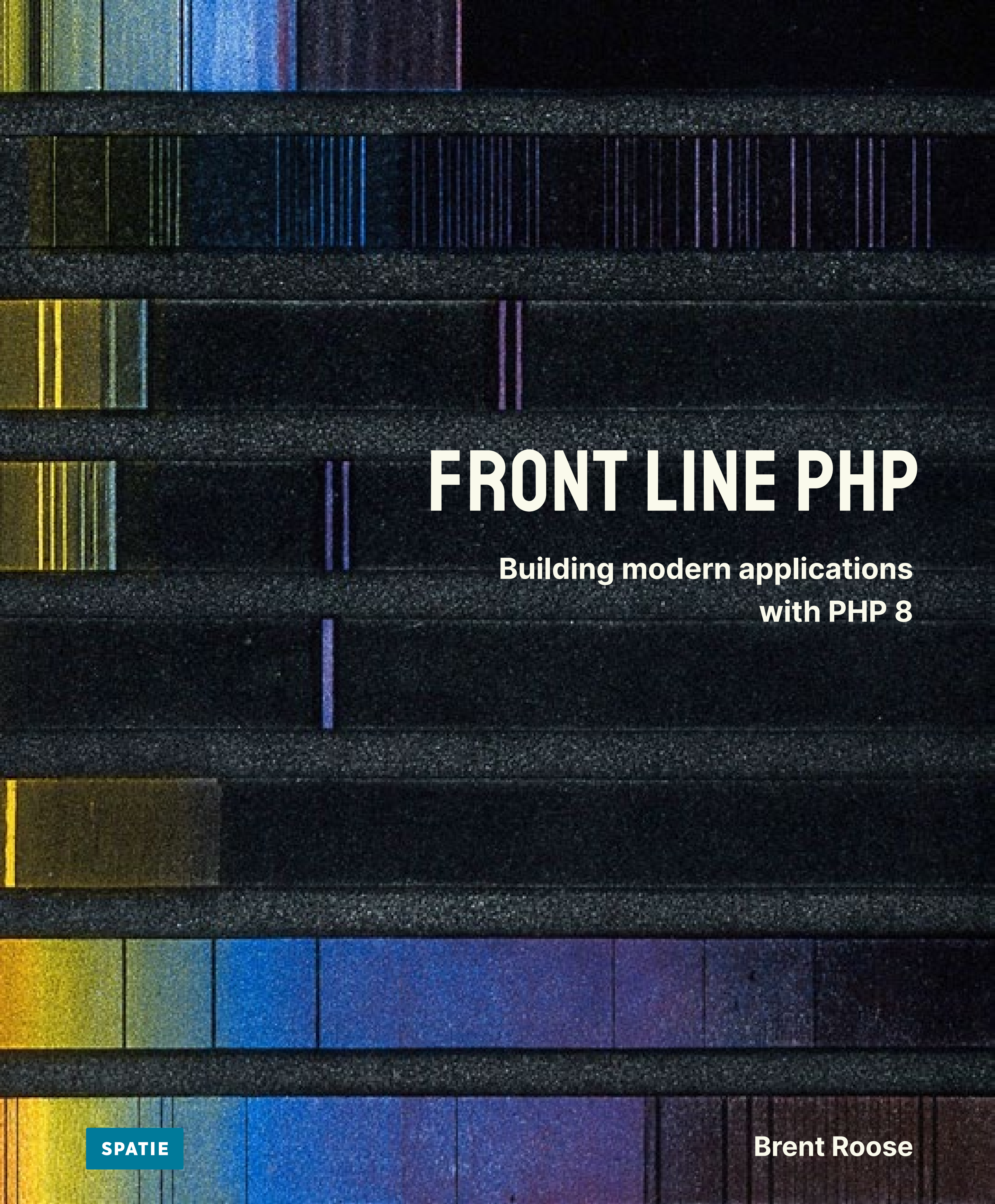

FRONT LINE PHP

Building modern applications with PHP 8

SPATIE

Brent Roose

FRONT LINE PHP

Building modern applications
with PHP 8

Brent Roose

Published 2020 by Spatie.

Reviewers: Freek Van der Herten, Matthias Noback, Stephen Moon, Wouter Brouwers
Design & composition: Sebastian De Deyne, Willem Van Bockstal

This book is a distillation of the lessons learned by the Spatie team the past three years: Adriaan, Alex, Brent, Jef, Freek, Niels, Rias, Ruben, Sebastian, Willem, and Wouter.

Version 3a15e11-1
To report errors, please send a note to info@spatie.be.

Created in Belgium

TABLE OF CONTENTS

FOREWORD

By Roman Pronskiy

PHP has changed the world. More than 5 million developers are now using this language worldwide to create a variety of applications. But the world is changing PHP as well. Rasmus Lerdorf, the creator of the language, once noted that "a good solution should steal/borrow existing technology". And this is true. PHP takes the best ideas from other ecosystems and improves on them.

It's been 5 years since the release of PHP 7.0. In this time, as the community has matured and needed new features, the language team has been listening to the community and implementing the features. This is what has led PHP to introduce stricter typing, more concise syntax, static analyzers, etc.

“It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is most adaptable to change.” - Leon C. Megginson’s interpretation of Darwin’s ideas is an insight that guides PHP year after year.

I was still in school when I first saw PHP, and at the time I had no idea that I would go on to work with it professionally. Back then, in 2004, the main problem the internet had was searching for information. Today, because so much information is readily available, the problem is not searching information but filtering and identifying what is useful. It is, to lean on an old cliché, like looking for a $needle in a $haystack – or was it the other way around? Anyway, how do we dig up the really valuable things in these endless feeds mixed with ads and notifications?

It's impossible to give a comprehensive answer for all areas, but for PHP the answer is here – Front Line PHP.

This book gives a core of knowledge about everything that concerns the PHP’s development today and in the near future. This is not a paraphrase of documentation, but rather a thoughtful and meaningful development experience in short and clear formulations. It offers a fresh look at the newest features of the language, as well as at some that are already familiar.

Writing quality code in PHP has simultaneously become both easier and more complicated. On the one hand, there are many new features and tools at a developer's disposal. On the other hand, you need to juggle them elegantly in order to solve real problems beautifully and easily.

That’s what this book is really about. Complex things in simple words. How to apply them. Principles and practices. It is a fishing rod, in a sea full of fish.

There has never been a better time to work with PHP. And this book is one of the best ways to learn it.

I hope you enjoy it!

PREFACE

When I started writing PHP, there were little resources available that could teach the quality I expected to learn. Many online resources are either outdated or focused on absolute beginners. So about four years ago, I decided I wanted to contribute in a qualitative way to the PHP community. I started a blog, which has grown quite popular by now, and the writing never stopped.

The downside of a blog is that each post is a small island on its own. You can't expect visitors to have read all your previous posts. It's difficult to tell a story from start to end, to have a continuous narrative on a blog.

Meanwhile I was fortunate enough to be able to work at a company that valued all I've written before, and where each colleague is passionate about their work. Together we decided to work on something that's more than individual posts or thoughts, something that tells the story of writing modern day PHP, something that would eventually become this book.

This book is going to be a great stepping stone for all PHP developers; whether you're a beginner or expert: you'll be able to learn a lot. This book is a unified story, it connects the dots that we've been working on for several years. This was my personal goal when writing it, and I believe we've managed to do just that.

I wish you a great time reading, let's begin!

PART I

PHP, THE LANGUAGE

CHAPTER 01

PHP TODAY

I've been programming with PHP for 10 years now. During that time I've come to really appreciate and enjoy the language. I realise not everyone would agree with that sentiment: many consider PHP to be a broken and quirky language. Yet still, it runs large portions of the web. So why do I enjoy writing it and why is it used so abundantly? PHP's success, for over more than 25 years, isn't thanks to it being the perfect language, but rather because it's an accessible one. PHP is easy to set up, run and to write. Unfortunately, it's also easy to write terrible code, but still have a working program. There are lots of online resources showing quick, easy and dirty ways to get things done in PHP. Quick and easy, sure; but also insecure, unmaintainable and slow.

On top of that, it was a language where you seemingly almost always had to write quirky code to get things working. That was certainly the case 10 years ago. However, those days are gone, at least if you choose to embrace PHP's modern features. It has seen a remarkable evolution over the past decade and has become a language I enjoy everyday.

PHP isn't the same old crappy language remembered by many. Today, the language is used to build high-profit, large-scale applications, and it does so just fine. With PHP 7 and PHP 8 we received a significant improvement in the language syntax and feature-set, as well as its performance. In addition, with frameworks like Laravel and Symfony came versatile and flourishing communities.

PHP still isn't the perfect web-development language. Just like Ruby, Python, Java, C# and JavaScript also aren't. Yet PHP programmers manage to achieve great results, and enjoy writing code whilst doing so. Throughout this book we'll look at PHP as it is today. We'll:

- explore what changed over the years,
- ponder on how the language will evolve in the future,
- discuss the ecosystem and community surrounding it, and
- show how PHP can be used to write modern-day, performant, and maintainable code.

We'll also cover topics like:

- new syntax and language constructs,
- PHP's type system,
- what proper object oriented programming looks like in PHP,
- new core technologies like preloading, the JIT and FFI, and
- writing readable code.

Finally, we'll explore PHP's ecosystem:

- MVC and testing frameworks,
- async PHP, and
- tooling like static analysers.

You'll learn to love the modern language that is PHP.

Naturally we'll see lots of code in this book, so I figure it's important to understand what that code exactly means. In the next few chapters we'll start from the ground up and discuss everything new in PHP's recent evolution.

CHAPTER 02

NEW VERSIONS

Since this book aims to bring you up to speed with modern PHP practices, it's important to know what happened to the language over the past decade. With the development and release of PHP 7, the PHP landscape changed dramatically. I like to think of the PHP 7.x versions as a maturity phase for the language, so this is where we will start.

First and foremost, PHP 7.0 comes with a significant performance boost. Much of PHP's core was rewritten, which resulted in a noticeable difference. It's not uncommon to see your application run two or three times as fast, just by using PHP 7 or higher. Thanks to one of PHP's core values — maintaining backward compatibility — old PHP 5 codebases can easily be updated to benefit from these changes.

WHAT HAPPENED TO 6?

PHP 5.6 was the latest version in the five series, with the next one being 7.0. What happened to version 6? The core team started working on it, only to realise rather late that there were significant internal implementation issues. They decided to rework the engine once again, but "PHP 6" was already being written. To avoid confusion, they decided to skip version 6 and jump straight to PHP 7.

The story of PHP 6 has become folklore within the community. If you want to know the in-depth story, you can do a quick Google search.

Even though PHP 7.0 was such a milestone, we've since moved on from it as well. PHP 7.0 is already considered old: it was released in 2015 and didn't receive updates anymore five years later. Around the late 5.x releases, PHP adopted a strict release cycle: every year brings one new version. Most versions are actively supported for two years, followed by one year of additional security support. After three years, you should update, as the version you are using doesn't get security patches anymore.

Arguably, it's even better to follow along with the latest version. There will always be minor breaking changes and deprecations, but most code can easily be updated. There even are great automated tools that will take your existing codebase, detect any upgrading errors, and fix them automatically.

AUTOMATION

One of those automated tools is called Rector which has grown in popularity over the years: https://github.com/rectorphp/rector. Rector can update your codebase automatically across several PHP versions and is a great tool to know about if you ever have to deal with legacy projects.

In the next few chapters, we'll take a deep dive into the features added in PHP 7 and PHP 8. Before doing that, we'll start by looking at the smaller yet significant changes in this chapter.

TRAILING COMMAS

Support for trailing commas has been added incrementally up until PHP 8.0. They are now supported in arrays, function calls, parameter lists, and closure `use` statements.

Trailing commas are a somewhat controversial topic among developers. Some people like them; others hate them. One argument in favour of trailing commas is that they make diffs easier. For example, imagine an array with two elements:

```
$array = [
    $foo,
    $bar
];
```

Next, you'd need to add a third:

```
$array = [
    $foo,
    $bar,
    $baz
];
```

Version control systems such as GIT would list two changes instead of one because you did make two actual changes. If you'd always used trailing commas, you wouldn't need to alter the existing lines:

```
$array = [
    $foo,
    $bar,
    $baz,
];
```

Besides version control diffs, trailing commas can also be considered "easier" to reason about because you never have to worry about adding an extra comma or not. You might not prefer this writing style, and that's fine. We'll discuss the importance of a style guide in a later chapter.

FORMATTING NUMERIC VALUES

You can use an underscore to format numeric values. This underscore is completely ignored when the code is parsed but makes long numbers easier to read by humans.

This can be especially useful in tests. Take the following example: we're testing an invoice flow and we need to pass in an amount of money. It's a good idea to store money in cents to prevent rounding errors, so using an `_` makes it clearer:

```php
public function test_create_invoice()
{
    // $100 and 10 cents
    $invoice = new Invoice(100_10);

    // Assertions
}
```

ANONYMOUS CLASSES

Anonymous classes were added in PHP 7.0. They can be used to create objects on the fly; they can even extend a base class. Here's an example in a test context:

```php
public function test_dto_type_checking_with_arrays(): void
{
    $dto = new class(
        ['arrayOfInts' => [1, 2]]
    ) extends DataTransferObject {
        /** @var int[] */
        public array $arrayOfInts;
    };

    // Assertions
}
```

FLEXIBLE HEREDOC

Heredoc can be a useful tool for larger strings, though they had an annoying indentation quirk in the past: you couldn't have any whitespaces in front of the closing delimiter. In effect, that meant you had to do this:

```php
public function report(): void
{
    $query = <<<SQL
        SELECT *
        FROM `table`
        WHERE `column` = true;
SQL;

    // …
}
```

Fortunately, you can now do this:

```php
public function report(): void
{
    $query = <<<SQL
        SELECT *
        FROM `table`
        WHERE `column` = true;
    SQL;

    // …
}
```

The whitespace in front of the closing marker will be ignored on all lines.

EXCEPTION IMPROVEMENTS

Let's take a look at two smaller additions in PHP 8. First: a throw statement is now an expression. That means you can use it in more places, such as the null coalescing righthand operand, or in short closures; we'll cover both of those features in-depth later.

```
// Invalid before PHP 8
$name = $input['name'] ?? throw new NameNotFound();

// Valid as of PHP 8
$name = $input['name'] ?? throw new NameNotFound();

$error = fn (string $class) ⇒ throw new $class();
```

Second: exceptions support non-capturing catches. There might be cases where you want to catch an exception to keep the program flow going and not do anything with it. You always had to assign a variable to the caught exception, even when you didn't use it. That's not necessary anymore:

```
try {
    // Something goes wrong
} catch (Exception) {
    // Just continue
}
```

Weak references and maps

PHP 7.4 added the concept of weak references. To understand what they are, you need to know a little bit about garbage collection. Whenever an object is created, it requires some memory to keep track of its state. If an object is created and assigned to a variable, then that variable is a *reference* to the object. As soon as there are no references to an object anymore, it cannot be used, and it doesn't make sense to keep it in memory. That's why the garbage collector sometimes comes along looking for those kinds of objects and removes them, freeing memory.

Weak references are references to objects which don't prevent them from being garbage collected. This feature on its own might seem a little strange, but they allow an interesting use case as of PHP 8, combined with weak maps: a map of objects which are referenced using weak references.

Take the example of ORMs: they often implement caches which hold references to entity classes to improve the performance of relations between entities. These entity objects can not be garbage collected, as long as this cache references them, even if the cache is the only thing referencing them.

If this caching layer uses weak references and maps instead, PHP will garbage collect these objects when nothing else references them anymore. Especially in the case of ORMs, which can manage several hundred, if not thousands of entities within a request; weak maps can offer a better, more resource-friendly way of dealing with these objects.

Here's what weak maps look like:

```
class EntityCache
{
    public function __construct(
        private WeakMap $cache
    ) {}

    public function getSomethingWithCaching(object $object): object
    {
        if (! isset($this->cache[$object])) {
            $this->cache[$object] = $this
                ->computeSomethingExpensive($object);
        }

        return $this->cache[$object];
    }

    // …
}
```

In this example, we cache the result of an expensive operation related to an object in a cache. If the result doesn't exist yet, we'll compute it once. Thanks to weak maps, this object can still be garbage collected if there are no other references anymore.

THE JIT

The JIT — just in time — compiler is a core mechanism added in PHP 8 which can significantly speed up PHP code.

A JIT compiler will look at code while running and tries to find pieces of that code that are executed often. The compiler will take those parts and compile them to machine code — on the fly — and use the optimised machine code from then on out. The technique can be used in interpreted languages like JavaScript, and now also in PHP. It has the potential to improve an application's performance significantly.

I say "potential" because there are a few caveats we must take into account. I'll keep those and everything else JIT-related for the chapter dedicated to the JIT.

CLASS SHORTHAND ON OBJECTS

The `::class` shorthand was added in PHP 5.5 to quickly get the full class name of an imported one:

```
$className = Offer::class;
```

In PHP 8 this also works with variables:

```
$className = $offer::class;
```

IMPROVED STRING TO NUMBER COMPARISONS

One of PHP's dynamic type system strengths is one of its weaknesses at the same time: type juggling. PHP will try to change a variable's type when it encounters a strange piece of code. Here's an example where strings and numbers are compared:

```
'1' == 1 // true
```

If you don't want this type juggling, you can use the triple `===` sign instead:

```
'1' === 1 // false
```

There are strange edge cases when using weak comparisons, though. The following returns true in versions before PHP 8:

```
'foo' == 0 // true
```

As of PHP 8, this behaviour is improved. The example above, as well as other edge cases, would now return false correctly.

```
'foo' == 0 // false
```

CHANGES TO THE ERROR CONTROL OPERATOR

One last notable change is that the `@` operator no longer silences fatal errors. You can still use it to silence other kinds of errors, like so:

```
$file = @file_get_contents('not/an/existing/path');
```

In cases where you don't use the `@` operator, an error would be triggered, but with `@`, `$file` will simply be false. If, on the other hand, a fatal error is thrown in PHP 8, the `@` operator wouldn't ignore it anymore.

DEEP DOWN IN THE CORE

There are two low-level components added to PHP's core with the 7.4 update: FFI and preloading. These are two complex topics on their own, each with their own chapter. I'll briefly mention them here, though.

FFI — a.k.a. foreign function interface — allows PHP to speak to shared libraries written in, for example C, directly. To put that in other words: it's possible to write PHP extensions and ship them as composer packages without installing the low-level PHP extensions themselves. I promise we'll dive deeper into this soon!

The other one — preloading — enables you to compile PHP code on server startup. That code will be kept in memory for all subsequent requests. It can speed up your average PHP web framework since it doesn't have to boot anymore on every request. Again: this one deserves a chapter on its own.

So with these random little features out of the way, let's dig deeper into all the great things added to PHP over the past years!

CHAPTER 03

PHP'S TYPE SYSTEM

One of the most significant features of PHP 7, besides performance, is its improved type system. Granted: it took until PHP 8 before most key features were implemented, but overall, PHP's type system has improved significantly over the years. With its maturing type system, some community projects started to use types to their full extent. Static analysers were built, which opened the doors for a whole other way of programming. We'll dig deeper into the benefits of those static analysers in the next chapter. For now, we'll focus on what exactly changed within PHP's type system between version 5 and 8.

First of all, more built-in types were added: so called "scalar" types. These types are integers, strings, booleans and floats:

```
public function formatMoney(int $money): string
{
    // …
}
```

Next (you might have noticed this in the previous example already) return types were added. Before PHP 7 you were already able to use types but only for input parameters. This led to a mix of doc block types and inline types — some might call it a mess:

```php
/**
 * @param \App\Offer $offer
 * @param bool $sendMail
 *
 * @return \App\Offer
 */
public function createOffer(Offer $offer, $sendMail)
{
    // …
}
```

Some developers who didn't want to deal with this kind of messy code chose not to use types at all. After all, there was only a limited level of "type safety" since doc block types aren't interpreted at all. Your IDE, on the other hand, would be able to understand the doc block version, but for some people, this wasn't enough of a benefit. As of PHP 7.0, however, the previous example could be rewritten like so:

```php
public function createOffer(Offer $offer, bool $sendMail): Offer
{
    // …
}
```

It's the same information, but much more concise and actually checked by the interpreter — we'll dive into the benefits of a type system and type safety later.

Besides parameter and return types, there are now also typed properties for classes. Just like parameter and return types, they are opt-in; PHP won't do any type checks.

```php
class Offer
{
    public string $offerNumber;

    public Money $totalPrice;
}
```

You can see both scalar types and objects are allowed, just like with parameter and return types.

