodies on POST. A JSON body, whatever the method, is read raw from `php://input`. A form sent with PUT or PATCH is not parsed at all unless you ask: `request_parse_body()` (PHP 8.4) returns the fields and files for those methods too.

Nothing in these arrays is trustworthy. `HTTP_HOST` is whatever the client sent. `REQUEST_URI` can contain anything. A field you expected as a string arrives as an array if the client writes `name[]=x`. Treat every value as untyped user input, validate it with `filter_var` or your own checks, and only then let it near your logic.



Writing the response

Whatever your script outputs is the **response body**. `echo`, `print`, and any text outside `<?php ?>` tags go to the client. Status and headers are set with two functions, and they must be called before the first byte of output, because the headers travel first:

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

http_response_code(201);
header('Content-Type: application/json; charset=utf-8');
header('Cache-Control: no-store');
setcookie('theme', 'dark', [
    'expires' => time() + 86400 * 30,
    'path' => '/',
    'secure' => true,
    'httponly' => true,
    'samesite' => 'Lax',
]);

echo json_encode(['created' => true], JSON_THROW_ON_ERROR);
```

Print something first, even a stray newline before `<?php`, and `header()` fails with “headers already sent”. Output buffering (`ob_start()` at the top, `ob_end_flush()` at the end) holds the body in memory until the script finishes and makes the order irrelevant, which is what frameworks do.

Templates are PHP

PHP started as a templating language, and it still is one. An HTML file with `<?= $expr ?>` in it is a template; `include` it and it prints. The one rule is on output: **escape every value with `htmlspecialchars` before it lands in HTML**, or the first user named `<script>` owns your page.

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

function e(string $value): string
{
    return htmlspecialchars($value, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
}

$todos = ['Write the chapter', 'Escape <everything>'];
?>
<ul>
<?php foreach ($todos as $todo): ?>
    <li><?= e($todo) ?></li>
<?php endforeach; ?>
</ul>
```

The `foreach (...): ... endforeach;` form exists for exactly this interleaving. A two-line `e()` helper is the whole escaping story for HTML; attributes, URLs and JavaScript contexts each need their own encoding, which is the part template engines automate.

A complete application

Here is a working application in one file: a list of notes, stored in SQLite, with a form to add one. It runs with `php -S localhost:8000 index.php` and nothing else.

```

<?php
declare(strict_types=1);

$db = new PDO('sqlite:' . __DIR__ . '/notes.db', options: [
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
]);
$db->exec('CREATE TABLE IF NOT EXISTS notes (id INTEGER PRIMARY KEY,
body TEXT NOT NULL)');

function e(string $value): string
{
    return htmlspecialchars($value, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-
8');
}

$route = $_SERVER['REQUEST_METHOD'] . ' ' .
parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

match ($route) {
    'GET /' => (function () use ($db): void {
        $notes = $db->query('SELECT id, body FROM notes ORDER BY id
DESC')->fetchAll();
        echo '<h1>Notes</h1><form method="post" action="/notes">',
            '<input name="body" required> <button>Add</button></form>
<ul>';
        foreach ($notes as $note) {
            echo '<li>', e($note['body']), '</li>';
        }
        echo '</ul>';
    })(),
    'POST /notes' => (function () use ($db): void {
        $body = trim($_POST['body'] ?? '');
        if ($body === '') {
            http_response_code(422);
            echo 'A note needs a body.';
            return;
        }
        $stmt = $db->prepare('INSERT INTO notes (body) VALUES
(:body)');

```

```

        $stmt->execute(['body' => $body]);
        http_response_code(303);
        header('Location: /');
    })(),
    default => (function (): void {
        http_response_code(404);
        echo 'Not found';
    })(),
};

```

Three things to take from it. **The router is a match on method and path**, which scales to about ten routes before you want a real one. **PDO is the database API**, one interface for SQLite, MySQL, PostgreSQL and others, and **ERRMODE_EXCEPTION** turns every failure into a thrown `PDOException` instead of a `false` you forget to check. **The query uses a named placeholder and `execute()` binds the value**; the SQL text and the data never meet as a string, so there is no injection to worry about. PHP 8.4 adds driver-specific subclasses (`Pdo\Sqlite`, `Pdo\MySQL`, `Pdo\Pgsql`) through `Pdo::connect()`, exposing each driver's extras with proper types.

Building the SQL with interpolation, `"WHERE id = $id"`, is the one habit from old PHP tutorials that the language still lets you keep. Do not.

Sessions and passwords

A session is server-side storage keyed by a cookie. Call `session_start()` before any output, and `$_SESSION` becomes an array that survives across requests for that visitor. By default the data lives in files on the server; frameworks swap in a database or a cache store through `session_set_save_handler()`. Nothing but the session id travels in the cookie.

```

<?php
declare(strict_types=1);

session_start();

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $ok = password_verify($_POST['password'] ?? '', $storedHash ?? '');
    if ($ok) {
        session_regenerate_id(true);
        $_SESSION['user_id'] = 42;
    }
}

$csrf = $_SESSION['csrf'] ??= bin2hex(random_bytes(32));

```

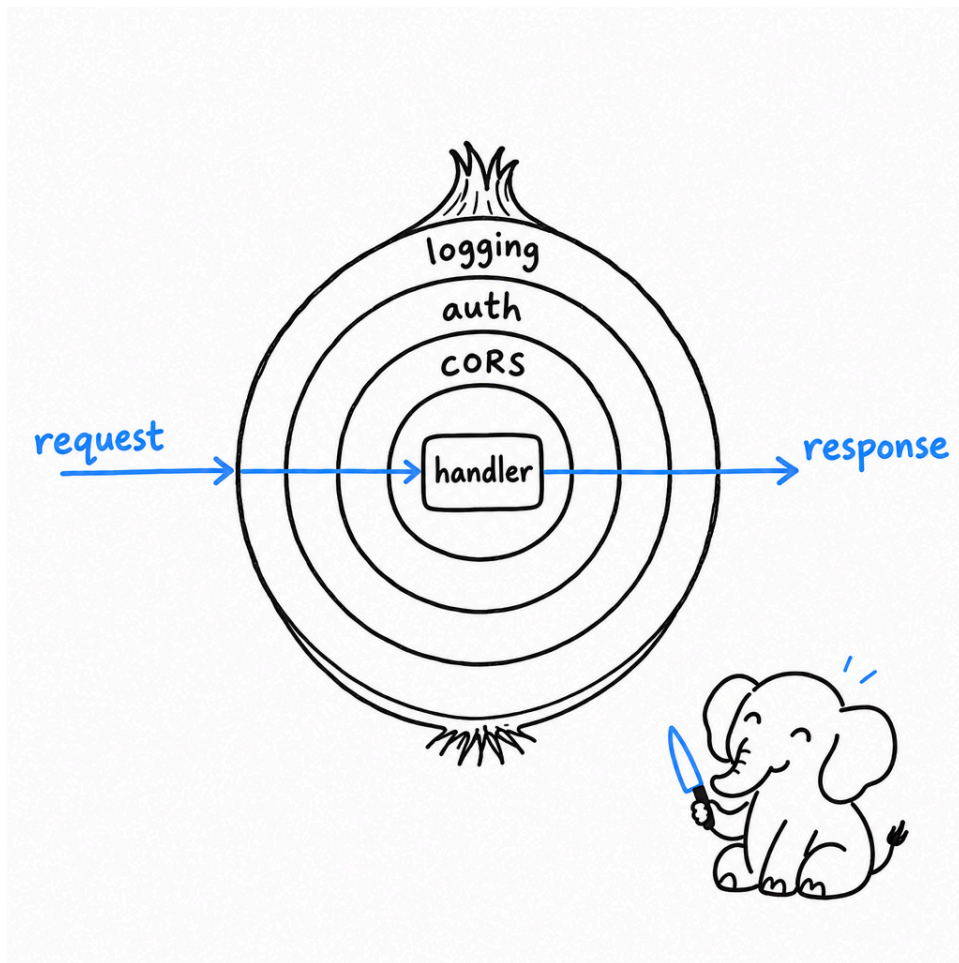
`password_hash` and `password_verify` from [Strings, Numbers, Dates, and JSON](#) are the entire password story. Regenerate the session id on login. Put a random token in the session, print it as a hidden field in every form, and compare on submit with `hash_equals()`: that is CSRF protection in four lines, and every framework does the same under a nicer name.