Now, typed properties have an interesting feature to them. Take a look at the following example and notice that despite what you might think at first sight, it is valid and won't throw any errors:

```php
class Money
{
    public int $amount;
}

$money = new Money();
```

In this example, `Money` doesn't have a constructor, and no value is set in its `$amount` property. Even though the value of `$amount` isn't an integer after constructing the

`Money` object, PHP will only throw an error when we're trying to access the property, later in the code:

```
var_dump($money->amount);

// Fatal error: Typed property Money::$amount
// must not be accessed before initialization
```

As you can see in the error message, there's a new kind of variable state: uninitialized. If `$amount` didn't have a type, its value would simply be `null`. Typed properties can be nullable, so it's important to make a distinction between a type that was forgotten and a type that's `null`. That's why "uninitialized" was added.

There are a few essential things to remember about uninitialized:

- As we just saw, you cannot read from uninitialized properties; doing so will result in a fatal error.
- Because the uninitialized state is checked when accessing a property, you're able to create an object with an uninitialized property.
- You're allowed to write to an uninitialized property before reading it, which means you could do `$money->amount = 1` after constructing it.
- Using `unset` on a typed property will make it uninitialized, while unsetting an untyped property will make it `null`.
- While the uninitialized state is only checked when reading a property's value, type validation is done when writing to it. This means that you can be sure no invalid type will ever end up as a property's value.

I realise this behaviour might not be what you'd expect a programming language to do. I've mentioned before that PHP tries very hard not to break backwards compatibility with their updates, which sometimes results in compromises like the uninitialized state. While I think that a variable's state should be checked immediately after constructing an object, we'll have to make do with what we have. At least it's another good reason to make use of static analysis, which we'll discuss in the next chapter.

DEALING WITH NULL

While we're on the topic of uninitialized state, let's also discuss `null`. Some have called the concept of `null` a "billion-dollar mistake", arguing it allows for a range of edge cases that we have to consider when writing code. It might seem strange to work in a programming language that doesn't support `null`, but there are useful patterns to replace it, and get rid of its pitfalls.

Let's illustrate those downsides first with an example. Here we have a `Date` value object with a timestamp variable, format function and static constructor called `now`.

```
class Date
{
    public int $timestamp;

    public static function now(): self { /* … */ }

    public function format(): string { /* … */ }

    // …
}
```

Next, we have an invoice with a payment date:

```
class Invoice
{
    public ?Date $paymentDate = null;

    // …
}
```

The payment date is nullable because invoices can be pending and not yet have a payment date.

As a side note: take a look at the nullable notation; I've already mentioned it but haven't shown it until now. We've prefixed `Date` with a question mark, indicating that it can either be `Date` or `null`. We've also added a default `= null` value, ensuring the value is never uninitialized to prevent all those runtime errors you might encounter.

Back to our example: what if we want to do something with our payment date's timestamp?

```
$invoice->paymentDate->timestamp;
```

Since we're not sure `$invoice->paymentDate` is a `Date` or `null`, we risk running into runtime errors:

```
// Trying to get property 'timestamp' of non-object
```

Before PHP 7.0, you'd use `isset` to prevent those kinds of errors:

```php
isset($invoice->paymentDate)
    ? $invoice->paymentDate->timestamp
    : null;
```

That's rather verbose, though, and is why a new operator was introduced: the null coalescing operator.

```php
$invoice->paymentDate->timestamp ?? null;
```

This operator will automatically perform an `isset` check on its left-hand operand. If that returns false, it will return the fallback provided by its righthand operand. In this case, the payment date's timestamp or null. It is a nice addition that reduces the complexity of our code.

PHP 7.4 added another null coalescing shorthand: the null coalescing assignment operator. This one not only supports the default value fallback, but will also write it directly to the left-hand operand. It looks like this:

```php
$temporaryPaymentDate = $invoice->paymentDate ??= Date::now();
```

So if the payment date is already set, we'll use that one in `$temporaryPaymentDate`, otherwise we'll use `Date::now()` as the fallback for `$temporaryPaymentDate` and also write it to `$invoice->paymentDate` immediately.

A more common use case for the null coalescing assignment operator is a memoization function: a function that stores the result once it's calculated.

This function will perform a regex match on a string with a pattern, but if the same string and same pattern are provided, it will simply return the cached result.

```php
function match_pattern(string $input, string $pattern) {
    static $cache = [];

    return $cache[$input][$pattern] ??=
        (function (string $input, string $pattern) {
            preg_match($pattern, $input, $matches);

            return $matches[0];
        })($input, $pattern);
}
```

Before we had the null coalescing operator assignment, we'd need to write it like so:

```php
function match_pattern(string $input, string $pattern) {
    static $cache = [];

    $key = $input . $pattern;

    if (! isset($cache[$key])) {
        $cache[$key] = (function (string $input, string $pattern) {
            preg_match($pattern, $input, $matches);

            return $matches[0];
        })($input, $pattern);
    }

    return $cache[$key];
}
```

There's one more null-oriented feature in PHP added in PHP 8: the nullsafe operator. Take a look at this example:

```
$invoice->paymentDate->format();
```

What happens if our payment date is null? You'd again get an error:

```
// Call to a member function format() on null
```

Your first thought might be to use the null coalescing operator, but that wouldn't work:

```
$invoice->paymentDate->format('Y-m-d') ?? null;
```

The null coalescing operator doesn't work with method calls on `null`. So before PHP 8, you'd need to do this:

```
$paymentDate = $invoice->paymentDate;

$paymentDate ? $paymentDate->format('Y-m-d') : null;
```

Fortunately there's the nullsafe operator which will only perform method calls when possible and otherwise return `null` instead:

```
$invoice->getPaymentDate()?->format('Y-m-d');
```

DEALING WITH NULL — THERE'S ANOTHER WAY

I started this section saying null is called a "billion-dollar mistake", but next, I showed you three ways PHP is embracing null with fancy syntax. The reality is that null is a frequent occurrence in PHP, and it's a good thing we have syntax to deal with it in a sane way. However, it's also good to look at alternatives to using null altogether. One such alternative is the null object pattern.

Instead of one `Invoice` class that manages internal state about whether it's paid or not; let's have two classes: `PendingInvoice` and `PaidInvoice`. The `PendingInvoice` implementation looks like this:

```php
class PendingInvoice implements Invoice
{
    public function getPaymentDate(): UnknownDate
    {
        return new UnknownDate();
    }
}
```

`PaidInvoice` looks like this:

```
class PaidInvoice implements Invoice
{
    // …

    public function getPaymentDate(): Date
    {
        return $this->date;
    }
}
```

Next, there's an `Invoice` interface:

```
interface Invoice
{
    public function getPaymentDate(): Date;
}
```

Finally, here are the two date classes:

```
class Date
{
    // …
}
```

```
class UnknownDate extends Date
{
    public function format(): string
    {
        return '/';
    }
}
```

The null object pattern aims to replace `null` with actual objects, objects that behave differently because they represent the "absence" of the real object. Another benefit of using this pattern is that classes become more representative of the real world: instead of a "date or null", it's a "date or unknown date", instead of an "invoice with a state" it's a "paid invoice or pending invoice". You wouldn't need to worry about `null` anymore.

```
$invoice->getPaymentDate()->format(); // A date or '/'
```

You might not like this pattern, but it's important to know that the problem can be solved this way.

CHANGING TYPES

Something happened in the above code samples that I wrote down as a fact, yet it's a significant addition to PHP's type system. We were able to change method signatures during inheritance. Take a look at the `Invoice` interface:

```
interface Invoice
{
    public function getPaymentDate(): Date;
}
```

It declares the return type of `getPaymentDate` as `Date`, yet we changed it in our `PendingInvoice` to `UnknownDate`:

```
class PendingInvoice implements Invoice
{
    public function getPaymentDate(): UnknownDate
    {
        /* … */
    }
}
```

This powerful technique is called type variance; it's supported as of PHP 7.4. It's so powerful (and a little complex) that we'll spend a whole chapter on the topic later in this book. For now, all you need to know is that return types and input parameter types are allowed to change during inheritance, but both have different rules to follow.

Another interesting detail has to do with nullable types. There's a difference between a nullable type using `?` and a default `= null` value; you've seen them used together already in a previous example. The difference is this: if you make a type nullable in

PHP, you're still expected to pass something to that function; you can't just skip the parameter:

```php
function createOrUpdate(?Offer $offer): void
{
    // …
}

createOrUpdate();

// Uncaught ArgumentCountError:
//     Too few arguments to function createOrUpdate(),
//     0 passed and exactly 1 expected
```

So by adding an explicit `= null` default value, you're allowed to omit the value altogether:

```php
function createOrUpdate(?Offer $offer = null): void
{
    // …
}

createOrUpdate();
```

Unfortunately, because of backwards compatibility, there's a little quirk with this system. If you assign a default `= null` value to a variable, it'll always be nullable.

Regardless of whether the type is explicitly made nullable or not, so this will be allowed:

```
function createOrUpdate(Offer $offer = null): void
{
    // …
}

createOrUpdate(null);
```

UNION AND OTHER TYPES

There are still a few more things worth mentioning in this chapter, and the most exciting one is union types. They allow you to type hint a variable with several types. Note: the input needs to be one of those declared types. Here's an example:

```
interface Repository
{
    public function find(int|string $id);
}
```

Be careful not to overuse union types, though. Too many types in the same union can indicate that this function is trying to do too much at once.

For example: a framework might allow you to return both a `Response` and `View` object from controller methods. Sometimes it's convenient to directly return a `View`, while other times you want to have fine-grained control over the response. These are cases where a union type on `Response|View` is fine. On the other hand, if you have a function that accepts a union of `array|Collection|string`, it's probably an indication that the function has to do too many things. It's best to think about finding a common ground in those cases; maybe only accept `Collection` or `array`.

Finally, there are three more built-in types available.

There's the `static` return type available in PHP 8. It indicates that a function returns the class where that function was called from. It differs from `self` since that one always refers to the parent class, while `static` can also indicate the child class:

```php
abstract class Parent
{
    public static function make(): static { /* … */ }
}

class Child extends Parent { /* … */ }

$child = Child::make();
```

In this example, static analysis tools and IDEs now know that `$child` is an instance of `Child`. If `self` was used as a return type, they would think it was an instance of `Parent`. Also note that the `static` keyword and `static` return type are two different concepts; they just happen to have the same name (which is the case in many other languages).

There's also the `void` return type added in PHP 7.1. It checks whether nothing is returned from a function. Note that `void` cannot be combined into a union.

Finally, there's the `mixed` type, also available as of PHP 8. mixed
can be used to type hint "anything". It's a shorthand for the union of `array|bool|callable|int|float|null|object|resource|string`. Think of `mixed` as the opposite of `void`.

Generics and enums

There are still two major features missing in PHP's type system today: generics and enums. I wish I could write about them in this book, but unfortunately, they aren't supported by the language yet.

There have been efforts in the past to add both features, but there hasn't been any consensus about how enums should be implemented, and generics would have too large an impact on runtime performance.

This raises an interesting question: why do we need to check types at runtime? Isn't that too late? If something goes wrong, the program crashes anyway. That's exactly the topic we'll look at in-depth in the next chapter.

CHAPTER 04

STATIC ANALYSIS

Having spent a whole chapter on PHP's type system, I realise I haven't discussed *why* you want to use it. I realise a significant part of the community doesn't like using PHP's type system, so it's important to discuss both the pros and cons thoroughly. That's what we'll do in this chapter. We'll start by discussing the value provided by a type system.

Many think of programming languages with a stricter type system to have fewer or no runtime errors. There's the popular saying that a strong type system prevents bugs, but that's not entirely true. You can easily write bugs in a strongly typed language, but a range of bugs is not possible anymore since a good type system prevents them.

Are you unsure what the difference is between strong, weak, static and dynamic type systems? We'll cover the topic in-depth later in this chapter!

Imagine a simple function: `rgbToHex`. It takes three arguments, each of which are integers between 0 and 255. The function then converts the integers to a hexadecimal string. Here's what this function's definition might look like without using types:

```
function rgbToHex($red, $green, $blue) {
    // …
}
```

Since we want to ensure our implementation of this function is correct, we write tests:

```
assert(rgbToHex(0, 0, 0) === '000000');

assert(rgbToHex(255, 255, 255) === 'ffffff');

assert(rgbToHex(238, 66, 244) === 'ee42f4');
```

These tests ensure that everything works as expected. Right?

Well... we're only testing three out of the 16,777,216 possible RGB combinations. But logic reasoning tells us that if these three sample cases work, we're probably safe. What happens though if we pass floats instead of integers?

```
rgbToHex(1.5, 20.2, 100.1);
```

Or numbers outside of the allowed range?

```
rgbToHex(-504, 305, -59);
```

What about `null`?

```
rgbToHex(null, null, null);
```

Or strings?

```
rgbToHex('red', 'green', 'blue');
```

Or the wrong number of arguments?

```
rgbToHex();
rgbToHex(1, 2);
rgbToHex(1, 2, 3, 4);
```

Or a combination of the above?

I can easily think of five edge-cases we need to test before there's relative certainty that our program does what it needs to do. That's at least eight tests we need to write, and I'm sure you can come up with a few others. These are the kinds of problems a type system aims to partially solve, and note the word *partially* because we'll come back to it. If we filter the input by a specific type, many of our tests become obsolete. Say we'd only allow integers:

```
function rgbToHex(int $red, int $green, int $blue)
{
    // …
}
```

You can think of types as a subcategory of all available input; it's a filter that only allows specific items.

Let's take a look at the tests that aren't necessary anymore thanks to our `int` type hint:

- Whether the input is numeric
- Whether the input is a whole number
- Whether the input isn't `null`

We still need to check whether the input number is between 0 and 255. At this point, we run against the limitations of many type systems, including PHP's. Sure we can use `int`, though in many cases, the category described by this type is still too large for our business logic: `int` would also allow `-100` to be passed, which wouldn't make any sense for our function. Some languages have a `uint` or "unsigned integer" type; yet it is also too large a subset of "numeric data".

Luckily, there are ways to address this issue.

One approach could be to use "configurable" or generic types, for example `int<min, max>`. The concept of generics is known in many programming languages, though unfortunately not in PHP. In theory, a type could be preconfigured so that it is smart enough to know about all your business logic.

Languages like PHP lack the flexibility of generic types, but we do have classes, which can be used as types themselves. We could, for example, represent a configurable "integer" with a class `IntWithinRange`:

```php
class IntWithinRange
{
    private int $value;

    public function __construct(int $min, int $max, int $value)
    {
        if ($value <= $min || $value >= $max) {
            throw new InvalidArgumentException('…');
        }

        $this->value = $value;
    }

    // …
}
```

So whenever we're using an instance of `IntWithinRange`, we can ensure its value is constrained within a subset of integers. But this only works when constructing `IntWithinRange`. In practice, we can't ensure its minimum and maximum value in our `rgbToHex` function, meaning we can't say we only accept `IntWithinRange` object with a minimum of `0` and a maximum of `255`. Therefore we can only say we accept any object with the type of `IntWithinRange`:

```
function rgbToHex(
    IntWithinRange $red,
    IntWithinRange $green,
    IntWithinRange $blue
) {
    // …
}
```

To solve this, we need an even more specific type: `RgbValue`:

```
class RgbValue extends IntWithinRange
{
    public function __construct(int $value)
    {
        parent::__construct(0, 255, $value);
    }
}
```

We've arrived at a working solution. By using `RgbValue`, most of our tests become redundant. We now only need one test to test the business logic: "given three RGB-valid colors, does this function return the correct HEX value?" — a great improvement!

```
function rgbToHex(RgbValue $red, RgbValue $green, RgbValue $blue)
{
    // …
}
```

BUT HOLD ON...

If one of the claimed benefits of type systems is to prevent runtime bugs and errors, then we're still not getting anywhere with our `RgbValue`. PHP will check this type at runtime and throw a type error when the program is running. In other words: things can still go horribly wrong at runtime, maybe even in production. This is where static analysis comes in.

Instead of relying on runtime type checks (and throwing errors to handle them), static analysis tools will test your code without running it. If you're using any kind of IDE, you're already making use of it. When your IDE tells you what methods are available on an object, what input a function requires, or whether you're using an unknown variable, it's all thanks to static analysis.

Granted, runtime errors still have their merit: they stop code from further executing when a type error occurs, so they probably are preventing actual bugs. They also provide useful information to the developer about what exactly went wrong and where. But still, the program crashed. Catching the error before running code in production will always be the better solution.

Some other programming languages even go as far as to include a static analyser in their compiler: if static analysis checks fail, the program won't compile. Since PHP doesn't have a standalone compiler, we'll need to rely on external tools to help us.

PHP COMPILER

Even though PHP is an interpreted language, it still has a compiler. PHP code is compiled on the fly when, for example, a request comes in. There are of course, caching mechanisms in play to optimise this process, but there's no standalone compilation phase.

This allows you to easily write PHP code and immediately refresh the page without having to wait for a program to compile, one of the well-known strengths of PHP development.

Luckily there are great community-driven static analysers for PHP available. They are standalone tools that look at your code and all its type hints allowing you to discover errors without ever running the code. These tools will not only look at PHP's types, but also at doc blocks, meaning they allow for more flexibility than normal PHP types would.

Take a look at how Psalm would analyse your code and report errors:

```
Analyzing files...

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□   60 / 1038 (5%)
…
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ 1038 / 1038 (100%)

ERROR: TooFewArguments
    …
    Too few arguments for method …\PriceCalculatorFactory::withproduct


ERROR: TooFewArguments
    …
    Too few arguments for method …\Checkable::ischeckable

------------------------------
2 errors found
------------------------------
```

Here we see Psalm scanning over a thousand source files and detecting where we forgot to pass the correct amount of arguments to a function. It does so by analysing method signatures and comparing them to how those methods are called. Of course, type definitions play an important role in this process.

Most static analysers even allow custom doc block annotations that support, for example, generics. By doing so they can do much more complex type checks than PHP could do at runtime. Even when running the code wouldn't perform any checks, the static analyser could tell you when something is wrong beforehand. Such static type checks could be done locally when writing code, built into your CI pipeline, or a mix of both.

In fact, the static analysis community is getting so much traction these days that PhpStorm — the most popular IDE for writing PHP code — added built-in support for them. This means that the result of several type checks performed by your static analyser can be shown immediately when writing code.

Tools like Psalm, PHPStan, and Phan are great, but they also lack the eloquence you'd get from built-in language support. I've gone on and on about removing doc blocks in favour of built-in types in the previous chapter, and now we're adding them again to support static analysis. Now, to be clear: these tools also work with PHP's built-in type system (without using any doc blocks), but those doc block annotations offer a lot more functionality, because PHP's syntax doesn't limit them; they are comments, after all.

On the other hand (I've said this before in the previous chapter), there's little chance that features like generics will be added anytime soon in PHP itself since they pose such a threat to runtime performance. So if there's nothing better, we'll have to settle with doc blocks anyway if we want to use static analysis to its full extent.

If only… Can you see where I'm going with this?

What if PHP supported the generic syntax, but didn't interpret it at runtime? What if you'd *need* to use a static analyser to guarantee correctness (when using generics), and wouldn't worry about them when running your code. That's exactly the point of static analysis.

You might be afraid of PHP not enforcing those type checks at runtime. Still, you could also argue that static analysers are way more advanced in their capabilities, exactly because they aren't run when executing code. I don't think it's such a convoluted idea at all, and in fact, other languages already use this approach. Think about TypeScript, which has grown in popularity tremendously over the years. It's compiled to JavaScript, and all its type checks are done during that compilation phase, without running the code. Now I'm not saying we need another language that compiles to PHP; I'm only saying that static analysers are very powerful tools. If you decide to embrace them, you'll notice how you can reduce the number of tests and how rarely runtime type errors occur.

Where does that leave us now? Unfortunately, not very far. I'd recommend using a static analyser in your projects, regardless of whether you want to use its advanced annotations or not. Even without those, static analysers offer a great benefit. You've got much more certainty strange edge cases aren't possible, and you need to write fewer tests, all without ever running that code once. It's a great tool to have in your toolbox, and maybe one day, we'll see PHP fully embrace its benefits.

IN ACTION

Ready to see what static analysis can do for you? I'd recommend to go to psalm.dev and play around with their interactive playground. Some great examples show the full power of static analysis.