The standards layer

Superglobals and `header()` work, but they are global state, which makes code hard to test and impossible to compose. The PHP-FIG answered with interfaces:

- **PSR-7** defines immutable `RequestInterface` and `ResponseInterface` objects, so a request is a value you pass around and a response is a value you return.
- **PSR-15** defines middleware: a handler takes a request and returns a response, and middleware wraps handlers. Authentication, CORS, logging, rate limiting are each one class.
- **PSR-17** defines the factories that create those objects, so a library never depends on a specific implementation.
- **PSR-18** defines an HTTP client, the outgoing side of the same objects.

A library written against PSR-7 and PSR-15 runs in any framework that speaks them, which today is most of them. CakePHP, Laminas, Laravel, Symfony and Yii, and the micro-frameworks Mezzio and Slim, each add routing, dependency injection, templating and a database layer on top of these primitives; the primitives underneath are the ones you have just seen.



In production, the front controller stays the same. What changes is who calls it: PHP-FPM behind nginx, Apache or Caddy, or a long-running runtime such as FrankenPHP or RoadRunner. Nothing in this chapter needs to change for either.

The application above has no tests and no static analysis. [Tests, Static Analysis, and Tooling](#) fixes that.

Tests, Static Analysis, and Tooling

A maintained PHP project runs four tools on every commit: a test runner, a static analyser, a code style fixer, and Composer's own audit. None of them ship with the language. All of them install with one `composer require --dev` and run from `vendor/bin/`. If you have used pytest with mypy, or Jest with tsc and Prettier, you already know the shape. Only the names change.

This chapter names two tools for each job. That is not indecision. Both are widely used, both are good, and the project you just joined has already picked one.

Tests

Two test runners dominate. **PHPUnit** is the xUnit-style runner every other PHP test tool builds on. **Pest** is a describe-and-it layer on top of PHPUnit's engine. They share assertions, mocks, configuration and the coverage machinery; they differ in how a test reads.

The function under test:

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

namespace App;

function slugify(string $title): string
{
    $slug = strtolower(trim($title));
    $slug = preg_replace('/[^a-z0-9]+/', '-', $slug);

    return trim($slug, '-');
}
```

A namespaced function is not a class, so PSR-4 cannot find it: the file goes in a `files` entry of the `autoload` block in `composer.json`, and Composer requires it on every run.

The same test, PHPUnit first:

```

<?php
declare(strict_types=1);

namespace App\Tests;

use PHPUnit\Framework\Attributes\DataProvider;
use PHPUnit\Framework\Attributes\Test;
use PHPUnit\Framework\TestCase;
use function App\slugify;

final class SlugifyTest extends TestCase
{
    #[Test]
    public function itLowercasesAndJoinsWithDashes(): void
    {
        self::assertSame('hello-world', slugify('Hello World'));
    }

    #[Test]
    #[DataProvider('edgeCases')]
    public function itHandlesEdgeCases(string $input, string
$expected): void
    {
        self::assertSame($expected, slugify($input));
    }

    public static function edgeCases(): iterable
    {
        yield 'leading punctuation' => ['!!Hi', 'hi'];
        yield 'empty' => ['', ''];
        yield 'unicode is stripped' => ['café', 'caf'];
    }
}

```

Then Pest:

```

<?php
declare(strict_types=1);

use function App\slugify;

it('lowercases and joins with dashes', function () {
    expect(slugify('Hello World'))->toBe('hello-world');
});

it('handles edge cases', function (string $input, string $expected) {
    expect(slugify($input))->toBe($expected);
})->with([
    'leading punctuation' => ['!!Hi', 'hi'],
    'empty' => ['', ''],
    'unicode is stripped' => ['café', 'caf'],
]);

```

Run them with `vendor/bin/phpunit` or `vendor/bin/pest`. Configuration lives in `phpunit.xml` at the project root: which folders hold tests, whether to fail on warnings, which environment variables to set. Pest reads the same file.

The PHPUnit conventions worth knowing on day one: a test class ends in `Test` and extends `TestCase`, a test method carries the `#[Test]` attribute or starts with `test`, `setUp()` runs before each test, `self::assertSame()` is the strict assertion (there is an `assertEquals()`, and it juggles types the way `==` does, so prefer `assertSame()`), and `$this->createMock(SomeInterface::class)` returns a test double you configure with `->method('name')->willReturn($value)`.

Coverage is not built in. It needs an extension that watches which lines execute: Xdebug (with `xdebug.mode=coverage`) or PCOV, which does only coverage and does it faster. Either way, `vendor/bin/phpunit --coverage-text` prints the report.

The unicode test above documents a bug: `slugify('café')` drops the `é` instead of transliterating it. A test that pins down current behaviour is still a test. Fix the function later with `intl`'s `Transliterator`.

Static analysis

PHP checks types at runtime, one call at a time. It will not tell you that a function three files away can return `null` and you never handle it. **PHPStan** and **Psalm** are the compiler PHP does not have.

They read the whole codebase, follow every type through every call, and report what would fail before anything runs. Think mypy for Python, or the type checker inside TypeScript's `tsc`.

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

function findUser(int $id): ?User
{
    return $id === 1 ? new User('Ada') : null;
}

echo findUser(2)->name;
```

PHP runs this and crashes on the second line with “Attempt to read property on null”. Both analysers refuse it before that:

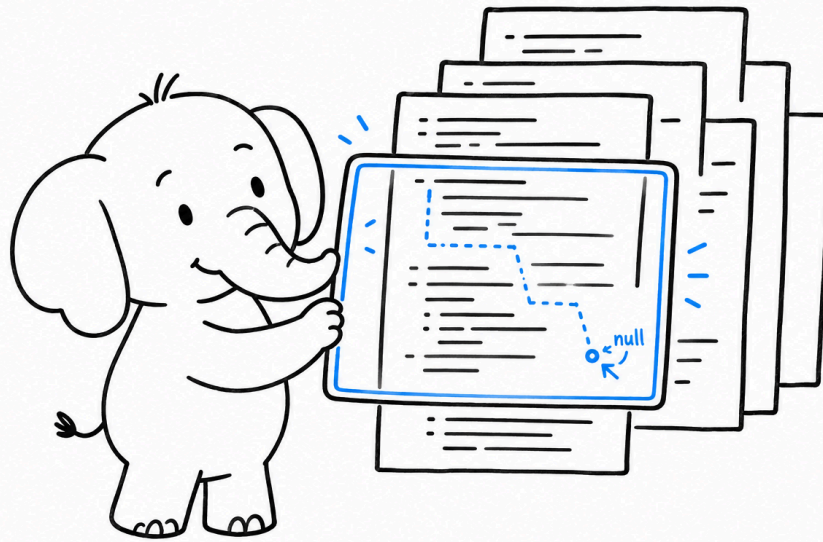
```
Cannot access property $name on User|null.
```

Both tools have levels. PHPStan goes from 0 (obvious mistakes only) to 10 (every `mixed` must be narrowed). Psalm goes the other way, from 8 (lenient) to 1 (strict). New projects start at the strictest level they can bear and never go down. Legacy projects generate a baseline, a file listing every current error so that only new errors fail the build, and then shrink the baseline over time.

A minimal `phpstan.neon`:

```
parameters:
  level: 8
  paths:
    - src
    - tests
```

Both tools read docblocks for what the language cannot express: `@param list<int> $ids`, `@return array<string, User>`, and the `@template` generics that [Types](#) introduced. This is where generics live in PHP. The runtime ignores the docblock; the analyser enforces it.



Code style

PER Coding Style, published by the **PHP-FIG**, is the style guide. It succeeded PSR-12, which succeeded PSR-2, and every framework and most libraries follow it: four spaces, braces on their own line for classes and functions, on the same line for control structures, one class per file. You do not have to learn it. You run a tool.

PHP-CS-Fixer rewrites files to match a rule set. PHP_CodeSniffer reports violations with `phpcs` and fixes what it can with `phpcbf`. Both accept PER as a one-line configuration. For PHP-CS-Fixer, `.php-cs-fixer.dist.php`:

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

$finder = PhpCsFixer\Finder::create()->in([__DIR__ . '/src', __DIR__ .
'/tests']);

return (new PhpCsFixer\Config())
    ->setRules(['@PER-CS' => true])
    ->setFinder($finder);
```

For PHP_CodeSniffer, `phpcs.xml`:

```
<?xml version="1.0"?>
<ruleset name="project">
    <rule ref="PSR12"/>
    <file>src</file>
    <file>tests</file>
</ruleset>
```

Pick one, run it in CI, and stop discussing brace placement in code review.

Upgrades

Rector rewrites your code to a newer PHP version, or to a newer version of a framework, automatically. It knows that `Foo $x = null` must become `?Foo $x = null`, that a `switch` returning a value is a `match`, that a constructor assigning promoted-looking properties can be promoted. Point it at a target version and it does the mechanical part of a migration:

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

use Rector\Config\RectorConfig;
use Rector\ValueObject\PhpVersion;

return RectorConfig::configure()
    ->withPaths([__DIR__ . '/src', __DIR__ . '/tests'])
    ->withPhpVersion(PhpVersion::PHP_84)
    ->withPreparedSets(deadCode: true, codeQuality: true);
```

`vendor/bin/rector --dry-run` shows the diff. Without the flag it applies it. The chapter [Returning to PHP After Years Away](#) is, in effect, a list of what Rector will do to the code you left behind.

Debugging

`var_dump($value)` prints a value with its type and stops nothing. `var_dump($value); exit;` is the fastest debugger there is. Frameworks add a nicer `dump()` and `dd()` (dump and die), but the language one works everywhere.

Xdebug is the step debugger, and the only one. Install the extension, set `xdebug.mode=debug` in `php.ini`, and your editor stops on breakpoints, shows the stack and lets you inspect variables, in

web requests and in CLI scripts alike. It is a development-only extension: it slows everything down, so production builds leave it out, and when you need speed locally,

```
php -d xdebug.mode=off script.php
```

 turns it off for one run.

Editors: PhpStorm has the language built in. VS Code needs a PHP extension. Both read the same docblocks the analysers do, so the generics you write for PHPStan or Psalm also drive autocomplete.

Tying it together

Composer scripts give the project one vocabulary regardless of which tools it picked.

In `composer.json`:

```
{
  "scripts": {
    "test": "phpunit",
    "lint": "php-cs-fixer fix --dry-run --diff",
    "lint:fix": "php-cs-fixer fix",
    "analyse": "phpstan analyse",
    "check": ["@lint", "@analyse", "@test"]
  }
}
```

`composer check` runs the three. Composer puts `vendor/bin` on the path for scripts, so tool names need no prefix. A new team member reads `composer.json` and knows how the project is checked without opening a README.

CI runs the same commands, on every PHP version the project supports (the `php` constraint in `composer.json` says which), plus `composer audit`, which checks the lock file against the known vulnerability database and fails on a hit. `composer outdated` lists what has newer releases; a dependency update bot can open the pull requests for you.

For a local runtime, `php -S localhost:8000 -t public` serves the project as [A Web Request, Without a Framework](#) showed, and the official `php:8.5-cli` Docker image gives everyone the same interpreter. Any `php.ini` setting can be overridden for one command with `php -d memory_limit=1G`. One thing to know: `.env` files are a library convention (several packages parse them into the environment), not something PHP reads on its own.

The trap

Two mistakes, both about timing.

The first is tests that hit the database by default. They pass on the author's machine, take minutes in CI, and stop being run. Test the function, pass the dependency through the constructor, and keep the integration tests in their own folder with their own `phpunit.xml` test suite so they run on purpose.

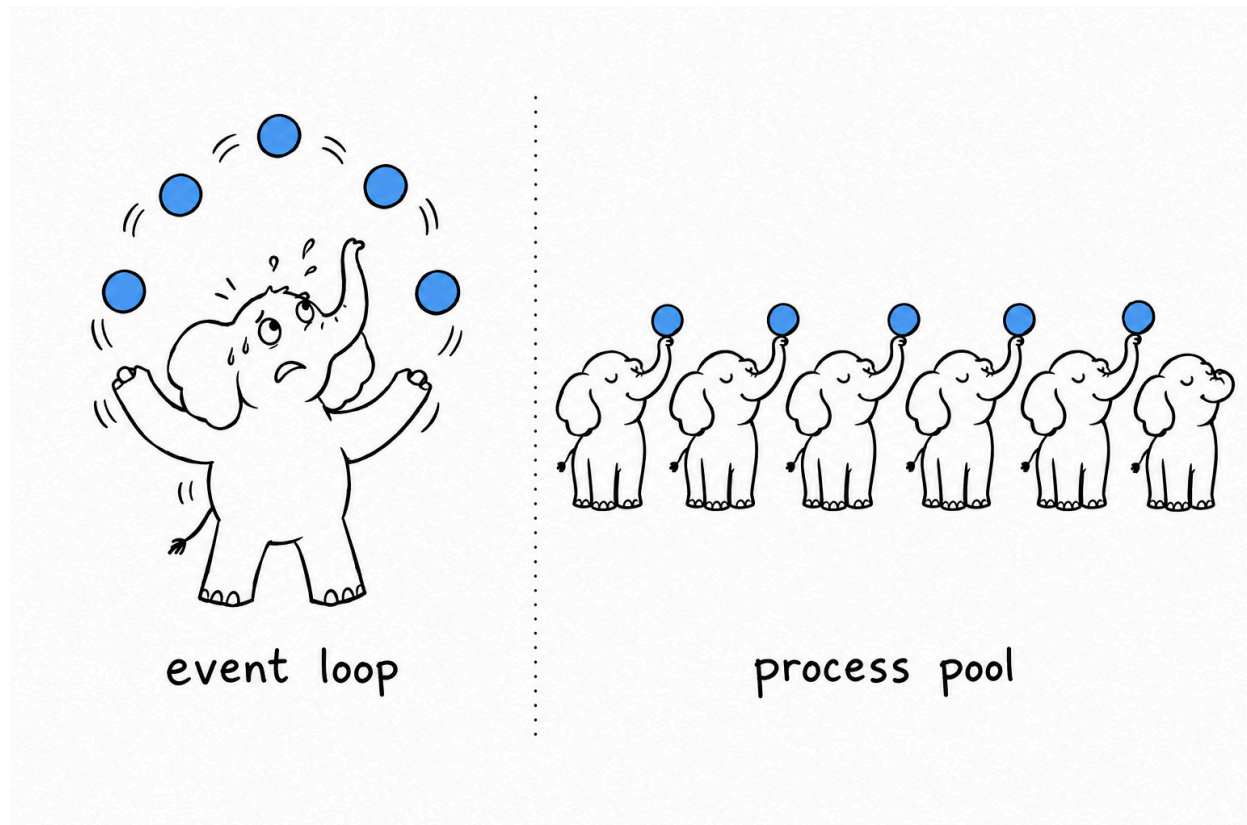
The second is running static analysis at the end. A codebase that reaches level 8 from day one stays there for free. A codebase that first meets PHPStan after two years greets it with four thousand errors and a baseline that nobody shrinks. Add the analyser to the first commit, at the highest level that passes, and raise it when it does.