CHAPTER 05

PROPERTY PROMOTION

Having spent two chapters on the topic of PHP's type system, it's time to look at some other features in-depth. In this chapter, we'll look at an addition to PHP's syntax that gets rid of much unnecessary boilerplate code.

You might have noticed I prefer to use value objects and data transfer objects whenever possible. I like to work with simple objects only containing data and pass them around to be used within complex processes. In a later chapter, I'll share more on this view of object-oriented code. These kinds of data-focussed objects come with lots of boilerplate code, though. Here's an example of a customer data transfer object:

```
class CustomerDTO
{
    public string $name;

    public string $email;

    public DateTimeImmutable $birth_date;

    public function __construct(
        string $name,
        string $email,
        DateTimeImmutable $birth_date
    ) {
        $this->name = $name;
        $this->email = $email;
        $this->birth_date = $birth_date;
    }
}
```

In traditional PHP before PHP 8, you'd need to write each property's name four times. Thanks to constructor property promotion, you can rewrite the above like this:

```php
class CustomerDTO
{
    public function __construct(
        public string $name,
        public string $email,
        public DateTimeImmutable $birth_date,
    ) {}
}
```

That's quite a difference! Let's look at this feature in-depth.

HOW IT WORKS

The basic idea is simple: ditch the class properties and the variable assignments, and prefix constructor parameters with public, protected or private. PHP will take that new syntax, and transform it to normal syntax under the hood, before executing the code.

So when you write this code:

```
class Person
{
    public function __construct(
        public string $name = 'Brent',
    ) {
        // …
    }
}
```

PHP will transform it under the hood to this:

```
class Person
{
    public string $name;

    public function __construct(
        string $name = 'Brent'
    ) {
        $this->name = $name;

        // …
    }
}
```

And only executes it afterwards.

This code transformation is probably known under a more common name, at least if you're somewhat familiar with the JavaScript community: transpiling. That's right: PHP will transpile itself at runtime (and cache the results for better performance). That's an interesting thought, given the previous chapter on static analysis and how I shared the language's vision of adding features that are only used during static analysis.

Let's look further at what you can and cannot do with promoted properties.

Only in constructors

Promoted properties can only be used in constructors. That might seem obvious, but I thought it was worth mentioning, just to be clear.

Combining promoted and normal properties

Not all constructor properties must be promoted - you can also mix and match:

```
class MyClass
{
    public string $b;

    public function __construct(
        public string $a,
        string $b,
    ) {
        $this->b = $b;
    }
}
```

Be careful mixing the syntaxes because it can make your code less clear. Consider using a normal constructor if you're mixing promoted and non-promoted properties.

NO DUPLICATES

You're not able to declare a class property and a promoted property with the same name. That's rather logical since the promoted property is transpiled to a class property at runtime:

```php
class MyClass
{
    public string $a;

    public function __construct(
        public string $a,
    ) {}
}
```

UNTYPED PROPERTIES

You're allowed to promote untyped properties, though as I've argued in the previous chapters, you're better off using types wherever possible:

```php
class MyDTO
{
    public function __construct(
        public $untyped,
    ) {}
}
```

SIMPLE DEFAULTS

Promoted properties can have default values, but expressions like `new …` are not allowed:

```
public function __construct(
    public string $name = 'Brent',
    public DateTimeImmutable $date = new DateTimeImmutable(),
) {}
```

This makes sense since you're also not able to have such complex defaults with normal class properties.

WITHIN THE CONSTRUCTOR BODY

You're allowed to read the promoted properties in the constructor body. This can be useful if you want to do extra validation checks. You can use the local variable and the instance variable as both work fine:

```
public function __construct(
    public int $a,
    public int $b,
) {
    assert($this->a >= 100);

    if ($b >= 0) {
        throw new InvalidArgumentException('…');
    }
}
```

DOC BLOCKS

You can add doc blocks to promoted properties:

```php
class MyClass
{
    public function __construct(
        /** @var string */
        public $a,
    ) {}
}

$property = new ReflectionProperty(MyClass::class, 'a');

$property->getDocComment(); // "/** @var string */"
```

ATTRIBUTES

Attributes are the topic of an upcoming chapter, so consider this a sneak-peek: they are allowed on promoted properties. When transpiled, they will be present both on the constructor parameter, as well as the class property:

```php
class MyClass
{
    public function __construct(
        #[MyAttribute]
        public $a,
    ) {}
}
```

Will be transpiled to:

```
class MyClass
{
    #[MyAttribute]
    public $a;

    public function __construct(
        #[MyAttribute] $a,
    ) {
        $this->a = $a;
    }
}
```

NOT ALLOWED IN ABSTRACT CONSTRUCTORS

I must admit I didn't know abstract constructors were a thing, and I've never used them. Moreover, promoted properties are not allowed in them:

```
abstract class A
{
    abstract public function __construct(
        public string $a,
    ) {}
}
```

ALLOWED IN TRAITS

Promoted properties *are* allowed in traits. It makes sense to support promoted properties in traits since the transpiled code would result in a valid trait. Whether it's a good thing or not to have constructors in traits is another question.

```
trait MyTrait
{
    public function __construct(
        public string $a,
    ) {}
}
```

VAR IS NOT SUPPORTED

Old – I mean, experienced – PHP developers might have used `var` in the distant past to declare class variables but it's not allowed with constructor promotion. Only `public`, `protected` and `private` are valid keywords.

```
public function __construct(
    var string $a,
) {}
```

VARIADIC PARAMETERS CANNOT BE PROMOTED

Since you can't convert to a type that's `array of type`, it's not possible to promote variadic parameters.

```
public function __construct(
    public string ...$a,
) {}
```

DID YOU KNOW

Variadic functions make use of the rest operator `...`, a feature added in PHP 5.6. It allows you to define a function input variable which will take all the "rest" of the variables and combine them into an array. In other words: a shorthand for `func_get_args()`. We'll cover variadic functions in chapter 9.

REFLECTION FOR ISPROMOTED

Both `ReflectionProperty` and `ReflectionParameter` have a new `isPromoted()` method to check whether the class property or method parameter is promoted.

INHERITANCE

Since PHP constructors don't need to follow their parent constructor's declaration, there's little to be said: inheritance is allowed. If you need to pass properties from the child constructor to the parent constructor, you'll need to pass them manually:

```
class A
{
    public function __construct(
        public $a,
    ) {}
}

class B extends A
{
    public function __construct(
        $a,
        public $b,
    ) {
        parent::__construct($a);
    }
}
```

That's about all there is to say about property promotion. I was hesitant to use them at first, but once I gave it a try, I quickly got used to them. I must admit: promoted properties are probably my favourite feature of PHP 8.

CHAPTER 06

NAMED ARGUMENTS

Just like constructor property promotion, named arguments are a new syntactical addition in PHP 8. They allow you to pass variables to a function based on the argument name in that function, instead of their position in the argument list. Here's an example with a built-in PHP function:

```
setcookie(
    name: 'test',
    expires: time() + 60 * 60 * 2,
    secure: true,
);
```

And here's it used when constructing a DTO:

```
class CustomerData
{
    public function __construct(
        public string $name,
        public string $email,
        public int $age,
    ) {}
}

$data = new CustomerData(
    age: $input['age'],
    email: $input['email'],
    name: $input['name'],
);
```

Named arguments are a great new feature and will have a significant impact on my day-to-day programming life. You're probably wondering about the details. What if you pass a wrong name, or what about combining ordered and named arguments? Well, let's look at all those questions in-depth.

WHY NAMED ARGUMENTS?

This feature was a highly debated one. There were some concerns about adding them, especially with regards to backwards compatibility. If you're maintaining a package and decide to change an argument's name, this now counts as a breaking change. Take for example this method signature, provided by a package:

```php
public function toMediaCollection(
    string $collection,
    string $disk = null
): void { /* … */ }
```

If the users of this package would use named arguments, they write something like this:

```php
$media->toMediaCollection(
    collection: 'default',
    disk: 'aws',
);
```

Now imagine you, the package maintainer, want to change the name of `$collection` to `$collectionName`. This means the code written by your users would break.

I agree this is a problem in theory, but being an open source maintainer myself, I know from experience that such variable name changes very rarely occur. The only times I can remember that we did such a change was because we were working on a new major release anyway, where breaking changes are allowed.

While I recognise the theoretical problem, I firmly believe this is a non-issue in practice. Such name changes rarely happen. And even if you really wouldn't want to be bothered with managing variable names as an open source maintainer, you could still add a warning in your package's README file. It could tell your users that variable names might change, and they should use named arguments at their own risk. I prefer just to keep it in mind, and remember that I should be careful changing variable names in the future. No big deal.

Despite this minor inconvenience, I'd say the benefits of named arguments are far more significant. The way I see it: named arguments will allow us to write cleaner and more flexible code.

For one, named arguments allow us to skip default values. Take a look again at the cookie example:

```
setcookie(
    name: 'test',
    expires: time() + 60 * 60 * 2,
    secure: true,
);
```

Its method signature is actually the following:

```
setcookie(
    string $name,
    string $value = '',
    int $expires = 0,
    string $path = '',
    string $domain = '',
    bool $secure = false,
    bool $httponly = false,
) : bool
```

In the example with named arguments, we didn't need to set a cookie `$value`, but we did need to set an expiration time. Named arguments made this method call a little more concise, because otherwise it would have looked like this:

```php
setcookie(
    'test',
    '',
    time() + 60 * 60 * 2,
    '',
    '',
    true
);
```

Besides skipping arguments with default values, there's also the benefit of clarifying which variable does what, something that's especially useful in functions with large method signatures. We could say that lots of arguments are usually a code smell; we still have to deal with them no matter what. So it's better to have a good way of doing so than nothing at all.

NAMED ARGUMENTS IN DEPTH

Let's look at what named arguments can and cannot do with the basics out of the way.

First of all, they can be combined with unnamed — also called ordered — arguments. In that case, the ordered arguments must always come first. Take our DTO example from before:

```php
class CustomerData
{
    public function __construct(
        public string $name,
        public string $email,
        public int $age,
    ) {}
}
```

You could construct it like so:

```php
$data = new CustomerData(
    $input['name'],
    age: $input['age'],
    email: $input['email'],
);
```

However, having an ordered argument after a named one would throw an error:

```php
$data = new CustomerData(
    age: $input['age'],
    $input['name'],
    email: $input['email'],
);
```

Next, it's possible to use array spreading in combination with named arguments:

```
$input = [
    'age' ⇒ 25,
    'name' ⇒ 'Brent',
    'email' ⇒ 'brent@spatie.be',
];

$data = new CustomerData(...$input);
```

HEADS UP!

Just like with variadic functions, arrays can be spread using the `...` operator. We'll take all arguments from the input array and spread them into a function. If the array contains keyed values, those key names will map onto named properties as well, and that is what's happening in the above example.

If, however, there are missing required entries in the array, or if there's a key not listed as a named argument, an error will be thrown:

```
$input = [
    'age' ⇒ 25,
    'name' ⇒ 'Brent',
    'email' ⇒ 'brent@spatie.be',
    'unknownProperty' ⇒ 'This is not allowed',
];

$data = new CustomerData(...$input);
```

It *is* possible to combine named and ordered arguments in an input array, but only if the ordered arguments follow the same rule as before: they must come first!

```
$input = [
    'Brent',
    'age' => 25,
    'email' => 'brent@spatie.be',
];

$data = new CustomerData(...$input);
```

Be careful mixing ordered and named arguments though. I personally don't think they improve readability at all.

VARIADIC FUNCTIONS

If you're using variadic functions, named arguments will be passed with their key name into the variadic arguments array. Take the following example:

```php
class CustomerData
{
    public static function new(...$args): self
    {
        return new self(...$args);
    }

    public function __construct(
        public string $name,
        public string $email,
        public int $age,
    ) {}
}

$data = CustomerData::new(
    email: 'brent@spatie.be',
    age: 25,
    name: 'Brent',
);

// [
//     'age' => 25,
//     'email' => 'brent@spatie.be',
//     'name' => 'Brent',
// ]
```

ATTRIBUTES

Here we are again with the mysterious attributes feature - we'll cover them in depth soon! For now, I can tell you that they also support named arguments when constructing them:

```
class ProductSubscriber
{
    #[ListensTo(event: ProductCreated::class)]
    public function onProductCreated(ProductCreated $event) { /* … */ }
}
```

OTHER THINGS WORTH NOTING

It's not possible to have a variable as the argument name:

```
$field = 'age';

$data = CustomerData::new(
    $field: 25,
);
```

And finally, named arguments will deal in a pragmatic way with name changes during inheritance. Take this example:

```php
interface EventListener {
    public function on($event, $handler);
}

class MyListener implements EventListener
{
    public function on($myEvent, $myHandler)
    {
        // …
    }
}
```

PHP will silently allow changing the name of `$event` to `$myEvent`, and `$handler` to `$myHandler`; *but* if you decide to use named arguments using the parent's name, it will result in a runtime error:

```php
public function register(EventListener $listener)
{
    $listener->on(
        event: $this->event,
        handler: $this->handler,
    );
}
```

This pragmatic approach was chosen to prevent a major breaking change where all inherited arguments would have to keep the same name. It seems like an excellent solution to me. You can expect to see named arguments used in this book here and there. I think they are great syntactical sugar.

CHAPTER 07

ATTRIBUTES

I've mentioned them twice before, and we've finally arrived at the topic: attributes. I remember doing my thesis back in college, and one of the central topics was "annotations in PHP". At that time there were custom annotations parsers, essentially parsing doc block strings on the fly and interpreting those as annotations. There was a big debate about whether annotations should or shouldn't be added to the core. It's great to see them finally arrive in PHP, albeit with a different name: attributes.

Attributes can be used to add meta data to your code: stuff that would otherwise end up in config files or other places. One of the best-known examples is routing in Symfony, where you could write this:

```php
class BlogController
{
    /**
     * @Route("/blog", name="blog_index")
     */
    public function index()
    {
        // ...
    }
}
```

With attributes in PHP 8, it would look like this:

```
class BlogController
{
    #[Route('/blog', name: 'blog_index')]
    public function index()
    {
        // ...
    }
}
```

WHAT ABOUT THE SYNTAX?

It took months before a final syntax for attributes was decided. Options went from `<<` and `>>` to `@@` and other exotic variations. The simplest option — `@` — wasn't possible to use because that's already the error suppression operator.

Finally, we settled on `#[ ]`, the same syntax used by Rust annotations. Its benefit is that it can group several attributes at once.

The advantage of attributes is that you don't have to parse a blob of text at runtime anymore; PHP has built-in reflection support for them. We'll cover all of it in depth in this chapter!

RUNDOWN

Let's look at another example of attributes in the wild:

```
use Support\Attributes\ListensTo;

class ProductSubscriber
{
    #[ListensTo(ProductCreated::class)]
    public function onProductCreated(ProductCreated $event) { /* … */ }

    #[ListensTo(ProductDeleted::class)]
    public function onProductDeleted(ProductDeleted $event) { /* … */ }
}
```

Here we see attributes being used for event listeners: whenever a method is marked with `ListensTo`. When registering event listeners, we could use this attribute to build a mapping of which methods handle which events.

REDUNDANT ATTRIBUTES?

When I showed the ListensTo example online, some people told me you could just look at the method's typed parameter to know what event it handles, and adding a dedicated attribute seemed redundant. For simple applications, I'd agree, but let's consider the following example.

Imagine a `LoggableEvent` interface that events may implement:

```php
interface LoggableEvent
{
    public function getEventName();
}
```

And a `MailLogEventSubscriber`:

```php
class MailLogEventSubscriber
{
    public function handleLoggableEvent(LoggableEvent $event)
    {
        // …
    }
}
```

Now imagine — because the business requires it — that *some* `LoggableEvent` objects should be handled by sending a log mail, but not all. If you're relying on the method signatures to determine what events they should handle, you will end up with something like this:

```php
class MailLogEventSubscriber
{
    public function handleOrderCreatedEvent(
        OrderCreatedEvent $event
    ): void {
        $this->actuallyHandleTheEvent($event);
    }

    public function handleInvoiceCreatedEvent(
        InvoiceCreatedEvent $event
    ): void {
        $this->actuallyHandleTheEvent($event);
    }
    public function handleInvoicePaidEvent(
        InvoicePaidEvent $event
    ): void {
        $this->actuallyHandleTheEvent($event);
    }

    private function actuallyHandleTheEvent(
        LoggableEvent $event
    ): void {
        // …
    }
}
```

Using attributes, however, would allow you to write something like this:

```
class MailLogEventSubscriber
{
    #[
        ListensTo(OrderCreatedEvent::class),
        ListensTo(InvoiceCreatedEvent::class),
        ListensTo(InvoicePaidEvent::class),
    ]
    public function handleLoggableEvent(LoggableEvent $event)
    {
        // …
    }
}
```

Now, if you're building an application with only a few dozen events, the simple approach is fine. If you're dealing with thousands of events, though, it becomes cumbersome. It's those cases where I prefer the explicit approach because it saves time in the end.

Back to our example. How would this `ListensTo` attribute work under the hood? First of all, custom attributes are simple classes, annotated with the `#[Attribute]` attribute. Here's what it would look like:

```
#[Attribute]
class ListensTo
{
    public function __construct(
        public string $event
    ) {}
}
```

That's it — pretty simple, right? Keep in mind the goal of attributes: they are meant to add meta data to classes and methods, nothing more.

We still need to read that meta data and register our subscribers somewhere. You would read attributes when you're instantiating the event bus, and in my Laravel projects, I would use a service provider to do so.

In Laravel, service providers are used to set up the application when booting. There can be multiple service providers, each doing their own thing. A common use case is container registration, a topic we'll cover in an upcoming chapter.

In this example, we'd read the attributes from certain files and register them as event subscribers.

Here's the boilerplate setup, to provide a little context:

```
class EventServiceProvider extends ServiceProvider
{
    // In real life scenarios,
    //  we'd automatically resolve and cache all subscribers
    //  instead of using a manual array.
    private array $subscribers = [
        ProductSubscriber::class,
    ];

    public function register(): void
    {
        // The event dispatcher is resolved from the container
        $eventDispatcher = $this->app->make(EventDispatcher::class);

        foreach ($this->subscribers as $subscriber) {
            // We'll resolve all listeners
            //  (methods with the ListensTo attribute)
            //  and add them to the dispatcher.
            foreach (
                $this->resolveListeners($subscriber)
                as [$event, $listener]
            ) {
                $eventDispatcher->listen($event, $listener);
            }
        }
    }
    private function resolveListeners(string $subscriberClass): array
    {
        // Return an array with [eventName => handler] items.
    }
}
```

You can see we're using the `as [$event, $listener]` syntax in the for-loop. This is called array destructuring and something we'll also cover in an upcoming chapter.

What's most interesting is the implementation of `resolveListeners`. Here it is:

```
private function resolveListeners(string $subscriberClass): array
{
    $reflectionClass = new ReflectionClass($subscriberClass);

    $listeners = [];

    foreach ($reflectionClass->getMethods() as $method) {
        $attributes = $method->getAttributes(ListensTo::class);

        foreach ($attributes as $attribute) {
            $listener = $attribute->newInstance();

            $listeners[] = [
                // The event that's configured on the attribute
                $listener->event,

                // The listener for this event
                [$subscriberClass, $method->getName()],
            ];
        }
    }

    return $listeners;
}
```

Using reflection, we can read the attributes from the class' methods, and instantiate our custom attribute classes with the `$attribute->newInstance()` call. This is an important detail: our attribute objects are only constructed when we call `newInstance()` on them and not when they are loaded; it doesn't happen magically beforehand. When an attribute is constructed it will take the parameters we've given it when writing `#[ListensTo(OrderCreatedEvent::class)]`, and pass them to the `ListensTo` constructor.

This means that, technically, you don't even need to construct the custom attribute. You could call `$attribute->getArguments()` directly. On the other hand, instantiating the class means you've got the constructor's flexibility to parse the input in whatever way you like.

Another thing worth mentioning is the use of `ReflectionMethod::getAttributes()` — the function that returns all attributes for a method. You can pass two arguments to it, to filter its output. In order to understand this filtering though, there's one more thing you need to know about attributes first: it's possible to add several attributes to the same method, class, property or constant. You could, for example, do this:

```php
#[
    Route(Http::POST, '/products/create'),
    Autowire,
]
class ProductsCreateController
{
    public function __invoke() { /* … */ }
}
```

I came up with the `Autowire` attribute just as an example. It indicates this class could be autowired by the dependency container — a topic we'll cover in-depth in a later chapter.