The tools are in place. What remains is the question every polyglot asks in the first week: where is the async? [Concurrency and Performance](#) answers it.

Concurrency and Performance

PHP is synchronous. One process runs one request, blocks on every I/O call, and that is the design. There is no event loop to feed, no `async` keyword to sprinkle, no goroutine to spawn. Concurrency comes from outside the process: the FPM pool runs as many copies of your script as you have configured, each one alone in its own memory, and the operating system schedules them across cores.

If you come from Node, this feels like a step backward. It is not. Node needs an event loop because one process serves every connection, so one blocking call would freeze them all. PHP gave each request its own process, so blocking costs nothing that anyone else can see. A database query that takes 40 milliseconds holds one worker for 40 milliseconds. The other workers do not notice.



What you do not get

There are no threads in userland. A `parallel` extension exists for thread-safe (ZTS) builds, and almost nobody uses it. The standard build is NTS, non-thread-safe, because the shared-nothing model never needed threads.

There is also no built-in scheduler. **Fibers (PHP 8.1) are stackful coroutines: a function can suspend itself, and whoever holds the fiber can resume it later.** That is all. Nothing decides when

to resume, nothing multiplexes sockets. A fiber is a building block, and async libraries build on it so that ordinary-looking code can yield in the middle of a blocking call.

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

$fiber = new Fiber(function (string $greeting): string {
    $name = Fiber::suspend('who is there?');

    return "$greeting, $name";
});

$question = $fiber->start('Hello'); // runs until suspend()
echo $question, PHP_EOL;           // who is there?

$fiber->resume('Ada');              // runs to the return
echo $fiber->getReturn(), PHP_EOL;  // Hello, Ada
```

You will read code like this inside a library. You will not write it in an application. The Python equivalent is a generator-based coroutine before `asyncio` existed: the mechanism without the runtime.

When you do need async

Some workloads do not fit one-request-one-process: a websocket server holding ten thousand idle connections, long polling, a crawler making a hundred outbound HTTP calls at once. **For those, PHP has async runtimes, and they are libraries, not language features.** Alphabetically: AMPHP, ReactPHP, and Swoole or its fork OpenSwoole, the last two being extensions. The first two are pure PHP built on fibers and stream selection; Swoole brings its own event loop in C.

The worker runtimes from [How PHP Runs](#), FrankenPHP and RoadRunner, are a different answer to a different question: they keep your application booted between requests, still one request at a time per worker. They cut startup cost. They do not make your code concurrent.

Before reaching for any of these, ask whether the problem is actually concurrency. A typical web application never needs them. Ten more FPM workers cost a configuration line.

Background work

The request has thirty seconds and a response to send. Anything longer, or anything the user does not wait for, leaves the request.

The idiom is a queue and a worker. The request pushes a job (a row in a table, a message in a broker) and returns. A CLI script, started by a process supervisor, loops forever pulling jobs and running them. It has no time limit and no memory limit unless you set them, so set them: `memory_limit` in the ini, and a counter that exits cleanly after a few thousand jobs so the supervisor restarts a fresh process. Leaking memory in a loop that never ends is the one place PHP's per-request cleanup does not save you.

Cron covers the scheduled case. The CLI has the rest of the toolbox: `proc_open()` runs a subprocess with pipes, `pcntl_fork()` forks the current one (CLI only, never under FPM), and `curl_multi_exec()` performs parallel HTTP requests with no library at all:

```

<?php
declare(strict_types=1);

$urls = ['https://example.com/a', 'https://example.com/b',
'https://example.com/c'];
$multi = curl_multi_init();
$handles = [];

foreach ($urls as $url) {
    $handle = curl_init($url);
    curl_setopt($handle, CURLOPT_RETURNTRANSFER, true);
    curl_multi_add_handle($multi, $handle);
    $handles[$url] = $handle;
}

do {
    $status = curl_multi_exec($multi, $running);
    if ($running) {
        curl_multi_select($multi);
    }
} while ($running && $status === CURLM_OK);

foreach ($handles as $url => $handle) {
    echo $url, ': ', strlen((string) curl_multi_getcontent($handle)), "
bytes\n";
    curl_multi_remove_handle($multi, $handle);
}

```

Three requests, one wait. That is as much parallelism as most scripts ever need.

Where the time goes

The interpreter is rarely the bottleneck. A request spends its time waiting on the database, the cache, the filesystem and other services. Optimising a loop that runs in two milliseconds while a query takes eighty is the classic mistake, and it is language-independent.

The one setting that matters is OPcache, described in [How PHP Runs](#). Make sure it is on and that `opcache.memory_consumption` is large enough for the whole codebase (`opcache_get_status()` tells you). Two refinements sit on top:

- **Preloading** (`opcache.preload=preload.php`) compiles a list of files once at FPM startup and keeps them linked in memory, so classes need no autoloading at all. It requires a restart to pick up changes, which is why it is a production setting.
- **The JIT** compiles hot code paths to machine code. It makes CPU-bound work faster, sometimes a lot, and makes a typical web request faster by very little. Enable it (`opcache.jit=tracing` , `opcache.jit_buffer_size=64M`), measure, keep it if it helped.

Autoloading has a cost, and Composer can remove most of it. `composer dump-autoload -o` generates a class map so no filesystem lookup happens per class; `--classmap-authoritative` goes further and never touches the filesystem for a class that is not in the map. Both belong in the deployment script. `realpath_cache_size` in the ini, a few megabytes, keeps PHP from re-resolving paths on every request.

Measuring

Nothing above is worth doing before a measurement. The language provides the two primitives:

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

$numbers = range(1, 1_000_000);

$start = hrtime(true);
$doubled = array_map(fn (int $n): int => $n * 2, $numbers);
$mapTime = hrtime(true) - $start;

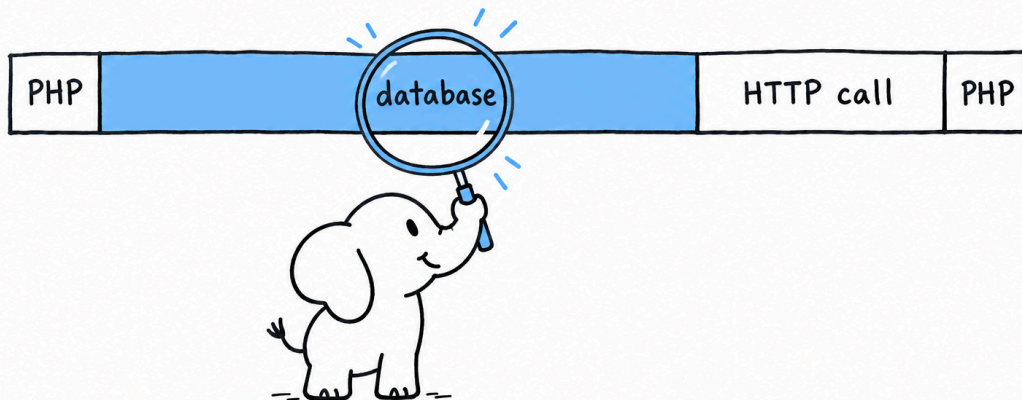
$start = hrtime(true);
$doubled = [];
foreach ($numbers as $n) {
    $doubled[] = $n * 2;
}
$loopTime = hrtime(true) - $start;

printf("array_map: %.1f ms\n", $mapTime / 1e6);
printf("foreach:   %.1f ms\n", $loopTime / 1e6);
printf("peak memory: %.1f MB\n", memory_get_peak_usage() / 1e6);
```

Both lines land in the tens of milliseconds for a million elements. The gap between them is small and depends on the PHP version. The lesson is not which one wins; it is that a million iterations cost less than one slow query, so write the readable one.

For a real profile, Xdebug has a profiler mode (`xdebug.mode=profile`) that writes call graphs your editor can open, and sampling profilers exist as extensions for production, where Xdebug's overhead is unacceptable. Point either at one slow request and read the top of the list.

Memory follows the same rule. A request that builds a hundred-thousand-row array and dies at `memory_limit` needs a generator, as [Functions and Closures](#) showed, not a bigger limit. `unset()` releases a variable, and the garbage collector handles reference cycles on its own; `gc_collect_cycles()` forces a pass, which a long-running worker may call between jobs.



The trap

The first mistake is importing a concurrency model because the last language needed it. An async runtime under a CRUD application adds a layer, a set of libraries that must be fiber-aware, and a class of bugs (shared state between requests) that FPM made impossible. The gain is nothing, because the requests were never waiting on each other.

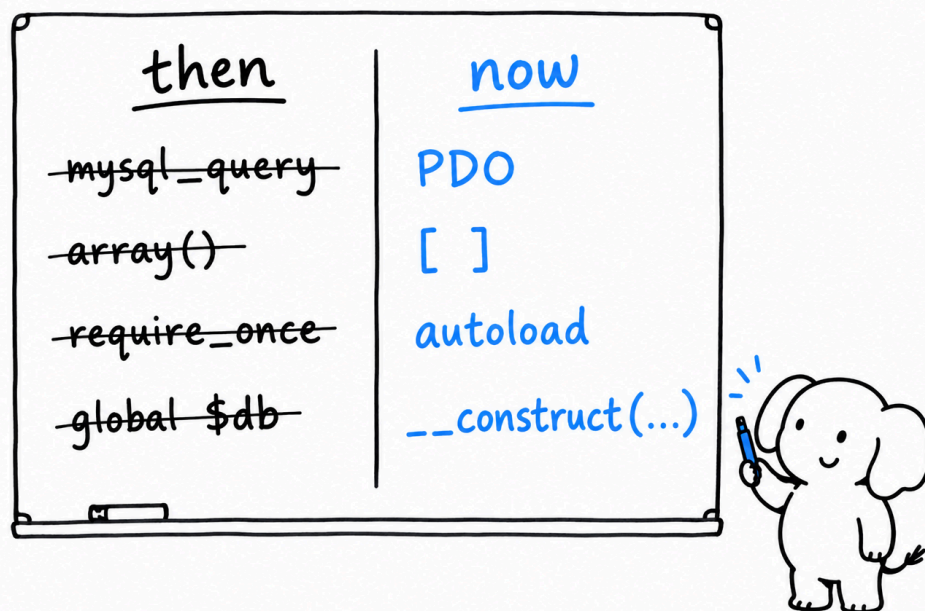
The second is optimising without a number. The JIT flag, the class map, the rewritten loop: each is a hypothesis. `hrtime()` on the slow path, before and after, turns it into a result.

With the runtime, the language and the tools covered, one reader is left. [Returning to PHP After Years Away](#) is for the developer whose last PHP had `mysql_query` in it.

Returning to PHP After Years Away

You wrote PHP once. Maybe a lot of it, maybe in 2008, maybe in 2015 on a project that was already old. You remember `mysql_query`, `array()`, `require_once` at the top of every file, and a language that let you get away with anything. Now you are back, and the first thing to know is this: **almost every habit you remember has a modern replacement, and most of the old forms are deprecated or gone.**

This chapter is a list of pairs. On the left, what you remember. On the right, what you write today. The rest of the book explains each replacement in depth; this is the map.



The database

You remember building SQL by hand and passing it to `mysql_query()`. The **`mysql_*` functions were removed in PHP 7.0**. Code that calls them does not run on any supported version of PHP. The replacement is PDO with prepared statements, which also closes the injection hole the old style opened:

```

<?php
declare(strict_types=1);

// then
// $result = mysql_query("SELECT * FROM users WHERE id = " .
$_GET['id']);

// now
$pdo = new PDO('sqlite::memory:', options: [PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION]);
$stmt = $pdo->prepare('SELECT * FROM users WHERE id = :id');
$stmt->execute(['id' => (int) ($_GET['id'] ?? 0)]);
$user = $stmt->fetch(PDO::FETCH_ASSOC);

```

`mysqli` still exists and is fine, but PDO speaks to every database with one API. [A Web Request, Without a Framework](#) shows it in context.

Loading code

You remember a wall of `require_once` lines, or a home-made `__autoload()` function. **Today one line loads everything:** `require __DIR__ . '/vendor/autoload.php';`. Composer generates that file from a namespace-to-folder map in `composer.json`, and `__autoload()` itself was removed in 8.0. Classes live one per file, named after the class, and are found on first use. [Namespaces, Composer, and Autoloading](#) covers the setup, which takes five minutes.

Composer also replaced the habit of copying a library into your project. Since 2012, `composer require vendor/package` fetches it from Packagist, pins the version in a lockfile, and updates it when you ask.

Syntax that got shorter

Small changes, but you will see them on every line.


```
// then
$list = array(1, 2, 3);
$name = isset($_GET['n']) ? $_GET['n'] : 'anon';
$double = function ($x) use ($factor) {
    return $x * $factor;
};
$callback = array($obj, 'method');
call_user_func_array($callback, array(1));
```

```
// now
$list = [1, 2, 3];
$name = $_GET['n'] ?? 'anon';
$double = fn($x) => $x * $factor;
$callback = $obj->method(...);
$callback(1);
```

[[array](#)] arrived in 5.4, [??](#) in 7.0, arrow functions [fn](#) in 7.4, and the first-class callable syntax [\\$obj->method\(...\)](#) in 8.1. String callables like `'Class::method'` and `call_user_func()` still work; nobody writes them anymore because the new form is checked by the editor and the analyser, and a callable is simply called with parentheses.

Types

You remember functions that accepted anything and returned whatever. Today functions declare their types, and one line per file makes PHP enforce them.

```

<?php
declare(strict_types=1);

// then
// function total($items, $rate) { ... }

// now
function total(array $items, float $rate): float
{
    return array_sum($items) * $rate;
}

total([10, 20], '1.2'); // TypeError: must be of type float, string
given

```

Scalar types came in 7.0, return types with them, nullable `?int` in 7.1, union types in 8.0, and typed properties in 7.4. `declare(strict_types=1)` switches off silent coercion for calls made from that file. [Types](#) has the precise rules.

Control flow and constants

You remember `switch` with its fallthrough and loose comparison, and a class full of `const STATUS_ACTIVE = 'active';`. `match` replaced the first, `enums` the second.

```

<?php
declare(strict_types=1);

enum Status: string
{
    case Active = 'active';
    case Archived = 'archived';
}

function label(Status $status): string
{
    return match ($status) {
        Status::Active => 'In use',
        Status::Archived => 'Put away',
    };
}

```

`match` (8.0) compares with `===`, does not fall through, and throws if no arm fits. Enums (8.1) are real types: a function typed `Status` cannot receive the string `'deleted'`. [Enums and match](#) goes further.