Say you're parsing controller routes and you're only interested in the `Route` attribute. In that case you can pass the `Route` class as a filter:

```
$attributes = $reflectionClass->getAttributes(Route::class);
```

There's a second parameter you can pass to `getAttributes()` which changes how the filtering is done. By default it'll only match attributes that exactly match the given class name. However, by using `ReflectionAttribute::IS_INSTANCEOF`, you're able to retrieve all attributes implementing a given interface. For example, say you're parsing container definitions that consist of several potential attributes:

```
$attributes = $reflectionClass->getAttributes(
    ContainerAttribute::class,
    ReflectionAttribute::IS_INSTANCEOF
);
```

If our `Autowire` attribute would implement the `ContainerAttribute` interface, only that one would be returned and not the `Route` attribute. It's a nice shorthand, built into the core.

ATTRIBUTES IN DEPTH

Now that you have an idea of how attributes work in practice, it's time for some more theory, making sure you understand them thoroughly. First of all something I mentioned briefly before: attributes can be added in several places. You can add them in classes, as well as anonymous classes:

```php
#[ClassAttribute]
class MyClass { /* … */ }

$object = new #[ObjectAttribute] class () { /* … */ };
```

Properties and constants:

```php
#[PropertyAttribute]
public int $foo;

#[ConstAttribute]
public const BAR = 1;
```

Methods and functions:

```php
#[MethodAttribute]
public function doSomething(): void { /* … */ }

#[FunctionAttribute]
function foo() { /* … */ }
```

As well as closures:

```php
$closure = #[ClosureAttribute] fn () => /* … */;
```

And method and function parameters:

```
function foo(#[ArgumentAttribute] $bar) { /* … */ }
```

If you want finer control over where your custom attributes are used, it's possible to configure them so they can only be used in specific places. For example you could make it so that `ClassAttribute` can only be used on classes, and nowhere else. Opting-in this behaviour is done by passing a flag to the `Attribute` attribute on the attribute class.

It looks like this:

```
#[Attribute(Attribute::TARGET_CLASS)]
class ClassAttribute
{
}
```

The following flags are available:

```
Attribute::TARGET_CLASS
Attribute::TARGET_FUNCTION
Attribute::TARGET_METHOD
Attribute::TARGET_PROPERTY
Attribute::TARGET_CLASS_CONSTANT
Attribute::TARGET_PARAMETER
Attribute::TARGET_ALL
```

These are bitmask flags, so you can combine them using a binary OR operation.

```
#[Attribute(Attribute::TARGET_METHOD | Attribute::TARGET_FUNCTION)]
class FunctionAttribute
{
}
```

Moving on, attributes can be declared before or after doc blocks:

```
/** @return void */
#[MethodAttribute]
public function doSomething(): void { /* … */ }
```

An attribute can take none, one or several arguments, which are defined by the attribute's constructor:

```
#[Attribute]
class ListensTo
{
    public function __construct(
        public string $event
    ) {}
}

#[ListensTo(ProductCreatedEvent::class)]
```

With regards to types of parameters you can pass to an attribute, you've already seen that constants, class names using `::class` and scalar types are allowed. In fact, attributes only accept so called *constant expressions* as input arguments. This means that scalar expressions are allowed — even bit shifts — as well as constants, arrays and array unpacking, boolean expressions and the null coalescing operator:

```
#[AttributeWithScalarExpression(1 + 1)]
#[AttributeWithClassNameAndConstants(PDO::class, PHP_VERSION_ID)]
#[AttributeWithClassConstant(Http::POST)]
#[AttributeWithBitShift(4 >> 1, 4 << 1)]
```

However, you can't give attributes a new instance of a class or a static method call for example - those are not constant expressions:

```
#[AttributeWithError(new MyClass())]
#[AttributeWithError(MyClass::staticMethod())]
```

By default, attributes cannot be repeated twice in the same place:

```
#[
    ClassAttribute,
    ClassAttribute,
]
class MyClass { /* … */ }
```

It is possible to change that behaviour though, again using a configuration flag like with target configuration:

```
#[Attribute(Attribute::IS_REPEATABLE)]
class ClassAttribute
{
}
```

Note that all configuration flags are only validated when calling `$attribute->newInstance()`, not earlier. This means that using an attribute in the wrong place might go unnoticed unless you're evaluating that attribute via reflection or static analysis.

BUILT-IN ATTRIBUTES

Once the base RFC for attributes had been accepted, new opportunities arose to add built-in attributes to the core. One such example is the `#[Deprecated]` attribute, as well as a `#[Jit]` attribute. I'm sure we'll see more and more built-in attributes in the future, but right now, none exist.

PhpStorm, the IDE, has done something very interesting though: they have added their own custom attributes in PhpStorm. These attributes aren't real classes in your codebase, but as we've seen you only need real attribute classes when you call `$reflectionAttribute->newInstance()`. The built-in attributes shipped by PhpStorm can't be instantiated, since they aren't real classes; but they can be understood by the IDE to add richer static analysis options. One example is the `#[ArrayShape]` attribute, which can teach PhpStorm what exactly is in an array:

```
#[ArrayShape([
    'key1' => 'int',
    'key2' => 'string',
    'key3' => 'Foo',
    'key3' => Foo::class,
])]
function legacyFunction(…): array
```

By adding such an attribute, PhpStorm now knows exactly what keys are in the array returned by `legacyFunction`, as well as the type of each key. I would be hesitant to write code that relies on array keys and types, though (I'd use DTOs for that). However, the `#[ArrayShape]` is an excellent way to document a legacy codebase, where you don't always have control over its design. There are more built-in attributes shipped by PhpStorm, by the way: `#[Deprecated]`, `#[Immutable]`, `#[Pure]`, `#[ExpectedValues]`, and even more.

It's interesting to see how PhpStorm embraces the static analysis mindset we've discussed earlier: we don't need to do any runtime checks since the static analyser (PhpStorm in this case) can tell us what we're doing wrong before running the code, thanks to attributes. We need to be careful though, that static analysis tools keep some interoperability between them. If each tool decides to implement their own version of attributes, it won't make the developer experience any better.

It'll be interesting to see what kind of attributes get added, both in the core and by third-party static analysers, and how they manage to work together.

CHAPTER 08

SHORT CLOSURES

Some might consider them long overdue, but they are finally supported since PHP 7.4: short closures. Instead of writing closures like the one passed as an `array_map` callback:

```
$itemPrices = array_map(
    function (OrderLine $orderLine) {
        return $orderLine->item->price;
    },
    $order->lines
);
```

You can write them in a short form:

```
$itemPrices = array_map(
    fn ($orderLine) => $orderLine->item->price,
    $order->lines
);
```

Short closures differ from normal closures in two ways:

- they only support one expression, and that's also the return statement; and
- they don't need a `use` statement to access variables from the outer scope.

Concerning everything else, they act like a normal closure would: they support references, argument spreading, type hints, return types… Speaking of types, you could rewrite the previous example in more strictly typed way, like so:

```php
$itemPrices = array_map(
    fn (OrderLine $orderLine): int => $orderLine->item->price,
    $order->lines
);
```

One more thing, references are also allowed, both for the arguments as well as the return values. If you want to return a value by reference, the following syntax should be used:

```php
fn&($x) => $x
```

NO MULTI-LINE

You might have noticed it already: short closures can only have one expression; it may be spread over multiple lines for formatting, but it must always be one expression. The reasoning is as follows: the goal of short closures is to reduce verbosity. `fn` is of course shorter than `function` in all cases. However, it was argued that if you're dealing with multi-line functions, there is less to be gained by using short closures. After all, multi-line closures are by definition already more verbose; so being able to skip two keywords (`function` and `return`) wouldn't make much of a difference.

While I can think of many one-line closures in my projects, there are also plenty of multi-line ones, and I admit to missing the short syntax in those cases. There's hope, though: it will be possible to add multi-line short closures in the future, but we'll have to wait still a little longer.

VALUES FROM OUTER SCOPE

Another significant difference between short and normal closures is that the short ones don't require the `use` keyword to be able to access data from the outer scope:

```php
$modifier = 5;

array_map(fn ($x) => $x * $modifier, $numbers);
```

It's important to note that you won't be able to modify variables from that outer scope: they are bound by value and not by reference, that is unless you're dealing with objects that are always passed by reference. That's the default behaviour for objects everywhere, by the way, not just in short closures.

One exception is of course the `$this` keyword, which acts exactly the same as normal closures:

```php
array_map(fn ($x) => $x * $this->modifier, $numbers);
```

Speaking of `$this`, you can declare a short closure to be `static`, which means you won't be able to access `$this` from within it:

```php
static fn ($x) => $x * $this->modifier;

// Fatal error: Uncaught Error: Using $this when not in object context
```

That's about all there is to say about short closures for now. You can imagine there's room for improvement. People have been talking about multi-line short closures and being able to use the syntax for class methods. We'll have to wait for future versions to see whether and how short closures will evolve.

CHAPTER 09

WORKING WITH ARRAYS

I've already used some array-specific syntax in the previous chapters, so it seems like a good idea to dedicate some time to them as well. Now don't worry: I won't discuss all array-related functions in PHP, there are too many of them, which would be rather boring. No, I'll only talk about what's made possible with arrays and PHP's syntax over the last year. There have been quite a lot of niceties added when dealing with arrays. We'll talk all about it in this chapter.

REST AND SPREAD

You've already seen the two uses of the `...` operator in previous examples: you can use it to "spread" array elements and pass them individually to functions, as well as make variadic functions that collect the "rest" of the arguments into an array.

Let's quickly recap. Here we're spreading array elements into a function:

```
$data = ['a', 'b', 'c'];

function handle($a, $b, $c): void { /* … */ }

handle(...$data);
```

And here, we're using a variadic function, which collects the remaining parameters and stores them in an array.

```
function handle($firstInput, ...$listOfOthers) { /* … */ }

handle('a', 'b', 'c', 'd');
```

In this case, `$firstInput` will be `'a'`, while `$listOfOthers` will be an array: `['b', 'c', 'd']`.

One interesting thing about variadic functions is that you can type hint them as well, so you can say that all variables passed into `$listOfOthers` should be, for example, strings:

```
function handle($firstInput, string ...$listOfOthers) { /* … */ }
```

You could also combine the two together. Here's a generic implementation of a static constructor for any class. It's wrapped in a trait so that it can be used in whatever class you want.

```
trait Makeable
{
    public static function make(...$args): static
    {
        return new static(...$args);
    }
}
```

In this example, we're taking a variable amount of input arguments and spreading them again into the constructor. This means that, whatever amount of variables the constructor takes, we can use `make` to pass those variables to it. Here's what that would look like in practice:

```
class CustomerData
{
    use Makeable;

    public function __construct(
        public string $name,
        public string $email,
        public int $age,
    ) {}
}

$customerData = CustomerData::make($name, $email, $age);

// Or you could use array spreading again:

$customerData = CustomerData::make(...$inputData);
```

One more thing to mention about array spreading: the syntax can be used to combine arrays as well:

```
$inputA = ['a', 'b', 'c'];

$inputB = ['d', 'e', 'f'];

$combinedArray = [...$inputA, ...$inputB];

// ['a', 'b', 'c', 'd', 'e', 'f']
```

It's a shorthand way to merge two arrays. There's one important note, though: you're only allowed to use this array-in-array-spreading syntax when the input arrays only have numeric keys - textual keys aren't allowed.

ARRAY DESTRUCTURING

Array destructuring is the act of pulling elements out of an array — it's about "destructuring" an array into separate variables. You can use both `list` or `[]` to do so. Note that the word is "destructure", not "destruction"!

Here's what that looks like:

```php
$array = [1, 2, 3];

// Using the list syntax:
list($a, $b, $c) = $array;

// Or the shorthand syntax:
[$a, $b, $c] = $array;

// $a = 1
// $b = 2
// $c = 3
```

Whether you prefer `list` or its shorthand `[]` is up to you. People might argue that `[]` is ambiguous with the shorthand array syntax and thus prefer list. I'll be using the shorthand version in code samples as that is my preference. I think that since the `[]` notation is on the left side of the assignment operator, it's clear enough that it's not an array definition.

So let's look at what's possible using this syntax.

SKIPPING ELEMENTS

Say you only need the third element of an array; the first two can be skipped by simply not providing a variable.

```
[, , $c] = $array;

// $c = 3
```

Also note that array destructuring on arrays with numeric indices will always start at index `0`. Take for example the following array:

```
$array = [
    1 => 'a',
    2 => 'b',
    3 => 'c',
];
```

The first variable pulled out would be `null`, because there's no element with index 0. This might seem like a shortcoming, but luckily there are more possibilities.

NON-NUMERICAL KEYS

PHP 7.1 allows array destructuring to be used with arrays that have non-numerical keys. This allows for more flexibility:

```php
$array = [
    'a' => 1,
    'b' => 2,
    'c' => 3,
];

['c' => $c, 'a' => $a] = $array;
```

As you can see, you can change the order however you want, and also skip elements entirely.

IN PRACTICE

One of the uses of array destructuring are with functions like `parse_url` and `pathinfo`. Because these functions return an array with named parameters, we can destructure the result to pull out the information we'd like:

```php
[
    'basename' => $file,
    'dirname' => $directory,
] = pathinfo('/users/test/file.png');
```

You can also see in this example that the variables don't need the same name as the key. If you're destructuring an array with an unknown key, PHP will issue a notice:

```php
[
    'path' => $path,
    'query' => $query,
] = parse_url('https://front-line-php.com');

// PHP Notice:  Undefined index: query
```

In this case, `$query` would be `null`.

You could observe one last detail in the example: trailing commas are allowed with named destructs, just like you're used to with arrays.

IN LOOPS

Array destructuring has even more use cases — you've already seen this one used in the attributes chapter. You can destructure arrays in loops:

```
$array = [
    [
        'name' => 'a',
        'id' => 1
    ],
    [
        'name' => 'b',
        'id' => 2
    ],
];

foreach ($array as ['id' => $id, 'name' => $name]) {
    // …
}
```

This could be useful when parsing, for example, a JSON or CSV file. Only be careful that undefined keys will still trigger a notice.

So there we have it: you're up to date with everything you can do using arrays in modern-day PHP. I find that most of these syntactical additions have their use cases. There are always other ways to achieve the same result that don't rely on shorthands - it's your choice. We'll talk more about these kinds of preferences in the chapter on style guides, but there are a few other topics to discuss first.

CHAPTER 10

MATCH

PHP 8 introduced the new `match` expression - a powerful feature that will often be the better choice compared to using `switch`. I say "often" because both `match` and `switch` also have specific use cases that aren't covered by the other. So what exactly are the differences? Let's start by comparing the two. Here's a classic `switch` example:

```
switch ($statusCode) {
    case 200:
    case 300:
        $message = null;
        break;
    case 400:
        $message = 'not found';
        break;
    case 500:
        $message = 'server error';
        break;
    default:
        $message = 'unknown status code';
        break;
}
```

And here is its `match` equivalent:

```php
$message = match ($statusCode) {
    200, 300 => null,
    400 => 'not found',
    500 => 'server error',
    default => 'unknown status code',
};
```

First of all, the match expression is significantly shorter:

- it doesn't require a `break` statement;
- it can combine different arms into one using a comma; and
- it returns a value, so you only have to assign the result once.

So from a syntactical point of view, `match` is always easier to write. There are more differences, though.

EXPRESSION OR STATEMENT?

I've called `match` an expression, while `switch` is a statement. There's indeed a difference between the two. An expression combines values and function calls, and is interpreted to a new value. In other words: it returns a result. This is why we can store the result of `match` into a variable, while that isn't possible with `switch`.

NO TYPE COERCION

`match` will do strict type checks instead of loose ones. It's like using `===` instead of `==`. However, there might be cases where you want PHP to automatically juggle a variable's type, which explains why you can't replace all switches with matches.

```php
$statusCode = '200';

$message = match ($statusCode) {
    200 => null
    default => 'Unknown status code',
};

// $message = 'Unknown status code'
```

UNKNOWN VALUES CAUSE AN ERROR

When there's no `default` arm and when a value comes in that doesn't match any given option, PHP will throw an `UnhandledMatchError` at runtime. Again more strictness, but it will prevent subtle bugs from going unnoticed.

```php
$statusCode = 400;

$message = match ($statusCode) {
    200 => 'perfect',
};

// UnhandledMatchError
```

ONLY SINGLE-LINE EXPRESSIONS, FOR NOW

Just like short closures, you can only write one expression. Expression blocks will probably get added at one point, but it's still not clear when exactly.

COMBINING CONDITIONS

I already mentioned the lack of `break`; this also means that `match` doesn't allow for fallthrough conditions, like the two combined `case` lines in the first `switch` example. On the other hand though, you can combine conditions on the same line, separated by commas:

```php
$message = match ($statusCode) {
    200, 300, 301, 302 => 'combined expressions',
};
```

COMPLEX CONDITIONS AND PERFORMANCE

When the `match` RFC was being discussed, some people suggested it wasn't necessary to add it, since the same was already possible without additional keywords but instead relying on array keys. Take this example where we want to match a value based on a more complex regex search. In here we're using the array notation some people mentioned as an alternative to `match`:

```
$message = [
    $this->matchesRegex($line) => 'match A',
    $this->matchesOtherRegex($line) => 'match B',
][$search] ?? 'no match';
```

But there's one big caveat: this technique will execute all regex functions first to build the array. `match`, on the other hand, will evaluate arm by arm, which is the more optimal approach.

THROWING EXCEPTIONS

Finally, because of throw expressions in PHP 8, it's also possible to directly throw from an arm, if you'd like to.

```
$message = match ($statusCode) {
    200 => null,
    500 => throw new ServerError(),
    default => 'unknown status code',
};
```

PATTERN MATCHING

There's one important feature still missing: proper pattern matching. It's a technique used in other programming languages to allow for intricate matching rules rather than simple comparisons. Think of it as regex, but for variables instead of text.

Pattern matching isn't supported right now because it's quite a complex feature. It has been mentioned as a future improvement for `match` though. I'm already looking forward to it!

SO, SWITCH OR MATCH?

If I'd need to summarise the `match` expression in one sentence, I'd say it's the stricter and more modern version of its little `switch` brother.

There are some cases — see what I did there? — where `switch` will offer more flexibility, especially with multiline code blocks and its type juggling. On the other hand, I find the strictness of `match` appealing, and the perspective of pattern matching would be a game-changer for PHP.

I admit I never wrote a `switch` statement in the past because of its many quirks; quirks that `match` actually solves. So while it's not perfect yet, there are use cases where `match` would be a good... match.

PART II

BUILDING WITH PHP

CHAPTER 11

OBJECT ORIENTED PHP

Now that you're up to speed with modern PHP syntax, it's time to look deeper into what kind of code we write with it. Throughout the next chapters, we'll zoom out to see the bigger picture. We'll start with a heavily debated topic: object-oriented programming.

Alan Kay, the inventor of the term "object-oriented programming", told a story once during a talk more than 20 years ago. You can build a dog house using only a hammer, nails, planks, and just a little bit of skill. I figure even I would be able to build it given enough time. Once you've built it you've earned the skills and know-how, and could apply it to other projects. Next, you want to build a cathedral, using the same approach with your hammer, nails, and planks. It's 100 times larger, but you've done this before — right? It'll only take a little longer.

While the scale went up by a factor of 100, its mass went up by a factor of 1,000,000 and its strength only by 10,000. Inevitably, the building will collapse. Some people plaster over the rubble, make it into a pyramid and say it was the plan all along; but you and I know what really went on.

You can watch Alan's talk here:
https://www.youtube.com/watch?v=oKg1hTOQXoY

Alan used this metaphor to explain a critical problem he saw with "modern OOP" 20 years ago. I think it still holds today: we've taken the solution to a problem — OO code — we've scaled it by a factor of 100, and expected it to work the same way. Even today, we don't think enough about architecture — which is rather crucial if you're building a cathedral — we use the OO solutions we learned without any extra thought. Most of us learned OO in isolation with small examples and rarely at scale. In most real-life projects, you cannot simply apply the patterns you've learned and expect everything to fall into place the same way it did with Animals, Cats, and Dogs.

This reckless scaling of OO code caused many people to voice their disapproval of it in recent years. I think that OOP is as good a tool as any other — functional programming being the modern-day popular contestant — *if* used correctly.