Classes

You remember a private property, a getter, and a setter, times ten per class. **Constructor promotion** and **`readonly`** collapse that to one line per property.

```

<?php
declare(strict_types=1);

final class Money
{
    public function __construct(
        public readonly int $cents,
        public readonly string $currency,
    ) {
    }
}

$price = new Money(1999, 'EUR');
echo $price->cents; // 1999
$price->cents = 0; // Error: Cannot modify readonly property

```

Promotion arrived in 8.0, `readonly` in 8.1. Property hooks (PHP 8.4) handle the case where a getter really did compute something, by attaching `get` and `set` blocks to the property itself. [Classes](#) shows all of it.

Two things you remember are now removed or deprecated. PHP 4 style constructors, where the method was named after the class, were removed in 8.0. Dynamic properties, assigning `$obj->whatever` without declaring it, have been deprecated since 8.2 and are planned to become an error in the next major version.

Errors

You remember `@mysql_connect(...)` or `die('no db')`, and warnings printed into the middle of the page. **Today errors are exceptions, and you catch them.** Internal functions throw `TypeError` and `ValueError` instead of returning `false` with a warning, division by zero throws, and frameworks convert the remaining warnings into exceptions with an error handler. `@` still exists. Treat it as a smell. [Errors and Exceptions](#) explains the hierarchy.

Globals

You remember `global $db;` at the top of every function. **Today the dependency comes in through the constructor.**

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

final class UserRepository
{
    public function __construct(private readonly PDO $pdo)
    {
    }
}
```

The object that needs a database receives one. It can be tested with a different one. Frameworks automate the wiring with a container; the idea needs none.

`register_globals`, which turned every request parameter into a variable, was removed in 5.4. `$_REQUEST` still exists and nobody uses it: read `$_GET` or `$_POST` and say which you mean. `extract()` and variable variables (`$$name`) are still legal and appear in no modern codebase.

Text and dates

You remember `date('Y-m-d', $timestamp)` and `utf8_encode()`. Dates are **`DateTimeImmutable`** objects, and text is UTF-8 everywhere.

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

$due = new DateTimeImmutable('2026-03-01', new DateTimeZone('UTC'));
echo $due->modify('+30 days')->format('Y-m-d'); // 2026-03-31
```

`utf8_encode()` and `utf8_decode()` only ever handled Latin-1 and have been deprecated since 8.2; `mb_convert_encoding()` does the job for any encoding. `ereg_*` was removed in 7.0, `preg_*` stayed. The `${var}` form of string interpolation has been deprecated since 8.2; write `{ $var }`. [Strings](#), [Numbers](#), [Dates](#), and [JSON](#) has the rest.

Small things that are gone

- `each()` and `create_function()`, removed in 8.0. Use `foreach` and closures.
- The `(unset)` cast, removed in 8.0.

- Implicit nullable parameters, `Foo $x = null` without the `?`, deprecated in 8.4. Write `?Foo $x = null`.
- `split()`, removed in 7.0. Use `explode()` or `preg_split()`.
- `mb_internal_encoding('UTF-8')` at the top of files. The default has been UTF-8 since 5.6.

Getting a PHP 5 codebase onto PHP 8

It will not run as is. The `mysql_*` calls alone guarantee that. The mechanical part, at least, is automated: **Rector rewrites old syntax to new syntax**, version by version, from a config that names your target. Point it at the code, review the diff, run the tests you hopefully have, and repeat. PHPStan or Psalm then find what Rector could not. [Tests, Static Analysis, and Tooling](#) introduces both.

Plan for the work. A site still running PHP 5 today is running a version that stopped receiving security fixes in 2018. It is a liability before it is a codebase.

The rhythm of releases

PHP now ships one minor version every November: 8.0 in 2020, 8.1 in 2021, and so on to 8.5 in 2025. Each version gets two years of active support and two more of security fixes; php.net/supported-versions has the dates. A yearly release means a yearly, small, deprecation list to read, and a codebase that never drifts far from current.

Judge the language by its release notes, not by the codebase you are returning to. The codebase is a snapshot of when it was written. The language kept moving.

[Where to Go from Here](#) lists the places to keep up.

Where to Go from Here

The book stops here. The language does not, and neither does the project you were handed. **Four places keep you current, and none of them belongs to a vendor.**

The manual

php.net is the reference, and it is a good one. Every function has a page with its signature, its changelog by version, and examples that run. The user-contributed notes under each page are a mixed bag; the text above them is not. Bookmark the migration guides, one per version (php.net/manual/en/migration85.php and its siblings): they list every deprecation and every new feature, and reading the one for your project's next version is the cheapest upgrade planning you will do.

The RFC wiki

Every change to the language goes through a public proposal, a discussion on the internals mailing list, and a vote by the core developers. The proposals live at wiki.php.net/rfc, accepted, declined, and in progress. **Reading the accepted RFCs for a version tells you not just what changed but why**, with the alternatives that were rejected and the arguments against. When a feature looks odd, its RFC usually explains the constraint that made it so.

The Foundation and the FIG

The [PHP Foundation](http://www.php-foundation.org) funds the core developers who maintain the interpreter and shepherds the language's direction. Its blog reports on releases and on what is being worked on. The [PHP-FIG](http://www.php-fig.org) (Framework Interoperability Group) publishes the PSRs and the PER Coding Style, the interfaces and conventions that let libraries from different authors work together. When a codebase mentions PSR-something, the FIG site has the two-page spec.

The framework your project uses

Most PHP projects sit on a framework, and the framework's own documentation is where to learn it. Full-stack frameworks: CakePHP, Laminas, Laravel, Symfony, Yii. Micro-frameworks built around PSR-15 middleware: Mezzio, Slim. Content platforms with their own conventions: Drupal, Joomla, TYPO3, WordPress.

One habit pays off from the first day. **When you read framework code, sort what you see into two piles: the language, and the framework.** A `readonly` promoted constructor, an enum in a `match`, a `?->` chain: that is PHP, and it means the same thing everywhere. A facade, a service container binding, a magic `__call` that routes to a query builder: that is the framework, and it is documented by the framework. Confusing the two piles is how people come to believe that PHP is whatever their first framework made it look like.

A longer road

This book skipped the basics on purpose. If you want the ground-up version, with a small game, a command line tool and a web application built from nothing, the companion volume, *The PHP Book*, takes that road at a walking pace. It is published from the same repository as this one.

Beyond the written word, PHP has user groups in most cities and conferences on most continents. A room full of people who have solved the problem you are about to meet is worth an afternoon.

The promise

Two to three hours ago, you opened a codebase and saw `$this->` and `::` and a runtime model you did not recognise. **You now know how PHP runs, how it types, how it packages code, and where it will surprise you.** The old reputation is filed where it belongs, in the chapter about the past.

The codebase is legible. Go read it.

Appendix

Three lookup tables for the moments when you need a fact, not a chapter. [A - Coming from Python, JavaScript, or Java](#) maps the constructs you already know to their PHP spelling, one row each. [B - PHP 8.0 to 8.5 at a Glance](#) lists what each version added, so you know what your project's PHP can use. [C - Vocabulary](#) defines the words that come up in PHP conversations and nowhere else.

A - Coming from Python, JavaScript, or Java

One row per construct, the spelling you know on the left, the modern PHP spelling on the right.
Where PHP offers an old form and a new one, only the new one is listed.