My takeaway from Alan's talk 20 years ago is that each object is a little program on its own, with its own internal state. Objects send messages between each other — packages of immutable data — which other objects can interpret and react to. You can't write all code this way, and that's acceptable — it's fine to not blindly follow these rules. Still, I have experienced the positive impact of this mindset first hand. Thinking of objects as little standalone programs, I started writing parts of my code in a different style. I hope that, now that we're going to look at object-oriented PHP, you'll keep Alan's ideas in mind. They taught me to critically look at what I took for granted as "proper OO", and learned there's more to it than you might think.

ALTERNATIVES TO OOP

I don't want to promote any tunnel vision in this book. I'm aware that there are other approaches to programming than only OOP. Functional programming, for example, has seen a tremendous increase in popularity in recent years. While I reckon that FP has its merits, PHP isn't optimised to program in a functional style. On the other hand, while OOP is the best match for PHP, all programmers can learn valuable lessons by learning about other programming styles, like a functional one.

I'd recommend you read a book called "Thinking Functionally in PHP" by Larry Garfield. In the book, Larry clearly shows why PHP isn't the perfect language to write functional programs with, but he also explains FP's mindset, visualised in PHP. And even though you wouldn't write functional PHP production code, there's lots of knowledge we can apply to OOP as well.

THE PITFALL OF INHERITANCE

I found it difficult to believe at first, but classes and inheritance have nothing to do with OOP the way Alan envisioned it. That doesn't mean they are bad things per se, but it is good to think about their purpose and how we can use (as well as abuse) them. Alan's vision only described objects — it didn't explain how those objects were created. Classes were added later as a convenient way to manage objects, but they are only an implementation detail, not OOP's core idea. With classes came inheritance, another useful tool when used correctly. That hasn't always been the case, though. Even when you might think it's one of the pillars of object-oriented design, they are misused very often, just like the doghouse Alan tried to scale up to a cathedral.

One of OOP's acclaimed strengths is that it models our code in ways humans think about the world. In reality, though, we rarely think in terms of abstractions and inheritance. Instead of using inheritance in places where it actually makes sense, we've been abusing it to share code, and configure objects in an obscure way. I'm going to show you a great example that illustrates this problem, though I want to say upfront that it isn't my own: it's Sandi Metz's, a great teacher on the subject of OOP. Let's take a look.

Sandi's talk: https://www.youtube.com/watch?v=OMPfEXIlTVE

There's a children's nursery rhyme called "The House That Jack Built" (it's also a horror movie, but that's unrelated). It starts like this:

```
This is the house that Jack built.
```

Every iteration, there's a sentence added to it:

```
This is the malt that lay in
        the house that Jack built.
```

And next

```
This is the rat that ate
        the malt that lay in
        the house that Jack built.
```

Get it? This is the final poem:

```
This is the horse and the hound and the horn that belonged to
        the farmer sowing his corn that kept
        the rooster that crowed in the morn that woke
        the priest all shaven and shorn that married
        the man all tattered and torn that kissed
        the maiden all forlorn that milked
        the cow with the crumpled horn that tossed
        the dog that worried
        the cat that killed
        the rat that ate
        the malt that lay in
        the house that Jack built.
```

Let's code this in PHP: a program that you can ask a given iteration, and it will produce the poem up until that point. Let's do it in an OO way. We start by adding all parts into a data array within a class; let's call that class `PoemGenerator` — sounds very OO, right? Good.

```php
class PoemGenerator
{
    private static array $data = [
        'the horse and the hound and the horn that belonged to',
        'the farmer sowing his corn that kept',
        'the rooster that crowed in the morn that woke',
        'the priest all shaven and shorn that married',
        'the man all tattered and torn that kissed',
        'the maiden all forlorn that milked',
        'the cow with the crumpled horn that tossed',
        'the dog that worried',
        'the cat that killed',
        'the rat that ate',
        'the malt that lay in',
        'the house that Jack built',
    ];
}
```

Now let's add two methods generate and phrase. generate will return the end result, and phrase is an internal function that glues the parts together.

```php
class PoemGenerator
{
    // …

    public function generate(int $number): string
    {
        return "This is {$this->phrase($number)}.";
    }

    protected function phrase(int $number): string
    {
        $parts = array_slice(self::$data, -$number, $number);

        return implode("\n        ", $parts);
    }
}
```

It seems like our solution works: we can use `phrase` to take x-amount of items from the end of our data array and implode those into one phrase. Next, we use `generate` to wrap the final result with `This is` and `.`. By the way, I implode on that spaced delimiter just to format the output a little nicer.

```
$generator = new PoemGenerator();

$generator->generate(4);

// This is the cat that killed
//         the rat that ate
//         the malt that lay in
//         the house that Jack built.
```

Exactly what we'd expect the result to be.

Then comes along... a new feature request. Let's build a random poem generator: it will randomise the order of the phrases. How do we solve this in a clean way without copying and duplicating code? Inheritance to the rescue — right? First, let's do a little refactor: let's add a protected `data` method so that we have a little more flexibility in what it actually returns:

```php
class PoemGenerator
{
    protected function phrase(int $number): string
    {
        $parts = array_slice($this->data(), -$number, $number);

        return implode("\n        ", $parts);
    }

    protected function data(): array
    {
        return [
            'the horse and the hound and the horn that belonged to',
            // …
            'the house that Jack built',
        ];
    }
}
```

Next we build our `RandomPoemGenerator`:

```php
class RandomPoemGenerator extends PoemGenerator
{
    protected function data(): array
    {
        $data = parent::data();

        shuffle($data);

        return $data;
    }
}
```

How great is inheritance! We only needed to override a small part of our code, and everything works just as expected!

```php
$generator = new RandomPoemGenerator();

$generator->generate(4);

// This is the priest all shaven and shorn that married
//          the cow with the crumpled horn that tossed
//          the man all tattered and torn that kissed
//          the rooster that crowed in the morn that woke.
```

Awesome!

Once again... a new feature request: an echo generator: it repeats every line a second time. So you'd get this:

```
This is the malt that lay in the malt that lay in
        the house that Jack built the house that Jack built.
```

We can solve this; inheritance — right?

Let's again do a small refactor in `PoemGenerator`, just to make sure our code stays clean. We can extract the array slicing functionality in phrase to its own method, which seems like a better separation of concerns.

```
class PoemGenerator
{
    // …

    protected function phrase(int $number): string
    {
        $parts = $this->parts($number);

        return implode("\n        ", $parts);
    }

    protected function parts(int $number): array
    {
        return array_slice($this->data(), -$number, $number);
    }
}
```

Having refactored this, implementing `EchoPoemGenerator` is again very easy:

```php
class EchoPoemGenerator extends PoemGenerator
{
    protected function parts(int $number): array
    {
        return array_reduce(
            parent::parts($number),
            fn (array $output, string $line) =>
                [...$output, "{$line} {$line}"],
            []
        );
    }
}
```

Can we take a moment to appreciate the power of inheritance? We've created two different implementations of our original `PoemGenerator`, and have only overridden the parts that differ from it in `RandomPoemGenerator` and `EchoPoemGenerator`. We've even used SOLID principles (or so we think) to ensure that our code is decoupled so that it's easy to override specific parts. This is what great OOP is about — right?

One more time... another feature request: please make one more implementation, one that combines both the random and echo behaviour: `RandomEchoPoemGenerator`.

Now what? Which class will that one extend?

If we're extending `PoemGenerator`, we'll have to override both our `data` and `parts` methods, essentially copying code from both `RandomPoemGenerator` and

`EchoPoemGenerator`. That's bad design, copying code around. What if we extend `RandomPoemGenerator`? We'd need to reimplement `parts` from `EchoPoemGenerator`. If we'd implement `EchoPoemGenerator` instead, it would be the other way around.

To be honest, extending `PoemGenerator` and copying both implementations seems like the best solution. Since then, we're at least making it clear to future programmers that this is a thing on its own, and we weren't able to solve it any other way.

But let's be frank: whatever solution, it's all crap. We have fallen into the pitfall that is inheritance. And this, dear reader, happens so often in real-life projects: we think of inheritance as the perfect solution to override and reuse behaviour, and it always seems to work great at the start. Next comes along a new feature that causes more abstractions, and causes our code to grow out of hand. We thought we mastered inheritance but it kicked our asses instead.

So what's the problem — the *actual* problem — with our code? Doesn't it make sense that `RandomPoemGenerator` extends from `PoemGenerator`? It is a poem generator, isn't it? That's indeed the way we think of inheritance: using *"is a"*. And yes, `RandomPoemGenerator` *is a* `PoemGenerator`, but `RandomPoemGenerator` isn't *only* generating a poem now, is it?

Sandi Metz suggests the following question to identify the underlying problem: "what changed between the two — what changed during inheritance?". Well... In the case of `RandomPoemGenerator`, it's the `data` method; for `EchoPoemGenerator`, it's the `parts` method. And it just so happens that having to combine those two parts is what made our inheritance solution blow up.

Do you know what this means? It means that `parts` and `data` are something on their own. They are *more* than a protected implementation detail of our poem generator. They are what is valued by the client, they are the *essence* of our program.

So let's treat them as such.

With two separate concerns identified, we need to give them a proper name. The first one is about whether lines should be randomised or not. Let's call it the `Orderer`; it will take an original array and return a new version of it with its items sorted in a specific way.

```
interface Orderer
{
    public function order(array $data): array;
}
```

The second concern is about formatting the output - whether it should be echoed or not. Let's call this concept a `Formatter`. Its task is to receive the array of lines and format all of those lines into one string.

```
interface Formatter
{
    public function format(array $lines): string;
}
```

And here comes the magic. We're extracting this logic from our `PoemGenerator`, but we still need a way to access it from within. So let's inject both an orderer and formatter into the `PoemGenerator`:

```
class PoemGenerator
{
    public function __construct(
        public Formatter $formatter,
        public Orderer $orderer,
    ) {}

    // …
}
```

With both available, let's change the implementation details of phrase and data:

```php
class PoemGenerator
{
    // ...

    protected function phrase(int $number): string
    {
        $parts = $this->parts($number);

        return $this->formatter->format($parts);
    }

    protected function data(): array
    {
        return $this->orderer->order([
            'the horse and the hound and the horn that belonged to',
            // ...
            'the house that Jack built',
        ]);
    }
}
```

And finally, let's implement `Orderer`:

```php
class SequentialOrderer implements Orderer
{
    public function order(array $data): array
    {
        return $data;
    }
}

class RandomOrderer implements Orderer
{
    public function order(array $data): array
    {
        shuffle($data);

        return $data;
    }
}
```

As well as `Formatter`:

```php
class DefaultFormatter implements Formatter
{
    public function format(array $lines): string
    {
        return implode("\n        ", $lines);
    }
}

class EchoFormatter implements Formatter
{
    public function format(array $lines): string
    {
        $lines = array_reduce(
            $lines,
            fn (array $output, string $line) =>
                [...$output, "{$line} {$line}"],
            []
        );

        return implode("\n        ", $lines);
    }
}
```

The default implementations, `DefaultFormatter` and `SequentialOrderer` might not do any complex operations, though still they are a valid business concern: a "sequential order" and "default format" are two valid cases needed to create the poem as we know it in its normal form.

Do you realise what just happened? You might be thinking that we're writing more code, but you're forgetting something... we can remove our `RandomPoemGenerator` and

`EchoPoemGenerator` altogether, we don't need them anymore, we can solve all of our cases, with only the `PoemGenerator`:

```php
$generator = new PoemGenerator(
    new EchoFormatter(),
    new RandomOrderer(),
);
```

We can make our lives still a little easier by providing proper defaults:

```php
class PoemGenerator
{
    public function __construct(
        public ?Formatter $formatter = null,
        public ?Orderer $orderer = null,
    ) {
        $this->formatter ??= new DefaultFormatter();
        $this->orderer ??= new SequentialOrderer();
    }
}
```

And using named properties, we can construct a `PoemGenerator` whatever way we want:

```php
$generator = new PoemGenerator(
    formatter: new EchoFormatter(),
);

$generator = new PoemGenerator(
    orderer: new RandomOrderer(),
);

$generator = new PoemGenerator(
    formatter: new EchoFormatter(),
    orderer: new RandomOrderer(),
);
```

No more need for a third abstraction!

This is *real* object-oriented programming. I told you that OOP isn't about inheritance, and this example shows its true power. By composing objects out of other objects, we're able to make a flexible and durable solution, one that solves all of our problems in a clean way. This is what composition over inheritance is about, and it's one of the most fundamental pillars in OO.

I'll admit: I don't always use this approach when I start writing code. It's often easier to start simply during the development process and not think about abstracts or composition. I'd even say it's a great rule to follow: don't abstract too soon. The important lesson isn't that you should always use composition. Instead, it's about identifying the problem you encounter and using the right solution to solve it.

WHAT ABOUT TRAITS?

You might be thinking about traits to solve our poem problem. You could make a `RandomPoemTrait` and `EchoPoemTrait`, implementing `data` and `phrase`. Yes, traits can be another solution, just like inheritance also is a working solution. I'm going to make the case why composition is still the better choice, but first let's show in practice what these traits would look like:

```php
trait RandomPoemTrait
{
    protected function data(): array
    {
        $data = parent::data();

        shuffle($data);

        return $data;
    }
}
```

```
trait EchoPoemTrait
{
    protected function parts(int $number): array
    {
        return array_reduce(
            parent::parts($number),
            fn (array $output, string $line) =>
                [...$output, "{$line} {$line}"],
            []
        );
    }
}
```

You could use these to implement `RandomEchoPoemGenerator` like so:

```
class RandomEchoPoemGenerator extends PoemGenerator
{
    use RandomPoemTrait;
    use EchoPoemTrait;
}
```

Traits indeed solve the problem of code reusability; that's exactly why they were added to the language. When I mentioned the latest feature request to add `RandomEchoPoemGenerator`, I remarked that there was no clean way to solve the problem without code duplication, which was the stepping stone to search for another solution — composition. Did I deliberately ignore traits to make my point? No. While they do solve the problem of reusability, they *don't* have the added benefits we discovered when exploring composition.

First we discovered that the order and format of the poem are crucial business rules and shouldn't be treated as protected implementation details somewhere in the class.

We made `Orderer` and `Formatter` to make our code better represent the real-world problem we're trying to solve. If we're choosing traits and subclasses instead, we lose this explicitness once again.

Second, our poem example shows two parts of the `PoemGenerator` that are configurable. What if there's three or four? If we're adding two more traits, we also have to create new subclass implementations for those traits, *and* all relevant combinations with existing traits. The number of subclasses would grow exponentially. Even with our current example, there's already three subclasses: `RandomPoemGenerator`, `EchoPoemGenerator` and `RandomEchoPoemGenerator`. Composition on the other hand only requires us to add only two new classes. Our code would grow out of hand if there were more complex business rules to account for.

I'm not suggesting traits shouldn't be used at all; just like inheritance, they have their uses. What's most important is that you critically assess the pros and cons of all solutions for a given problem instead of falling back to what seems the easiest at first.

I think this reasoning applies to everything programming related, whether you're coding in an object-oriented, procedural, or functional style. OOP got a bad name because people started to scale it out of hand, without rethinking their architecture. I hope we can change that.

CHAPTER 12

MVC FRAMEWORKS

The most popular way by far to build web applications in PHP is by using the Model View Controller (MVC) pattern. Most modern PHP frameworks build upon the same idea. At first, a request comes in and is mapped via its URL to a controller. This controller takes input from the request and passes it to the model. The model encapsulates all business logic and rules of an application. The result returned from the model can be passed by the controller to the view, which is the end result returned to the user.

The benefit of the MVC pattern is its loosely coupled parts: model code can be written without having to worry about how it will be presented in the UI, and view code can be written without worrying about how exactly its data is retrieved.

COMPARISONS?

No framework — Symfony, Laravel, and there are lots more — is the perfect one. There will always be things you like more about one than another. Those preferences often originate from your past experiences, your education, the languages you've used before, other frameworks, the kinds of projects you've done, as well as your character. There's no right or wrong choice, but there is one metric I'd take into account when choosing a framework for your next project: find one that is actively maintained and has a large ecosystem surrounding it. Both Symfony and Laravel fall within this category, so really, the choice is up to you.

The reason I'm only mentioning those two frameworks by the way is twofold: they are very popular within the PHP world, and I have worked personally with both of them, which I think is an essential requirement for me to write about them.

If you'd ask the community, they would probably say that Symfony is heavily inspired by the Java world, known for its robustness and durability. Laravel is inspired by Ruby on Rails and known for its ease-of-use and rapid application development mindset. This means very little, though: you can get a Symfony application up and running as fast as Laravel; the same way you can write large-scale applications with Laravel as well.

There are noticeable differences between the two. For one, the recommended code style: Laravel promotes code that's as easy as possible to read, while Symfony wants code to be written clearly and correct, even when it's more verbose. My advice to any developer regarding picking a framework is: don't lose yourself in debates over what's better or not. Both are great tools, and you can achieve the same great results with them. Also, don't try to fight the framework too much: everything is so much easier if you follow the way the framework intends you to.

Here's an example: Laravel has a powerful ORM implementation called "Eloquent".

ORMS

An ORM – Object Relational Mapper — is a system that exposes the data in, for example, a database as objects in your code. It's a powerful tool that abstracts away the implementation details of database queries and allows you to think with objects instead of arrays of raw data.

Eloquent has been a highly debated topic: it uses the active record pattern which is considered to be an anti-pattern by many software developers. It's still a highly popular pattern, also in other languages, so it's definitely legit. The active record pattern breaks all the rules I've described in the previous chapter: it relies entirely on inheritance, which comes with the setbacks we already discovered. There are some other issues with it still, and here's a critique on the pattern by Robert "Uncle Bob" Martin:

> The problem I have with Active Record is that it creates confusion about these two very different styles of programming. A database table is a data structure. It has exposed data and no behavior. But an Active Record appears to be an object. It has “hidden” data, and exposed behavior. I put the word “hidden” in quotes because the data is, in fact, not hidden. Almost all Active Record derivatives export the database columns through accessors and mutators.

Active record is not "correct" as defined by the rules. But Laravel has built an extremely rich API on top of it. It's by far one of the easiest ORM implementations to use. That is easiest for many cases, not all.

Like I said: don't fight the framework. You pick one for all the great things it provides, so you'll also have to deal with its little quirks and downsides. If I would write a framework from scratch, I probably wouldn't use the active record pattern to build my ORM layer upon. But is it so bad that I should spend hours and days trying to change it in Laravel? No. When I use Laravel, I'm fine with using Eloquent as-is. It offers so much value in other places that it simply isn't worth fighting it.

I once worked at a company which had written their own custom framework because all popular alternatives had little quirks. This had been a company decision five years before I joined, and they made somewhere between 100 and 200 websites with it. Most were smaller company websites, but some were complex systems. As the company grew, they took on more ambitious projects and refused to work with other frameworks.

More than a few of those ambitious projects failed or only partially worked because of this mentality. It also turned out their framework also had its limitations, and there was no community to fall back on. There were a few angry clients, which resulted in significant financial loss. Several colleagues — including myself — left the company because there was no room to grow in areas we believed were essential in the web development world. We were locked-in and decided to jump ship before it was too late. Finally, the company saw its mistakes and managed to switch to community-backed frameworks, but only after they had thousands, if not millions of lines of code, all tied to their own framework — legacy that they'll have to support for years to come.

Dare to make compromises. During the first year I used Laravel, I clashed with it on many different fronts. I was used to Symfony at that point and appreciated its strictness. I had a hard time adapting to Laravel, and it took me a whole year to realise I just had to stop fighting it. In the end, I embraced the framework and built on top of it, instead of trying to tear it down first partly. I learned that whatever framework or programming language you use isn't a religion; they are merely tools to help you get a job done.

Let's go back to MVC frameworks. I won't give you a step-by-step guide on setting up a Symfony or Laravel application; both communities already have excellent guidelines that cover all the topics you'll need to get started and I couldn't possibly match their quality in one chapter. Instead, I want to give you some general pointers. There's an important skill to learn when it comes to frameworks: independently finding your way around code.

Frameworks such as Laravel and Symfony have huge codebases. If you treat them like a black box, you'll never be able to use them to their full potential. It's essential to dive into source code, whether it's a framework or any other kind of codebase. It's a great skill to read code that's strange to you and follow its line of thinking. Even when you don't have any experience with a framework like Laravel or Symfony, if you're able to dive into the code to understand what's going on, you're miles ahead of many other programmers.

So let's do some code diving! I've mostly used Laravel the past three years, so I'm going to use that one as an example. Whether you have experience with it or not is irrelevant; we're interested in understanding what otherwise would be a black box. Because we're talking about MVC, let's look into how the request/response cycle is handled: the system that makes the MVC pattern work.

Imagine that we know absolutely nothing of this system. Where to start? Well `index.php` is usually a good place:

```php
/**
 * Laravel - A PHP Framework For Web Artisans
 *
 * @package  Laravel
 * @author   Taylor Otwell <taylor@laravel.com>
 */

define('LARAVEL_START', microtime(true));
```

```
/*
|--------------------------------------------------------------------------
| Register The Auto Loader
|--------------------------------------------------------------------------
|
| Composer provides a convenient, automatically generated class loader
for
| our application. We just need to utilize it! We'll simply require it
| into the script here so that we don't have to worry about manual
| loading any of our classes later on. It feels great to relax.
|
*/

require __DIR__.'/../vendor/autoload.php';

/*
|--------------------------------------------------------------------------
| Turn On The Lights
|--------------------------------------------------------------------------
|
| We need to illuminate PHP development, so let us turn on the lights.
| This bootstraps the framework and gets it ready for use, then it
| will load up this application so that we can run it and send
| the responses back to the browser and delight our users.
|
*/

$app = require_once __DIR__.'/../bootstrap/app.php';
```

```
/*
|--------------------------------------------------------------------------
| Run The Application
|--------------------------------------------------------------------------
|
| Once we have the application, we can handle the incoming request
| through the kernel, and send the associated response back to
| the client's browser allowing them to enjoy the creative
| and wonderful application we have prepared for them.
|
*/

$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
);

$response->send();

$kernel->terminate($request, $response);
```

As you can see, Laravel is nice enough to provide us some comments out of the box, explaining exactly what's going on! Personally, I find those comments a little... poetic, adding unnecessary noise, but we'll deal with it.

This is an interesting part:

```
/*
| …
|
| This bootstraps the framework and gets it ready for use, then it
| will load up this application so that we can run it and send
| the responses back to the browser and delight our users.
|
*/

$app = require_once __DIR__.'/../bootstrap/app.php';
```

Some important things are happening in `bootstrap/app.php`, let's take a look:

```
/* … */

$app = new Illuminate\Foundation\Application(
    $_ENV['APP_BASE_PATH'] ?? dirname(__DIR__)
);

/* … */

$app->singleton(
    Illuminate\Contracts\Http\Kernel::class,
    App\Http\Kernel::class
);
```

```php
$app->singleton(
    Illuminate\Contracts\Console\Kernel::class,
    App\Console\Kernel::class
);

$app->singleton(
    Illuminate\Contracts\Debug\ExceptionHandler::class,
    App\Exceptions\Handler::class
);

/* … */

return $app;
```

Again some comments — which I left out — and indeed, the setup of an `$app` variable. You may or may not know what those `singleton` calls are about; and that's OK for now. Based on the name of `$app`, we can guess that it is probably a central part of the framework, which is even made clearer given its own dedicated bootstrap file. Let's go back to `index.php` first, to see how `$app` is used:

```php
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
);

$response->send();

$kernel->terminate($request, $response);
```

Remember the three steps I discussed at the start of this chapter? Request to controller, controller to model and view, and that result returned as the response. All of that is happening in these lines of code:

```
$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
);

$response->send();
```

The first step is a little hidden since it's done inline: the request is captured — it's constructed based on the `$_SERVER`, `$_COOKIE`, `$_POST`, and `$_GET` global variables. Next, the captured request is passed to a kernel that handles it, which finally results in a response sent to the user.

For this example, we want to know how the controller is reached, so let's ignore the request capturing and response sending, and instead focus on the `$kernel->handle()`. There's the first real roadblock: what exactly is `$kernel`? We need to know where `handle` is implemented so that we can look at it.

`$kernel` isn't normally instantiated, instead it's created with `$app->make()` which is given an interface `Illuminate\Contracts\Http\Kernel` instead of a concrete class. There's actually two ways to solve our problem. First there's the implementation of

`$app->make()`, we know that `$app` is `Illuminate\Foundation\Application`, because it was manually constructed in the bootstrap file, so let's look over there:

```
/**
 * Resolve the given type from the container.
 *
 * @param  string  $abstract
 * @param  array  $parameters
 * @return mixed
 */
public function make($abstract, array $parameters = [])
{
    $this->loadDeferredProviderIfNeeded(
        $abstract = $this->getAlias($abstract)
    );

    return parent::make($abstract, $parameters);
}
```

In this case, the implementation isn't even important because the doc comment already explains what's going on: "Resolve the given type from the container". If you're unfamiliar with the dependency injection and dependency container patterns, you might still need to dive deeper. Based on this comment, however, most of you probably know what's going on: we're asking `$app` to make an instance of `Illuminate\Contracts\Http\Kernel`, which is seemingly registered in the container.

That's right, we've already seen this happening in `bootstrap.php`:

```
$app->singleton(
    Illuminate\Contracts\Http\Kernel::class,
    App\Http\Kernel::class
);
```

This definition says that when we're making `Illuminate\Contracts\Http\Kernel`, a concrete instance of `App\Http\Kernel` should be returned! Another way to find this out would be to look at which classes implement `Illuminate\Contracts\Http\Kernel`; or you can simply run the code, and inspect what object is represented in `$kernel` by dumping or debugging it.

Whatever solution works best, now we know that `App\Http\Kernel` is the place to look for the `handle` method. And indeed, this is the right place to be.

```php
/**
 * Handle an incoming HTTP request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function handle($request)
{
    try {
        $request->enableHttpMethodParameterOverride();

        $response = $this->sendRequestThroughRouter($request);
    } catch (Throwable $e) {
        $this->reportException($e);

        $response = $this->renderException($request, $e);
    }

    $this->app['events']->dispatch(
        new RequestHandled($request, $response)
    );

    return $response;
}
```

This might seem like another rabbit hole at first, but we can apply the same techniques just like before. It's also good to focus on the happy path first: we can assume that's the correct way to dive. So ignore that `catch` block, as well as whatever is happening with dispatching `RequestHandled` to `$this->app['events']`. Based on their names, it will dispatch an event after the request is handled.

That leaves us with only two lines of code:

```
$request->enableHttpMethodParameterOverride();

$response = $this->sendRequestThroughRouter($request);
```

And reading those lines, it's clear that `$this->sendRequestThroughRouter($request)` is where we need to go next.

```
/**
 * Send the given request through the middleware / router.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
protected function sendRequestThroughRouter($request)
{
    $this->app->instance('request', $request);

    Facade::clearResolvedInstance('request');

    $this->bootstrap();

    return (new Pipeline($this->app))
                ->send($request)
                ->through(
                    $this->app->shouldSkipMiddleware()
                        ? []
                        : $this->middleware
                )
                ->then($this->dispatchToRouter());
}
```

Next we see our request sent via a pipeline through middleware, to finally be dispatched to the router. If you're familiar with MVC frameworks, route middleware probably already rings a bell. Even if you don't know about middleware we're not blocked: `$this->dispatchToRouter()` is probably where we need to go.

```php
/**
 * Get the route dispatcher callback.
 *
 * @return \Closure
 */
protected function dispatchToRouter()
{
    return function ($request) {
        $this->app->instance('request', $request);

        return $this->router->dispatch($request);
    };
}
```

You can see the same pattern emerging throughout our dive. We read the code, determine what's relevant and what's not, and move into the next layer. You've also seen that theoretical knowledge can help: dependency injection and middleware are two popular techniques in many MVC frameworks.

For the sake of keeping this example somewhat readable, I'm going to skip through several layers where we move from one method to another. So after going even deeper, we've arrived at `Route::run`:

```
/**
 * Run the route action and return the response.
 *
 * @return mixed
 */
public function run()
{
    $this->container = $this->container ?: new Container;

    try {
        if ($this->isControllerAction()) {
            return $this->runController();
        }

        return $this->runCallable();
    } catch (HttpResponseException $e) {
        return $e->getResponse();
    }
}
```

Here we see the first mention of a controller:

```
if ($this->isControllerAction()) {
    return $this->runController();
}
```

So now it's a matter of going one step deeper still:

```php
/**
 * Run the route action and return the response.
 *
 * @return mixed
 *
 * @throws \Symfony\Component\HttpKernel\Exception\NotFoundHttpException
 */
protected function runController()
{
    return $this->controllerDispatcher()->dispatch(
        $this, $this->getController(), $this->getControllerMethod()
    );
}
```

That controller dispatcher opens a new rabbit hole! We're going to stop our dive, but you can go deeper yourself if you want to. You can apply the same techniques, and you'll very soon arrive at the actual controller code!

We did this exercise to show you that code diving shouldn't be all that difficult and is a great skill to have in your toolset. It makes you more flexible and independent as a developer and allows you to better understand and solve problems.

You can apply these code-diving techniques to every codebase, by the way, not just frameworks. I must admit that frameworks like Laravel and Symfony are generally more and better documented. We dove in rather clear waters today, but what if you need to work on a 10-year old legacy project? Well, the same techniques work just as well, but your dive will probably be much slower. Be patient, and take a break now and then.

CHAPTER 13

DEPENDENCY INJECTION

The previous two chapters hinted at the importance of the dependency injection pattern. When using composition, we injected dependencies from the outside into an object, and we also noticed that Laravel uses some kind of dependency container at its core. So what exactly is dependency injection?

There's often confusion amongst developers when it comes to this pattern. It is not because they don't know what it is, but because the pattern is often combined with other patterns that are wrongfully called "dependency injection". So let's make it clear up front: dependency injection is the technique of providing (injecting) an object's dependencies (the objects that it needs to "do its thing") from the outside. It says nothing about how those dependencies are injected; there are other patterns to solve that problem. Let's look at a comparison. Here's an example that doesn't use dependency injection:

```php
class QueryBuilder
{
    private Connection $connection;

    public function __construct() {
        $connectionString = config('db.connection');

        $this->connection = new Connection($connectionString);
    }
}
```

And this is the same example *with* dependency injection, note that we're using constructor property promotion again:

```
class QueryBuilder
{
    public function __construct(
        private Connection $connection
    ) {}
}
```

The idea of dependency injection is that an object's dependencies are passed into it from the outside so that the object doesn't need to worry about managing those dependencies itself. This pattern allows an object to focus on their task and not worry about constructing and managing related objects.

That leaves us with the question about how we're going to pass dependencies into an object. Should we always manually construct the query builder when using it?

```
class PostsController
{
    public function index()
    {
        $queryBuilder = new QueryBuilder(
            new Connection(config('db.connection'))
        );
    }
}
```

No. This is a very inefficient way of managing your code. Imagine the constructor of `QueryBuilder` changing - you now have to refactor tens, if not hundreds of places where it was used.

So very often, dependency injection comes together with another pattern: the dependency container. This is a class that knows how to construct objects in your codebase. You could configure such a container within PHP itself, though frameworks like Symfony allow YAML and XML configuration as well. For the sake of this example, we're going to write a simple container implementation from scratch in plain PHP. In fact; it's surprisingly simple:

```php
class Container
{
    private array $definitions = [];

    public function make(string $name): ?object
    {
        $definition = $this->definitions[$name] ?? fn () => null;

        return $definition($this);
    }

    public function register(string $name, Closure $definition): self
    {
        $this->definitions[$name] = $definition;

        return $this;
    }
}
```

The container itself is a simple class that has an array of definitions which you can register and use to make new instances with. Here's how we'd register the `Connection` and `QueryBuilder`:

```php
$container->register(
    Connection::class,
    fn () => new Connection(
        config('db.connection')
    ),
);

$container->register(
    QueryBuilder::class,
    fn (Container $container) => new QueryBuilder(
        $container->make(Connection::class)
    ),
);
```

And this is how we'd use it:

```php
$queryBuilder = $container->make(QueryBuilder::class);
```

Each definition is a closure on its own, and it's called by the container internally when we ask it for a dependency. By passing an instance of the container into the definition closure, we can resolve nested dependencies, as you can see when we're registering the `QueryBuilder` definition. In essence, the container is a store that you can save object construction definitions to.

Keep in mind that this is a simplified implementation; real-life containers also support features such as singletons and autowiring. These concepts are so widely used that we're going to discuss them in this chapter as well.

Singletons are objects that are only instantiated once, instead of every time you ask the container for them. If `Connection` was registered as a singleton, we could call `$container->make(Connection::class)` as many times as we'd want; it would only make a new object once, and reuse it over and over again.

There might be cases where an object takes a substantial amount of work to be constructed and where it's good to reuse the same object again in different places. For example: if `Connection` would have to test the credentials by connecting to a database server, it'd be good that it only did this once, instead of every time we requested `Connection` from the container.

So let's add that functionality to our container.

```
class Container
{
    private array $instances = [];

    // …
    public function singleton(string $name, Closure $definition): self
    {
        $this→register($name, function () use ($name, $definition) {
            if (array_key_exists($name, $this→instances)) {
                return $this→instances[$name];
            }

            $this→instances[$name] = $definition($this);

            return $this→instances[$name];
        });

        return $this;
    }
}
```

By registering a singleton definition, we're wrapping the original definition closure in another one, which will first check whether there's already an instance for the given name in the `$instances` array and if that's the case, we'll return it. Otherwise, we'll call the original closure and store its result in the `$instances` array, cached for the next time. Note that we need to use `array_key_exists` and not the null coalescing assignment operator: we want it to be possible for definitions to resolve to `null`. If we'd use null coalescing, our singleton wouldn't work correctly with `null` values.

The other feature I mentioned is autowiring: it allows the container to magically resolve classes based on their name, even when they weren't registered. This only works if all dependencies of the requested class can be autowired because as soon as there's a dependency that takes constructor arguments that can't be resolved by the container, we're stuck.

Here's a simplistic implementation:

```php
class Container
{
    // …
    public function make(string $name): object
    {
        $definition = $this->definitions[$name] ??
            $this->autowire($name);

        return $definition($this);
    }

    private function autowire(string $name): Closure
    {
        return function () use ($name) {
            $class = new ReflectionClass($name);

            $constructorArguments = $class
                ->getConstructor()
                ->getParameters();

            $dependencies = array_map(
                fn (ReflectionParameter $parameter) =>
                    $this->make($parameter->getType()),
                $constructorArguments,
            );

            return new $name(...$dependencies);
        };
    }
}
```

First, we're adding the autowire functionality as a fallback in `make`: if no existing definition is found, we will try and autowire it. The autowire method returns an on-the-fly definition closure that will look at the constructor arguments of the given class and try to resolve them via the container. Finally, when all dependencies are resolved, the actual class can be created.

This solution cuts a lot of corners, though: what if the dependencies can't be resolved? What if the class isn't a class? For the sake of the example, though, we'll keep it this simple.

By now, we've covered the container, singletons, and autowiring; they are all useful techniques built upon dependency injection. There's one more thing that's often being done with the dependency container, but one that should be avoided. It's called service location, and it is an anti-pattern.

Service location is the act of reaching in the container from within another class. In our query builder example, it would look like this:

```php
class QueryBuilder
{
    private Connection $connection;

    public function __construct(Container $container)
    {
        $this->connection = $container->make(Connection::class);
    }
}
```

The service locator pattern can come in different forms: here we inject the container and call its `make` method from within our constructor, but it could also be a standalone function or static method: `resolve(Connection::class)` or `Container::make(Connection::class)`. Whatever form, a service locator will reach into the global state of our program (often the dependency container), and will manually resolve dependencies.

The first problem with service location is that it disables the ability to use proper composition since we can't inject the dependencies anymore into the class. Second, the class now encapsulates and hides its dependencies. Looking at the constructor signature, we can't tell anymore what classes this one depends on. We've obfuscated our code. Third, we're losing static analysis capabilities: `$container->make(Connection::class)` relies on runtime reflection to build the right dependency, so static analysis won't have all the insights we'd want.

Even without service location, we're of course calling `$container->make()` very often. But those calls always happen within the container's context; they are in one centralised place instead of scattered around the codebase. This is the preferable approach: all places outside the container can assume the dependency is valid, and it's the container's task to resolve them properly.

My advice: avoid service location since there's almost always a cleaner way to solve the same problem. Truth be told, we've actually done service location in one of our previous examples, when we registered our `Connection` in the container.

```
$container->singleton(
    Connection::class,
    fn (Container $container) => new Connection(
        config('db.connection')
    ),
);
```

Can you spot the issue? The `config('db.connection')` call is actually reaching into the global application state - a sneaky form of service location!

Giving it some more thought: why not treat application config as an object itself? We could, for example, have a simple data object like this:

```php
class DatabaseConfig
{
    public function __construct(
        public string $connection,
        public ?string $port,
        // …
    ) {}
}
```

We still need to read environment variables to fill this config object, so we could register it as a singleton in the container like so:

```php
$container->singleton(
    DatabaseConfig::class,
    fn () => new DatabaseConfig(
        env('db.connection'),
        env('db.port'),
        // …
    ),
);
```

Next we can register the connection with this:

```
$container->singleton(
    Connection::class,
    fn (Container $container) => new Connection(
        $container->make(DatabaseConfig::class)->connection
    ),
);
```

We could even choose to inject `DatabaseConfig` directly into `Connection`, which allows us to use our container's autowiring capabilities to its full extent!

```
class Connection
{
    public function __construct(
        private DatabaseConfig $databaseConfig
    ) {}
}
```

The dependency injection pattern is truly powerful: it enables us to write clean and decoupled code and is the foundation for a range of other patterns to build upon.

CHAPTER 14

COLLECTIONS

While this book isn't meant to discuss every pattern you could think of, I find that a few are worth mentioning. That's why I'm also spending a chapter on collections: an alternative way of dealing with lists. We covered the built-in array functionality PHP has to offer and object-oriented programming, so collections that are a more functional way of solving problems fit well with these topics.

Maybe you haven't heard of collections before, so let's explain what they are first.

The core value of collections is that they allow for a more declarative programming style instead of an imperative one. The difference? An imperative programming style uses code to describe how something should be done; a declarative style describes the expected result.

Let's explain the difference further with an example. One of the best examples of a declarative language is SQL:

```
SELECT id, number
FROM invoices
WHERE invoice_date BETWEEN "2020-01-01" AND "2020-01-31";
```

An SQL query doesn't specify how data should be retrieved from a database, rather it describes what the expected result should be. In fact, SQL servers could apply different kinds of algorithms to solve the same query.

Compare the declarative style with an imperative one in PHP:

```php
$invoicesForJanuary = [];

foreach ($allInvoices as $invoice) {
    if (
        $invoice->paymentDate->between(
            new DateTimeImmutable('2020-01-01'),
            new DateTimeImmutable('2020-01-31')
        )
    ) {
        $invoicesForJanuary[] = [$invoice->id, $invoice->number];
    }
}
```

Our PHP implementation is more cluttered because it's concerned with the technical details of looping over a list of items and how filtering them should be done.

Collections aim to provide a more declarative interface. Here's what it'd look like:

```
$invoicesForJanuary = $allInvoices
    ->filter(fn (Invoice $invoice): bool =>
        $invoice->paymentDate->between(
            new DateTimeImmutable('2020-01-01'),
            new DateTimeImmutable('2020-01-31')
        )
    )
    ->map(fn (Invoice $invoice): array =>
        [$invoice->id, $invoice->number]
    )
```

A collection represents what would otherwise be a normal array, and provides lots of methods that have a more declarative approach. There's:

- `filter` to filter out results,
- `reject` being the counterpart of `filter`,
- `map` which transforms each item in the collection to something else,
- `reduce` which reduces to whole collection to a single result; and there's lots more.

You might get a functional programming vibe right now with functions such as `filter`, `map` and `reduce`. Collections do indeed find much of their inspiration in functional programming, but there are significant differences to real functional programming still: there's no guarantee our functions are pure, you can't compose functions out of others, and currying isn't really relevant in the context of collections. So while the collections API does have some similarities with functional programming, there also are significant differences.

Doing a deep dive in functional programming is outside this book's scope, especially since PHP isn't the best language to write real functional code with. I can highly recommend the book "Thinking Functionally in PHP" by Larry Garfield if you want to know more. In the book, Larry explains the core ideas of functional programming in the language you're familiar with. He also explains why you shouldn't use the approach in production PHP applications, but it's a great way to learn the functional programming concepts within a familiar language.

When you start thinking in collections, you'll start noticing many places in your code with loops or conditionals that could be refactored to collections. Refactoring to a declarative style can indeed make code easier to read and understand - an invaluable asset if you're working in large and complex code bases. Another book I'd highly recommend is called "Refactoring to Collections" By Adam Wathan (https://adamwathan.me/refactoring-to-collections/). In it Adam describes the ideas of collections more in depth: he explains all the building blocks needed to build collections, and gives lots of examples of using collections in the wild.