Concept	Python	JavaScript	Java
Variable	<code>x = 1</code>	<code>let x = 1</code>	<code>int x = 1;</code>
Constant	<code>X = 1</code> (convention)	<code>const X = 1</code>	<code>static final int X = 1;</code>
String interpolation	<code>f"hi {name}"</code>	<code>`hi \${name}`</code>	<code>"hi " + name</code>
String concat	<code>a + b</code>	<code>a + b</code>	<code>a + b</code>
Multi-line string	<code>"""..."""</code>	<code>`...`</code>	<code>"""..."""</code>
Integer division	<code>a // b</code>	<code>Math.trunc(a / b)</code>	<code>a / b</code>
Exponent	<code>a ** b</code>	<code>a ** b</code>	<code>Math.pow(a, b)</code>
Strict equality	<code>a == b</code>	<code>a === b</code>	<code>a.equals(b)</code>
Null coalescing	<code>a if a is not None else b</code>	<code>a ?? b</code>	<code>Optional.ofNullable(a).orElse(b)</code>
Ternary	<code>a if c else b</code>	<code>c ? a : b</code>	<code>c ? a : b</code>
List literal	<code>[1, 2]</code>	<code>[1, 2]</code>	<code>List.of(1, 2)</code>
Dict literal	<code>{"k": 1}</code>	<code>{k: 1}</code>	<code>Map.of("k", 1)</code>
List append	<code>xs.append(v)</code>	<code>xs.push(v)</code>	<code>xs.add(v)</code>
Dict lookup with default	<code>d.get("k", 0)</code>	<code>d.k ?? 0</code>	<code>d.getOrDefault("k", 0)</code>
Length	<code>len(xs)</code>	<code>xs.length</code>	<code>xs.size()</code>
String length	<code>len(s)</code>	<code>s.length</code>	<code>s.length()</code>
Slice	<code>xs[1:3]</code>	<code>xs.slice(1, 3)</code>	<code>xs.subList(1, 3)</code>
Iterate list	<code>for v in xs:</code>	<code>for (const v of xs)</code>	<code>for (var v : xs)</code>
Iterate dict	<code>for k, v in d.items():</code>	<code>for (const [k, v] of Object.entries(d))</code>	<code>for (var e : d.entrySet())</code>
Map	<code>[f(v) for v in xs]</code>	<code>xs.map(f)</code>	<code>xs.stream().map(f)</code>
Filter	<code>[v for v in xs if p(v)]</code>	<code>xs.filter(p)</code>	<code>xs.stream().filter(p)</code>
Lambda	<code>lambda x: x * 2</code>	<code>x => x * 2</code>	<code>x -> x * 2</code>
Closure capture	by reference	by reference	effectively final

Concept	Python	JavaScript	Java
Default argument	<code>def f(x=1):</code>	<code>function f(x = 1)</code>	overload
Named argument	<code>f(x=1)</code>	<code>f({x: 1})</code>	none
Variadic	<code>def f(*xs):</code>	<code>function f(...xs)</code>	<code>void f(int... xs)</code>
Class	<code>class A:</code>	<code>class A {}</code>	<code>class A {}</code>
Constructor	<code>def __init__(self, x):</code>	<code>constructor(x) {}</code>	<code>A(int x) {}</code>
Instance member	<code>self.x</code>	<code>this.x</code>	<code>this.x</code>
Static member	<code>A.x</code>	<code>A.x</code>	<code>A.x</code>
Interface	<code>Protocol</code>	none (TS: <code>interface</code>)	<code>interface A {}</code>
Enum	<code>class C(Enum):</code>	none (TS: <code>enum</code>)	<code>enum C { A, B }</code>
Catch exception	<code>except E as e:</code>	<code>catch (e)</code>	<code>catch (E e)</code>
Safe navigation	none	<code>a?.b</code>	<code>Optional.map</code>
String to int	<code>int(s)</code>	<code>parseInt(s)</code>	<code>Integer.parseInt(s)</code>
Type check	<code>isinstance(x, A)</code>	<code>x instanceof A</code>	<code>x instanceof A</code>
Print	<code>print(x)</code>	<code>console.log(x)</code>	<code>System.out.println(x)</code>
Import	<code>from a import B</code>	<code>import { B } from 'a'</code>	<code>import a.B;</code>
Package manager	pip, pyproject.toml	npm, package.json	Maven, pom.xml
Test runner	pytest	Jest, Vitest	JUnit
Formatter	black, ruff	Prettier	google-java-format
Static analysis	mypy, pyright	tsc	the compiler
Run a script	<code>python a.py</code>	<code>node a.js</code>	<code>java A.java</code>
REPL	<code>python</code>	<code>node</code>	<code>jshell</code>

Four things that are different, not just spelled differently

- **A request starts from nothing and ends with nothing.** No process stays alive between two requests. [How PHP Runs](#).
- **Arrays are values.** Assign or pass one and you get a copy. Objects are handles. [Arrays](#).
- **Closures capture by value, at creation time.** A later change to the outer variable is not seen. [Functions and Closures](#).
- **Strict typing is a per-file switch.** `declare(strict_types=1)` governs the calls made from that file, and only that file. [Types](#).

B - PHP 8.0 to 8.5 at a Glance

One section per version, headline features only. Find your project's version, then read down from there to see what you can use.

PHP 8.0, November 2020

- Named arguments: `str_pad(string: 'a', length: 3)`.
- Attributes: `#[Route('/home')]` as structured metadata read by reflection.
- Constructor property promotion: `public function __construct(private int $x) {}`.
- Union types: `int|string $id`.
- `match` expression: strict comparison, no fallthrough, exhaustive.
- Nullsafe operator: `$user?->address?->city`.
- `mixed` and `static` as types.
- `throw` as an expression: `$x = $y ?? throw new Exception();`.
- `str_contains()`, `str_starts_with()`, `str_ends_with()`.
- `Stringable` interface, implemented automatically by any class with `__toString()`.
- `WeakMap`.
- JIT compiler, inside OPcache.
- Saner string-to-number comparisons: `0 == 'foo'` is `false`.
- Trailing comma allowed in parameter lists.
- Internal functions throw `TypeError` and `ValueError` instead of warning and returning `null` or `false`.

PHP 8.1, November 2021

- Enums, pure and backed: `enum Suit: string { case Hearts = 'H'; }`.
- `readonly` properties: `public readonly int $x`.
- First-class callable syntax: `strlen(...)`.
- Fibers: stackful coroutines, the building block of async libraries.
- `new` in initializers: `public function __construct(private Logger $l = new NullLogger()) {}`.
- Pure intersection types: `Countable&Traversable`.
- `never` return type.
- `final` class constants.

- Array unpacking with string keys: `[...$defaults, ...$options]`.
- `array_is_list()`.
- Explicit octal notation: `0o16`.

PHP 8.2, December 2022

- `readonly` classes: `final readonly class Point {}`.
- Disjunctive normal form types: `(A&B)|null`.
- Standalone `true`, `false` and `null` types.
- Dynamic properties deprecated; `#[\AllowDynamicProperties]` opts a class back in.
- `#[\SensitiveParameter]` to redact an argument from stack traces.
- Constants in traits.
- `Random\Randomizer` and the `random` extension.
- Enum cases usable in constant expressions.

PHP 8.3, November 2023

- Typed class constants: `const string NAME = 'x';`.
- `#[\Override]` attribute: the engine checks that a parent method exists.
- `json_validate()`.
- Dynamic class constant fetch: `Foo::{$name}`.
- `readonly` properties can be reinitialised inside `__clone()`.
- `Randomizer::getBytesFromString()`, `Randomizer::getFloat()`.
- Negative array indices behave consistently.
- `mb_str_pad()`.