If you're looking for a production-ready implementation of collections, I'd highly recommend using `illuminate/collection`, which is the implementation also used by Laravel. Besides being a thorough and robust implementation, it's also very well documented: https://laravel.com/docs/8.x/collections.

CHAPTER 15
TESTING

Three, that's the number of chapters dedicated to type systems throughout this book. I realise though that not everyone in the PHP community likes to use types: some find them too verbose or think that they don't add enough value when you're not embracing static analysis. Personally, I'm in the "try to type everything" camp, but I respect the other opinion. I reckon there are cases where it's just simpler to embrace PHP's dynamic nature.

This chapter won't be about types though. It's about testing, so why do I bring up types again? I mentioned before that types reduce the number of tests you *should* write to be still able to know that the program works correctly. You can think of types as mini-tests on their own: built-into the language to ensure your code works correctly. Still, types can't possibly cover all business logic, which is why a proper test suite still has its use.

With testing being an integral part of your workflow, some questions can be answered: what kinds of tests are better? Which test framework should be used? I won't give any definitive answers. However, we will explore some options together in this chapter and discuss what makes a good test suite. It isn't my intention to give you a step-by-step guide on testing frameworks or testing strategies in this chapter. Instead, I want to show you the different options out there.

TYPES OF TESTS

There's unit testing, integration testing, acceptance testing, mutation testing, and more. I reckon it might be challenging to know where to start.

Some might say that unit testing is *the* way to go, while others will tell you that one integration test is worth a thousand unit tests. In turn, this argument is countered because integration tests are prone to breaking when they shouldn't. So, what's the best choice for you?

UNIT VS. INTEGRATION?

The definition of a unit test is that it should test a "unit" or "component" of your program in isolation. This isolation is often achieved by mocking dependencies so that a unit test wouldn't break if something changes within those dependencies. Using mocks comes with an added maintenance cost. There's also the question of how large a "unit" is: is it a function, class, module?

Integration tests, on the other hand, aim to test a group of units as a whole. They represent scenarios closer to real-life, such as a user submitting a form with all its side effects or a cron job running every hour. Integration tests will test how components work together.

Testing is a very careful balancing game. Even when you have 100% unit test coverage, there still might be bugs in how individual components work together or when certain input is passed. If, on the other hand, you only rely on integration tests, you'll have a hard time maintaining your test suite: it's bulkier, it's prone to break when making changes to the codebase, and a lot slower overall.

Instead of approaching the question from a theoretical point of view, let's try to address it in another way: what's our test suite's goal? Of course, it's to test whether the program works correctly, but there's more to it than that. One characteristic of a good test suite is that it can evolve and grow with your project. It's versatile. I've been confronted with test suites that didn't have this versatile nature: projects that started with proper test suites, only to find them withered away and ignored after a few months because of maintenance and code rot. As soon as we give up on our test suites, most hope is lost. So above all, you need a test suite that can evolve and is flexible, otherwise, it'll lose its value fairly quickly.

If you're building a small CMS-like website, you might get away with not having a test suite at all. But if you're working with a team of developers in a codebase with thousands upon thousands of lines of code, you'll get in trouble eventually. There is no way for any human to make changes in large codebases and ensure that everything still works without a proper test suite. Sooner or later, you'll deploy bugs in production; most will only be minor inconveniences, but one day there will be that one disastrous bug that makes you wish your code was tested properly.

So, find a testing strategy that can grow with your project. Any is better than none. If you and your team decide to invest in a fully unit-tested project that's fine but make sure the flow between components is also tested. If you're only investing in integration tests, be prepared to deal with the added maintenance cost in the long run, as well as a slow and bulky test suite.

I prefer to use unit tests in places where they add the most value. Places where there's the most complex decision logic and yet relatively little dependencies to be mocked. Domain code is usually a good candidate for a thorough unit test suite. An extensive and fast test suite is key to maintaining that code for years. On the other hand, infrastructure code, controllers, routing, middleware, request validation, etc., is served better from a few robust integration tests. I prefer to test these pieces of code as if my tests were the end-user or something similar. It allows you to write relatively small amounts of tests so that the added performance cost is relatively small.

ONE LEVEL HIGHER

When we go up a step, from unit to integration testing, our test suite covers more code at once, is generally slower to run, and more closely resembles how an end-user would interact with our code. There's another level above integration testing. It's again slower and bulkier but also resembles the end-user flow as close as possible: acceptance testing.

There are several strategies to do acceptance testing. We're all using at least one form: manually clicking through an application before shipping it; the very last verification before deploying to production. It's somewhat ironic that even with a full-blown test suite, we still feel the urge to do some manual testing to make sure everything works as expected for the end-user. On the other hand, it shouldn't be much of a surprise: our tests are written in code, which is not how the end-user interacts with our program. There's always a gap between how developers are testing and how the program will be used. And so we end up manually exploring the user-interface. There are better solutions. We're working with computers, so let's have them do all this work instead; let's automate acceptance testing.

While there are many choices and flavors of testing frameworks regarding unit and integration testing, there aren't many options to mimic actual user behaviour. Popular PHP acceptance testing frameworks like Codeception and Dusk all build upon the same software: Selenium. Selenium is a server that can open a web page in any given browser and do whatever a human would do: move the mouse, clicking buttons, filling in fields, etc. Selenium itself is written in Java, but you can communicate with it through API calls. There's an existing PHP package that takes care of this as well; it's called `php-webdriver/php-webdriver`.

Imagine the possibilities: writing scripts mimicking real-life users. Here's an example written in Dusk, which uses Selenium under the hood:

```php
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
          ->visit('/home')
          ->waitForText('Message');

    $second->loginAs(User::find(2))
           ->visit('/home')
           ->waitForText('Message')
           ->type('message', 'Hey')
           ->press('Send');

    $first->waitForText('Hey')
          ->assertSee('Name');
});
```

It is possible to automate almost everything we'd otherwise manually test. On the one hand, it's an excellent investment because such tests significantly reduce the amount of time spent manually testing, but on the other hand, these tests are even more prone to breaking. Imagine changing the text on a button, or a class or ID which is used by Selenium to find an element on the page: our test breaks. You could add special attributes to use as selectors, something like `data-selenium-login-button`, but that requires us to write lots of extra code in your HTML and manage that as well.

On top of that, these tests are much slower than any other kind: Selenium will run a headless browser in the background that needs to start and load the page, and Selenium needs to click around. Sure, it's much faster than doing it by hand, but they are super slow compared to unit or integration tests.

Again it's a balancing game: where to use Selenium tests and where not? Where do they add enough value given their maintenance cost and execution time, and where not? In my experience, it's best to use Selenium when you find yourself doing the same manual tests over and over again in the web browser. Selenium can be a great asset when testing, for example, an onboarding form or a JavaScript-heavy frontend tool.

TESTING TESTS

So there we have it: a well-balanced test suite that grows with our application - everything's great! There's one thing, though: how do we know whether our tests will test the right things? How do we know we're testing all the relevant code? Let me give you a simplified example:

```php
function specialFormat(string $input): string
{
    if ($input === 'a') {
        return str_repeat($input, 3);
    }

    return str_repeat($input, 5);
}
```

An example that doesn't make any real-life sense, but let's go with it for now. Say that this is our test:

```
public function test_special_format()
{
    $output = specialFormat('b');

    $this→assertEquals('bbbbb', $output);
}
```

It works, the test succeeds, but we've missed the `'a'` edge case! If we hadn't noticed that, our test suite would give a false sense of security. Since real-life code is more complex than this example, there's a very likely chance we'll miss edge cases here and there.

Again we can use our toolset to help us instead of trying to manage everything ourselves. One such tool is built-into phpunit: code coverage analysis. Such an analyser will look at our code while tests are running, note which parts are and aren't executed. It can even show a line-by-line analysis of untested parts of code. Such tools are great because we can run our existing test suite, and a percentage is returned about how much of our code was covered by it. It's an easy way to detect code that wasn't executed during our tests.

Besides detecting those areas, we can also analyse whether our tests are foolproof. This is where mutation testing comes in. A mutation test framework, like Infection PHP, will run our test suite several times, but it changes little things in the source code with every iteration. Such a change is called a mutation or a mutant. The reasoning is this: our test suite should be resilient enough to fail whenever such a mutation happens because a mutant that lives (meaning it's undetected and our test still runs) is a potential bug in our test suite.

Mutation tests will change little details such as changing a `<` to `<=` or `==`, `1` to `0`, `instanceof` checks to `true` or `false` etc. In the end, we're left with a report with tests that detected changes in our code base, and which did not. This results in a score, the "Mutation Score Indicator" — MSI for short. The higher our MSI, the more mutants were detected and killed, indicating a better test suite.

I realise there's more to tell about the frameworks and techniques I mentioned, but those are outside this book's scope. Luckily, most projects have great documentation, guiding you through the technical setup and explaining the mindset behind them.

Most important is a testing strategy that works for your project, one that's not ignored after two months of coding. Automated tests are very valuable to any professional project, so you should definitely invest in them: they do pay off!

CHAPTER 16

STYLE GUIDE

I've shown numerous code samples throughout this book, and you might have noticed me placing a bracket or comma in a place that you wouldn't. We're going to dedicate a whole chapter to this topic.

A style guide - a set of rules on how to visually structure your code - isn't only useful; it's a crucial part of professional software development. In fact, it's so crucial that the PHP community has a fixed set of rules which you can choose to follow, as well as automated tools to enforce those rules. However, before looking at specifics, let's discuss the use of a style guide. Isn't it all about personal preference of what you want your code to look like?

Let's take the following example, a constructor of an `InvoiceDTO`:

```
class InvoiceDTO
{
    public function __construct(string $number, ClientId $clientId, Date
        // …
    }
}
```

The problem is amplified because this is a book, but it's a problem no matter what size of screen you're working on. The argument list is too long for any developer to read and comprehend quickly, regardless of how much code is visible at once. If we're talking about code readability, there's more going on than you might think. Our job description is to write code, but if you'd look at your average workday, you're more likely to be reading code than writing it. Either you have to read documentation, recap what you've written the day before, find your way around legacy codebases, and so on. We're reading quite a lot of code day by day.

Just like writing code, reading requires your concentration because it puts a load on your brain. The official term is "cognitive load". If we manage to make our code more readable, we reduce cognitive load, allowing us to spend it on other things like writing code. The readability of a codebase has a significant impact on your day by day programmer life.

Also, think about your colleagues: the ones who have to maintain your legacy code ten years from now. Wouldn't you like to work in a clear codebase instead of an obfuscated one?

Back to our example. It's clear that the argument list is too long. We want to pull it more to the left, so that we can see this piece of code as a single block, not a long line. One solution could be to write something like this:

```php
public function __construct(string $number,
                            ClientId $clientId,
                            Date $invoiceDate,
                            Date $dueDate) {
    // …
}
```

There are two issues with this approach. First, the arguments are still rather far to the right side. If you're in web development, you probably know that people don't read websites; they rather scan them from left to right, top to bottom. We instinctively start by looking at the top left corner whenever we see something on a screen. On the other hand, with this formatting approach, we're pulling the argument list farther away from that point of focus.

The other problem has to do with refactoring. What if you decide to refactor this constructor to a static `create` method? You can see it breaks alignment:

```
public static function create(string $number,
                              ClientId $clientId,
                              Date $invoiceDate,
                              Date $dueDate): self {

    // …
}
```

It's clear that this isn't the ideal solution. Let's move on to another approach:

```
public function __construct(
    string $number, ClientId $clientId,
    Date $invoiceDate, Date $dueDate) {
    $this->number = $number;
    $this->clientId = $clientId;
    $this->invoiceDate = $invoiceDate;
    $this->dueDate = $dueDate;
}
```

Note that I've added the method body here - it's to make the problem clearer. I'm also deliberately not using promoted properties, for the sake of the example. In this case we've pulled the arguments more to the left, so that's great! However by doing so, we've introduced several points of focus, scattered throughout our code. Let's visualise that:

```
public function __construct(
    string $number, ClientId $clientId,
    Date $invoiceDate, Date $dueDate) {
    $this->number = $number;
    $this->clientId = $clientId;
    $this->invoiceDate = $invoiceDate;
    $this->dueDate = $dueDate;
}
```

There's the method start and end, there's the first and third arguments which align with the method body, and then there's the second and fourth arguments that don't align to anything. This makes the code even harder to read. So let's move on to another solution:

```
public function __construct(
    string $number,
    ClientId $clientId,
    Date $invoiceDate,
    Date $dueDate) {
    $this->number = $number;
    $this->clientId = $clientId;
    $this->invoiceDate = $invoiceDate;
    $this->dueDate = $dueDate;
}
```

The arguments are at the left again, they all align in a predictable way, and this seems like a great solution. There's one more issue. I can best visualise it by replacing all characters in our code with X's, in order to show the structure of it:

```
xxxxxx xxxxxxxx xxxxxxxxxxx(
    xxxxxx $xxxxxx,
    xxxxxxxx $xxxxxxxx,
    xxxx $xxxxxxxxxxx,
    xxxx $xxxxxxx) {
    $xxxx->xxxxxx = $xxxxxx;
    $xxxx->xxxxxxxx = $xxxxxxxx;
    $xxxx->xxxxxxxxxxx = $xxxxxxxxxxx;
    $xxxx->xxxxxxx = $xxxxxxx;
}
```

It's hard to see where the argument list ends and where the method body starts. There's that curly bracket opening the method body, but it is at the right side; not where our focus is by default! Colour coding helps us out a bit:

```
xxxxxx xxxxxxxx xxxxxxxxxxx(
    xxxxxx $xxxxxx,
    xxxxxxxx $xxxxxxxx,
    xxxx $xxxxxxxxxxx,
    xxxx $xxxxxxx) {
    $xxxx->xxxxxx = $xxxxxx;
    $xxxx->xxxxxxxx = $xxxxxxxx;
    $xxxx->xxxxxxxxxxx = $xxxxxxxxxxx;
    $xxxx->xxxxxxx = $xxxxxxx;
}
```

But still, we can do better. Let's add a structural, visual boundary between the argument list and the method body, just to make their difference as clear as possible. As it turns out, there is one right way where to place that curly bracket:

```
xxxxxx xxxxxxxx xxxxxxxxxxx(
    xxxxxx $xxxxxx,
    xxxxxxxx $xxxxxxxx,
    xxxx $xxxxxxxxxxx,
    xxxx $xxxxxxx
) {
    $xxxx->xxxxxx = $xxxxxx;
    $xxxx->xxxxxxxx = $xxxxxxxx;
    $xxxx->xxxxxxxxxxx = $xxxxxxxxxxx;
    $xxxx->xxxxxxx = $xxxxxxx;
}
```

On a new line! By doing so we've created a visual boundary that our eyes can scan for. Here's the end result:

```
public function __construct(
    string $number,
    ClientId $clientId,
    Date $invoiceDate,
    Date $dueDate
) {
    $this->number = $number;
    $this->clientId = $clientId;
    $this->invoiceDate = $invoiceDate;
    $this->dueDate = $dueDate;
}
```

Before moving on, I want to credit Kevlin Henney, who came up with this visualisation. He's a writer and programmer and has great talks about the readability of code: https://www.youtube.com/watch?v=ZsHMHukIlJY

Have you ever thought about code this way? There's a lot of detail and thought going into decisions like these, trying to optimise our code for when we read it. There's more to do to improve readability: choosing proper variable names or getting rid of noise such as redundant doc blocks. It all starts with a proper style guide, though.

Another strength of such a style guide is consistency within teams. Chances are you'll have to deal with code written by a colleague or vice-versa; it's best to have a consistent style guide to make understanding foreign code easier. Especially within a team of developers, we should embrace the style guide and follow it strictly, even when you or I don't agree with everything written in it. Consistency within the team transcends our personal preferences.

I've mentioned them before: the official PHP guidelines. It's official in the way that many large frameworks have agreed upon these guidelines. They call themselves the FIG (Framework Interpolation Group), and they made so-called "PSRs" (PHP Standards Recommendations). Many frameworks have since left the FIG, and it's significantly less relevant today, but their style guide still holds. It evolved together with the language over the years, so it's a relevant one up until this day. The most recent version is called PSR-12 and builds upon PSR-1, the original coding style.

There are rules on where to place brackets, name variables, structure classes, etc. I find it interesting to learn about these rules, but you can also use automation tools so that you don't have to think about them too much.

IDEs like PhpStorm have built-in support for these tools, and a popular one is called "PHP CS Fixer". It will analyse your code style and can automatically fix errors. It's based on a ruleset, for example, PSR-1, PSR-2, or PSR-12, but you can choose to add your own rules as well. The most important rule is to have sensible guidelines that your whole team agrees with.

Here's an example of such a CS Fixer config file for a Laravel project:

```
$finder = Symfony\Component\Finder\Finder::create()
    ->notPath('bootstrap/*')
    ->notPath('storage/*')
    ->notPath('vendor')
    ->in([
        __DIR__ . '/app',
        __DIR__ . '/tests',
        __DIR__ . '/database',
    ])
    ->name('*.php')
    ->notName('*.blade.php')
    ->ignoreDotFiles(true)
    ->ignoreVCS(true);

return PhpCsFixer\Config::create()
    ->setRules([
        '@PSR2' => true,
        'array_syntax' => ['syntax' => 'short'],
        'ordered_imports' => ['sortAlgorithm' => 'alpha'],
        'no_unused_imports' => true,
        'not_operator_with_successor_space' => true,
        'trailing_comma_in_multiline_array' => true,
        'phpdoc_scalar' => true,
        'unary_operator_spaces' => true,
        'binary_operator_spaces' => true,
        'blank_line_before_statement' => [
            'statements' => ['break', 'continue', 'declare', 'return',
                                    'throw', 'try'],
        ],
        'phpdoc_single_line_var_spacing' => true,
        'phpdoc_var_without_name' => true,
        'class_attributes_separation' => [
            'elements' => [
                'method',
            ],
        ],
        'method_argument_space' => [
            'on_multiline' => 'ensure_fully_multiline',
            'keep_multiple_spaces_after_comma' => true,
        ],
        'void_return' => true,
        'single_trait_insert_per_statement' => true,
    ])
    ->setFinder($finder);
```

My way of thinking about style guides is this: we need to make our code as readable as possible so that we lose as little time as possible reading it, in order to have time for more important things; things like writing code. I'd say automated tools like CS fixer are a must these days. You can run your formatter during CI or enforce it as a pre-commit hook. Whatever approach you choose: keep your code style consistent across your whole team, it'll make working in that code base a lot easier. It's the little details that make the difference!

PART III

PHP IN DEPTH

CHAPTER 17

THE JIT

There's been a lot of hype over the JIT in PHP 8. Some say it will revolutionize the PHP landscape. In this chapter we'll discuss what the JIT is about, how it can significantly impact PHP's performance, and look at some real-life benchmarks.

WHAT IS THE JIT?

First things first: "JIT" stands for "just in time". Its full name is "the just in time compiler". You might remember that, back in chapter 4, I explained the difference between a compiled language and an interpreted one: PHP falls in the latter category. So, where is this compiler coming from? Well, in fact: PHP has a compiler, but it's not one like you'd see in compiled languages: there is no standalone step to run it, and it doesn't generate a binary. The compiler is part of PHP's runtime engine: PHP code still needs to be compiled — translated — to machine code; it just happens on the fly when running PHP code.

A JIT compiler takes advantage of an interpreted language like this: it looks at the code while running and tries to identify the so-called "hot parts" of that code (parts that are executed more often than others). The JIT will take that source code and compile it on the fly to a more optimized, machine-friendly blob of code, which can be run instead. Interestingly, this is only possible in interpreted languages because programs which are fully compiled beforehand can't change anymore.

The mechanism that tries to identify hot parts is called a "monitor": it will look at the code, and monitor it while running. When it detects parts that are executed several times, it will mark them as "warm" or "hot". It can then compile the hot parts into optimized blobs of code. You can imagine there's a lot more complexity to this topic, but this book's goal isn't to explain the JIT compiler in depth. Instead, it's to teach you how and when to use it in PHP.

While this might sound great in theory, there are a few sidenotes to be made. First of all: monitoring code and generating JIT'ed code also comes with a performance cost. Luckily the benefits gained from the JIT outweigh that cost. At least, in some cases.