PHP 8.4, November 2024

- Property hooks: `public string $name { get => ...; set => ...; }`.
- Asymmetric visibility: `public private(set) int $x`.
- `new` without parentheses when chaining: `new Foo()->bar()`.
- Lazy objects: `ReflectionClass::newLazyGhost()`, `newLazyProxy()`.
- `#[\Deprecated]` attribute for your own code.
- `array_find()`, `array_find_key()`, `array_any()`, `array_all()`.
- `mb_trim()`, `mb_ltrim()`, `mb_rtrim()`, `mb_ucfirst()`, `mb_lcfirst()`.
- New DOM extension with an HTML5 parser: `Dom\HTMLDocument`.

- BCMath object API: `BcMath\Number`.
- PDO driver subclasses: `Pdo\Sqlite`, `Pdo\MySQL`, `Pdo\Pgsql`, via `Pdo::connect()`.
- `request_parse_body()` for form bodies on any HTTP method.
- Implicitly nullable parameters (`Foo $x = null` without `?`) deprecated.
- `exit` and `die` are now functions.

PHP 8.5, November 2025

- Pipe operator: `$slug = $title |> trim(...) |> strtolower(...);`.
- `clone` with property updates: `clone($point, ['x' => 3])`.
- `#[\NoDiscard]` attribute, warning when a return value is ignored; `(void)` cast to silence it on purpose.
- `array_first()`, `array_last()`.
- Closures and first-class callables in constant expressions (attribute arguments, default values, constants).
- Attributes on constants.
- New `uri` extension: `Uri\Rfc3986\Uri`, `Uri\WhatWg\Uri`.
- Fatal errors now include a backtrace.
- `get_error_handler()`, `get_exception_handler()`.
- `#[\DelayedTargetValidation]`.
- `PHP_BUILD_DATE` constant.
- The backtick shell-execution operator is deprecated.

Support policy

Each version gets two years of active support (bug fixes) followed by two years of security fixes only. A version released in November 2025 is therefore active until late 2027 and patched until late 2029. Dates move with each release; php.net/supported-versions has the current table. Running a version past its security window is a risk you take knowingly or not at all.

Finding your project's version

`php -v` prints the interpreter you run locally. `composer.json` declares what the project supports under `require.php` (`"php": "^8.3"`), and `config.platform.php` pins the version Composer resolves against, which is the one to trust when the two differ. Production may run something else again; `phpinfo()` or `php -v` on the server settles it.

C - Vocabulary

The words that come up in PHP conversations and rarely anywhere else. One or two sentences each.

SAPI. Server API: the layer that connects the interpreter to whatever runs it. The command line, PHP-FPM, `mod_php` and the embedded runtimes are all SAPIs over the same engine.

CLI. The command-line SAPI, invoked as `php file.php`. It has no execution time limit and no memory limit by default, unlike the web SAPIs.

FPM. FastCGI Process Manager: a pool of PHP processes waiting for requests behind a web server. The standard way to serve PHP.

FastCGI. The protocol the web server uses to hand a request to FPM and read the response back.

mod_php. The Apache module that embeds PHP in the web server process itself. Older than FPM and still around.

OPcache. The bytecode cache. Compiled files stay in shared memory so the next request skips parsing. On in every serious setup.

JIT. Just-in-time compilation of hot code to machine code, inside OPcache, since 8.0. Helps CPU-bound scripts; changes little for typical web traffic.

Preloading. An OPcache option that compiles and links a set of files once at server start, so every request begins with them already loaded.

APCu. A memory cache shared between the PHP processes of one machine. For values you compute once and read often, when an external store is overkill.

Extension. A compiled C module that adds functions or classes to PHP: `pdo_mysql`, `intl`, `mbstring`, `xdebug`. `php -m` lists the loaded ones.

PECL. The historical repository of extensions not bundled with PHP, with its own installer.

PIE. The newer installer for extensions, Composer-style, meant to succeed the PECL workflow.

ZTS and NTS. Thread-safe and non-thread-safe builds of the interpreter. NTS is the default and the one FPM uses; ZTS exists for the rare threaded SAPIs and the `parallel` extension.

php.ini. The configuration file. The CLI and FPM read different ones, which explains most “it works in the terminal” mysteries.

Composer. The package manager. Reads `composer.json`, writes `composer.lock`, fills `vendor/`, generates the autoloader.

Packagist. The public package registry Composer fetches from.

vendor. The folder Composer installs packages into. Never edited, never committed.

Autoload. The mechanism that loads a class file the first time the class is used, so no `require` line is ever written by hand.

PSR-4. The standard mapping from namespace to folder that autoloaders follow:

`App\Billing\Invoice` lives in `src/Billing/Invoice.php`.

PHP-FIG. Framework Interoperability Group: the body that publishes the PSRs and the coding style.

PSR. PHP Standards Recommendation: a numbered interface or convention (PSR-3 logging, PSR-7 HTTP messages, PSR-15 middleware) that libraries agree on so they can be swapped.

PER Coding Style. The current code style standard from the FIG, successor of PSR-12. What formatters enforce.

RFC. Request for Comments: the public proposal every language change goes through before a vote of the core developers.

PHP Foundation. The non-profit that, since 2021, employs core developers to maintain and advance the interpreter.

php-src. The source repository of the interpreter itself, written in C.

Zend Engine. The core of the interpreter: the compiler and the executor. The name survives in a few settings and error messages.

Superglobal. A built-in array visible in every scope: `$_GET`, `$_POST`, `$_SERVER`, `$_COOKIE`, `$_FILES`, `$_SESSION`, `$_ENV`.

Docblock. A `/** ... */` comment above a symbol, carrying `@param` and `@return` annotations that editors and static analysers read. Where generics live.

Attribute. Structured metadata attached to a class, method, property or parameter with `#[...]`, read through reflection. What annotations were in Java.

Trait. A block of methods and properties copied into any class that `use`s it. Horizontal reuse without inheritance.

Enum. A type with a fixed set of named cases, optionally backed by an int or a string, with methods and interfaces. Since 8.1.

Fiber. A stackful coroutine that can suspend and resume from anywhere in its call stack. The primitive async libraries build on; not something application code drives directly.

Generator. A function that `yield`s values one at a time and keeps its state between calls. Lazy iteration without building an array.

SPL. Standard PHP Library: the bundled set of data structures, iterators, exceptions and interfaces such as `ArrayIterator`, `SplQueue`, `Countable`, `RuntimeException`.

PDO. PHP Data Objects: the database abstraction with one API over every driver, prepared statements included.

mbstring. The multibyte string extension. `mb_strlen`, `mb_substr` and friends count characters where the plain functions count bytes.

intl. The internationalisation extension: collation, number and date formatting, Unicode normalisation, wrapping the ICU library.

Xdebug. The step debugger and profiler. Development only.

PHPUnit. The xUnit-style test framework most PHP tests are written with.

Pest. A test framework with a describe-and-expect syntax, running on PHPUnit's engine.

PHPStan. A static analyser that finds type errors and impossible code without running it, with levels from 0 to 10.

Psalm. The other static analyser, with levels from 8 down to 1 and a focus on type soundness and taint analysis.

Rector. An automated refactoring tool that rewrites code to a newer PHP version or a newer library API.

phpt. The test file format of the interpreter itself, in `php-src`. You meet it if you contribute to PHP or read its bug reports.

strict_types. The per-file declaration `declare(strict_types=1);` that makes scalar type mismatches on calls from that file throw instead of coerce.

Copy-on-write. The engine trick that makes array assignment cheap: the copy shares memory with the original until one of them is modified.

Late static binding. `static::` resolving to the class the call was made on, not the class where the method is written. `self::` is the other one.

Magic method. A method with a reserved double-underscore name that the engine calls on its own: `__construct`, `__toString`, `__get`, `__call`, `__clone`, `__invoke`.