The second problem is more important: there aren't many hot parts to optimise in a regular MVC application serving web requests. When the JIT was still in its early days, a popular example was shared within the PHP community. It showed a fractal being generated with and without the JIT. The JIT'ed version was significant — tens times — faster. But let's be honest: we're very rarely generating fractal images in our PHP applications.

Even on the framework level, there's very little code that benefits from an optimized JIT'ed version. The main performance bottleneck in web applications isn't the code itself; it's I/O operations like sending and receive requests, reading data from the file system, or communicating with a database server.

We're speculating about the JIT's performance at this point. Let's look at some real-life benchmarks instead. I took a Laravel project with its production database and decided to benchmark it.

REAL-LIFE BENCHMARKS

Let's set the scene first. These benchmarks were run on my local machine. Consequently, they don't say anything about absolute performance gains; here, I'm only interested in making conclusions about the relative impact the JIT has on real-life code.

I'll be running PHP FPM, configured to spawn 20 child processes, and I'll always make sure to only run 20 concurrent requests at once, to eliminate any extra performance hits on the FPM level. Sending these requests is done using the following command, with ApacheBench:

```
ab -n 100 -c 20 -l localhost:8000
```

JIT SETUP

With the project in place, let's configure the JIT itself. The JIT is enabled by specifying the `opcache.jit_buffer_size` option in `php.ini`. If this directive is excluded, the default value is set to 0, and the JIT won't run.

```
opcache.jit_buffer_size=100M
```

You'll also want to set a JIT mode, which will determine how the JIT will monitor and react to hot parts of your code. You'll need to use the `opcache.jit` option. Its default is set to `tracing`, but you can set it to `function` as well:

```
opcache.jit=function
; opcache.jit=tracing
```

The `tracing` or `function` mode will determine how the JIT will work. The difference is that the `function` JIT will only monitor and compile code within the context of a single function, while the `tracing` JIT can look across those boundaries. In our real-life benchmarks, I'll compare both modes with each other. So let's start benchmarking!

ESTABLISHING A BASELINE

First, it's best to establish whether the JIT is working properly or not. We know from the RFC that it does have a significant impact on calculating a fractal. So let's start with that example. I copied the mandelbrot example from the RFC and accessed it via the same HTTP application I'll run the next benchmarks on:

```php
public function index()
{
    for ($y = -39; $y < 39; $y++) {
        printf("\n");

        for ($x = -39; $x < 39; $x++) {
            $i = $this->mandelbrot(
                $x / 40.0,
                $y / 40.0
            );

            if ($i == 0) {
                printf("*");
            } else {
                printf(" ");
            }
        }
    }
    printf("\n");
}

private function mandelbrot($x, $y)
{
    $cr = $y - 0.5;
    $ci = $x;
    $zi = 0.0;
    $zr = 0.0;
    $i = 0;

    while (1) {
        $i++;

        $temp = $zr * $zi;

        $zr2 = $zr * $zr;
        $zi2 = $zi * $zi;

        $zr = $zr2 - $zi2 + $cr;
        $zi = $temp + $temp + $ci;

        if ($zi2 + $zr2 > 16) {
            return $i;
        }

        if ($i > 5000) {
            return 0;
        }
    }
}
```

After running `ab` for a few hundred requests, we can see the results:

```
                                         requests/second (more is better)
Mandelbrot without JIT                                               3.60
Mandelbrot with tracing JIT                                         41.36
```

Great, it looks like the JIT is working! There's even a ten times performance increase!

If you want to verify whether the JIT is running, you can use `opcache_get_status()`, it has a `jit` entry which lists all relevant information:

```
print_r(opcache_get_status()['jit']);

// array:7 [
//   "enabled" => true
//   "on" => true
//   "kind" => 5
//   "opt_level" => 4
//   "opt_flags" => 6
//   "buffer_size" => 104857584
//   "buffer_free" => 104478688
// ]
```

Having verified it works as expected, let's move on to our first real-life comparison. We're going to compare running our code without the JIT, with the function JIT, and the tracing JIT, both are using 100MB of memory. The page we're going to benchmark shows an overview of posts, so there's some recursion happening. We're also touching several core parts of Laravel: routing, the dependency container, as well as the ORM layer.

```
                                      requests/second (more is better)
No JIT                                                           63.56
Function JIT                                                     66.32
Tracing JIT                                                      69.45
```

Here we see the results: enabling the JIT only has a slight improvement. In fact, running the benchmarks repeatedly, the results differ slightly every time: I've even seen cases where a JIT enabled run performs worse than the non JIT'ed version. Before drawing conclusions, let's bump the memory buffer limit. We'll give the JIT a little more room to breathe with 500MB of memory instead of 100MB.

```
                                      requests/second (more is better)
No JIT                                                           71.69
Function JIT                                                     72.82
Tracing JIT                                                      70.11
```

Here we have a case of the JIT performing worse. Like I said at the beginning of this chapter: I want to measure the relative impact of the JIT on real-life web projects. It's clear from these tests that sometimes there might be benefits, but it's in no way as noticeable as the fractal example we started with. I admit I'm not surprised by that. Like I wrote before: there's very little hot code to be optimised in real-life applications. We're only rarely doing fractal-like computations.

So am I saying there's no need for the JIT? Not quite. I think the JIT can open up new areas for PHP: areas where complex computations do benefit from JIT'ed code. I'm thinking about machine learning and parsers, stuff like that. The JIT *might* give the PHP community opportunities that didn't exist yet, but it's unclear to say anything with certainty at this point.

For example, there's a package called `nikic/php-parser` - a PHP implementation that can take PHP code and parse into structured data. This package is actually used by static analysis tools like Psalm, and it turns out this one does benefit a lot from the JIT. So even today, there's already advantages, just not when running a web application.

A COMPLEXITY TO MAINTAIN

Adding the JIT to PHP's core also comes with the added maintenance cost. Because the JIT generates machine code, you can imagine it's complex material for a higher level programmer to understand. Say there is a bug in the JIT compiler. For that, you need a developer who knows how to fix it. In this JIT's case, Dmitry Stogov did most programming on it, and there's only a handful of people who understand how it works.

With just a few people being able to maintain such a codebase today, the question whether the JIT compiler can be maintained properly seems justified. Of course, people can learn how the compiler works, but it is a complex matter nevertheless.
I don't think this should be a reason to ditch the JIT, but the cost of maintenance should still be carefully considered. In the first instance, by the ones maintaining the code and the userland community, who should also be aware that some bugfixes or version updates might take longer than what we're used to right now.

DO YOU WANT IT?

If you think that the JIT offers little short-term benefits for your web applications, I'd say you're right. It's clear that its impact will be minimal at best.

Despite that, we should remember that the JIT can open many possibilities for PHP to grow, both as a web language and a more generally purposed language. So the question that needs answering: is this possibly a bright future worth the investment today?

Time will tell.

CHAPTER 18

PRELOADING

Another core feature focused on performance is preloading, added in PHP 7.4. It's a feature that can improve your code's performance significantly, by loading cached PHP files into memory on server startup. Preloading is done by using a dedicated preload script — which you have to write yourself or generate. The script is executed once on server startup, and all PHP files loaded from within that script will be available in-memory for all following requests. In this chapter, I'll show you how to set up and use preloading and share benchmarks as I did with the JIT.

Let's start by discussing preloading in depth.

OPCACHE, BUT MORE

Preloading itself is built on top of opcache, but it's not exactly the same. Opcache will take your PHP source files at runtime, compile them to "opcodes", and store those compiled files on disk. You can think of opcodes as a low-level representation of your code that can be easily interpreted at runtime. The next time a cached file is requested, the translation step can be skipped, and the file can be read from disk. In practice a simple PHP instructions like this one:

```
echo 1 + 2;
```

Would be translated to opcodes like so:

```
line     #        op            fetch          ext  return  operands
----------------------------------------------------------------------
   6     0        ADD                               -0      1,2
         1        ECHO                                      ~0
   7     2        RETURN                                    1
```

Opcache already speeds up PHP significantly, but there's more to be gained. Most importantly: opcached files don't know about other files. If you've got a class `Order` extending from a class `Model`, PHP still needs to link them together at runtime.

So this is where preloading comes into play: it will not only compile source files to opcodes but also link related classes, traits, and interfaces together. It will then keep this "compiled" blob of runnable code — that is: code usable by the PHP interpreter — in memory. When a request arrives at the server, it can now use parts of the codebase that were already loaded in memory, without any overhead.

Preloading improves performance even more compared to opcache, since it already links files, doesn't have to read cached opcodes from disk, and doesn't deal with invalidating cache. Once a file is cached, it's there to stay until the server restarts.

So, which files exactly can be preloaded? And how do you do that?

PRELOADING IN PRACTICE

For preloading to work, you — the developer — have to tell the server which files to load. This is done with a simple PHP script, which can include other files. The rules are simple:

- You provide a preload script and link it in your `php.ini` file using `opcache.preload`
- Every PHP file you want to be preloaded should be loaded within that script. You can either use `opcache_compile_file()` or `require_once` to do so.

Say you want to preload the Laravel framework. Your script will have to loop over all PHP files in the `vendor/laravel` directory and include them individually.

Here's how you'd link to this script in `php.ini`:

```
opcache.preload=/path/to/project/preload.php
```

And here's a dummy implementation of that preload file:

```
$files = /* An array of files you want to preload */;

foreach ($files as $file) {
    opcache_compile_file($file);
}
```

WARNING: CAN'T PRELOAD UNLINKED CLASS

Hang on; there's a caveat! For files to be preloaded, their dependencies — interfaces, traits, and parent classes — must also be preloaded. If there are any problems with the class dependencies, you'll be notified of it on server start up:

```
Can't preload unlinked class Illuminate\Database\Query\JoinClause:
Unknown parent Illuminate\Database\Query\Builder
```

This is the linking part I discussed earlier: the dependencies of preloaded files must also be loaded; otherwise, PHP can't preload them. This isn't a fatal error, by the way - your server will start up just fine - but it indicates that not all files you wanted to preload were able to do so.

Luckily, there's a way to ensure all dependencies of a PHP file are loaded as well: instead of using `opcache_compile_file` you can use `require_once`, and let the registered autoloader (probably composer's) take care of the rest.

```
$files = /* All files in eg. vendor/laravel */;

foreach ($files as $file) {
    require_once($file);
}
```

There are still some caveats. If you're trying to preload Laravel, for example, some classes within the framework have dependencies on other classes that don't exist yet. For example, the filesystem cache class `\Illuminate\Filesystem\Cache` has a dependency on `\League\Flysystem\Cached\Storage\AbstractCache`, which might not be installed in your project if you're never using filesystem caches.

You might run into "class not found" errors trying to preload everything. And the only solution is to skip those files from preloading. Luckily in a default Laravel installation, there's only a handful of these classes, which can easily be ignored. For convenience, I wrote a little preloader class to make ignoring files easier, and here's what it looks like:

```
class Preloader
{
    private array $ignores = [];

    private static int $count = 0;

    private array $paths;

    private array $fileMap;

    public function __construct(string ...$paths)
    {
        $this->paths = $paths;

        // We'll use composer's classmap
        // to easily find which classes to autoload,
        // based on their filename
        $classMap = require __DIR__ .
            '/vendor/composer/autoload_classmap.php';

        // We flip the array, so that file paths are the array keys
        // With it, we can search the corresponding class by its path
        $this->fileMap = array_flip($classMap);
    }
    public function paths(string ...$paths): Preloader
    {
        $this->paths = array_merge(
            $this->paths,
            $paths
        );

        return $this;
    }

    public function ignore(string ...$names): Preloader
    {
        $this->ignores = array_merge(
            $this->ignores,
            $names
        );

        return $this;
    }

    public function load(): void
    {
        // We'll loop over all registered paths
        // and load them one by one
        foreach ($this->paths as $path) {
            $this->loadPath(rtrim($path, '/'));
        }

        $count = self::$count;

        echo "[Preloader] Preloaded {$count} classes" . PHP_EOL;
    }
```

```php
    private function loadPath(string $path): void
    {
        // If the current path is a directory,
        // we'll load all files in it
        if (is_dir($path)) {
            $this->loadDir($path);

            return;
        }

        // Otherwise we'll just load this one file
        $this->loadFile($path);
    }
```

```php
private function loadDir(string $path): void
{
    $handle = opendir($path);

    // We'll loop over all files and directories
    // in the current path,
    // and load them one by one
    while ($file = readdir($handle)) {
        if (in_array($file, ['.', '..'])) {
            continue;
        }

        $this->loadPath("{$path}/{$file}");
    }

    closedir($handle);
}

private function loadFile(string $path): void
{
    // We resolve the classname from composer's autoload mapping
    $class = $this->fileMap[$path] ?? null;

    // And use it to make sure the class shouldn't be ignored
    if ($this->shouldIgnore($class)) {
        return;
    }

    // Finally we require the path,
    // causing all its dependencies to be loaded as well
    require_once($path);
```

```php
        self::$count++;

        echo "[Preloader] Preloaded `{$class}`" . PHP_EOL;
    }

    private function shouldIgnore(?string $name): bool
    {
        if ($name === null) {
            return true;
        }

        foreach ($this->ignores as $ignore) {
            if (strpos($name, $ignore) === 0) {
                return true;
            }
        }

        return false;
    }
}
```

By adding this class in the same preload script, we're now able to load the whole Laravel framework like so:

```php
// ...

(new Preloader())
    ->paths(__DIR__ . '/vendor/laravel')
    ->ignore(
        \Illuminate\Filesystem\Cache::class,
        \Illuminate\Log\LogManager::class,
        \Illuminate\Http\Testing\File::class,
        \Illuminate\Http\UploadedFile::class,
        \Illuminate\Support\Carbon::class,
    )
    ->load();
```

So keep in mind that every time you make a change to the preload script, or any of its preloaded files, you'll have to restart the server. I don't mean physically rebooting the whole server, though; restarting `php-fpm` is enough. If you're on a Linux machine it's as easy as running `sudo service php8.0-fpm restart`. Replace `8.0` with the version you are on.

DOES IT WORK?

With all those preloaded files, are we sure they were correctly loaded? You can simply test it by restarting the server, and dumping the output of `opcache_get_status()` in a PHP script. You'll see a key called `preload_statistics`, which will list all preloaded functions, classes, and scripts, as well as the memory consumed by the preloaded files.

There's one more important thing to mention about the operations side when using preloading. You already know that you need to specify an entry in `php.ini` for preloading to work. If you're using shared hosting, you won't be able to freely configure PHP whatever way you want. In practice, you'll need a dedicated (virtual) server to be able to optimise the preloaded files for a single project. So keep that in mind.

On to the most important question: does preloading improve performance? Well, let's benchmark it! Like with the JIT, I'll do a practical benchmark measuring relative results and measure a real-life project. It's important to know whether preloading is worth your time in your own projects, and not just in a theoretical benchmark. This project is the same Laravel project as in the previous chapter: it will again do some database calls, view rendering, etc.

Let's set the stage.

PRELOADING SETUP

Since I'm mostly interested in the relative impact preloading has on my code, I decided to run these benchmarks on my local machine using Apache Bench. I'll be sending 5000 requests, with 50 concurrent requests at a time. The webserver is Nginx, using PHP-FPM. Because there were some bugs in early versions of preloading, I'm only able to successfully run these benchmarks as early as PHP 7.4.2.

I'll be benchmarking three scenarios: one with preloading disabled, one with all Laravel and application code preloaded, and one with an optimised list of preloaded classes. The reasoning for that latter one is that preloading also comes with a memory overhead. If we're only preloading "hot" classes — classes that are used very often — we might be able to find a sweet spot between performance gain and memory usage.

PRELOADING DISABLED

We start php-fpm without preloading enabled and run our benchmarks:

```
./php-7_4_2/sbin/php-fpm --nodaemonize

ab -n 5000 -c 50 -l localhost:8000
```

These were the results: we're able to process 64.79 requests per second, with an average time of 771ms per request. This is our baseline scenario, where we can compare the next results to this one.

NAIVE PRELOADING

Next, we'll preload all Laravel and application code. This is the naive approach because we're never using all Laravel classes in a request. We're preloading many more files than strictly needed, so we'll have to pay a penalty for it. In this case, 1165 files and their dependencies were preloaded, resulting in 1366 functions and 1256 classes to be included.

Like I mentioned before, you can read that info from `opcache_get_status()`:

```
opcache_get_status()['preload_statistics'];
```

Another metric we get from `opcache_get_status()` is the memory used for preloaded scripts. In this case it's 17.43 MB. Even though we're preloading more code than we actually need, naive preloading already has a positive impact on performance.

```
                       requests/second          time per request
No preloading                    64.79                     771ms
Naive preloading                 79.69                     627ms
```

You can already see a performance gain: we can manage more requests per second, and the average amount of time to process one request has dropped by ±20%.

OPTIMISED

Finally, we want to compare the performance gain when we're using an optimised preloading list. For testing purposes, I started the server without preloading enabled and dumped all classes that are used within that request:

```
get_declared_classes();
```

Next, I only preloaded these classes (427 in total). Together with all their dependencies, this makes for 643 classes and 1034 functions being preloaded, occupying about 11.76 MB of memory.

These are the benchmark results for this setup:

```
                            requests/second            time per request
No preloading               64.79                                 771ms
Naive preloading            79.69                                 627ms
Optimised preloading        86.12                                 580ms
```

That's around a 25% performance gain compared to not using preloading and an 8% gain compared to using the naive approach. There's a flaw with this setup though, since I generated an optimised preloading list for one specific page. In practice, you would probably need to preload more code, if you want all your pages covered.

Another approach could be to monitor which classes are loaded how many times over several hours or days on your production server and compile a preload list based on those metrics. A package that does this is called `darkghosthunter/preloader`. It's definitely worth checking out.

It's safe to say that preloading — even using the naive "preload everything" approach — has a positive performance impact, also on real-life projects built upon a full-blown framework. However, there's still an important side note to be made. Real-life applications will most likely not experience a 25% increase in performance. That is because they do many more things than just booting a framework. An important thing I can think about is I/O: communicating with a database server, reading and writing to the filesystem, integrating with third party services, etc. So while preloading can optimise the booting part of your code, there still might be other areas that have a much larger impact on performance. How much exactly there is to be gained will depend on your code, server, and framework you're using. I'd say go try it out, and don't forget to measure the results.

CHAPTER 19

FFI

PHP has a rich ecosystem of extensions, and most of them can be installed using a simple `pecl install …`. Many of those extensions provide their functionality by integrating with third-party libraries. Take, for example, an extension like imagick: it's written in C and exposes the underlying ImageMagick library as functions to PHP. The imagick extension itself doesn't provide the image processing functionality; it only provides a bridge between PHP and ImageMagick.

FFI — "foreign function interface" in full — makes the process of exposing such underlying libraries "easier". I use quotes because you still need to understand how those libraries work under the hood, but you don't need to write and distribute a dedicated extension anymore. FFI makes this process easier because it allows you to write PHP where you'd otherwise had to implement an extension in C.

In other words: if the required libraries are present on your server or local development environment, you can install extension-like functionality by using composer and downloading only PHP code. You can integrate with any language, as long as it can compile to a shared library, which are `.so` files on Unix and Linux systems or `.dll` files on Windows.

To get a feeling of how FFI code would look, let's look at an example taken from the official FFI documentation page:

```
$ffi = FFI::cdef("
    typedef unsigned int time_t;
    typedef unsigned int suseconds_t;

    struct timeval {
        time_t      tv_sec;
        suseconds_t tv_usec;
    };

    struct timezone {
        int tz_minuteswest;
        int tz_dsttime;
    };

    int gettimeofday(struct timeval *tv, struct timezone *tz);
", "libc.so.6");

$tv = $ffi->new("struct timeval");
$tz = $ffi->new("struct timezone");

var_dump($ffi->gettimeofday(
    FFI::addr($tv),
    FFI::addr($tz))
);

var_dump($tv->tv_sec);

var_dump($tz);
```

This short example already shows the complexity of implementing FFI: you need to load header definitions into the `$ffi` variable, and use the exposed structs and functions from that variable. Furthermore, you can't just copy this code snippet and run it via a web request: FFI is only enabled by default in the CLI and the preloading script because of security reasons. Imagine if you'd be able to manipulate FFI code from a web request, and gain access to all system binaries and potentially exploit security flaws within them. Mind you: you can enable FFI in web requests, but you should probably think twice before doing so. Finally, you need opcache enabled for FFI to work, so if you're running this script via the CLI, you'll have to enable opcache specifically since it's disabled by default over there.

HEADER FILES

Header files document what functionality is available in the shared library you're integrating with. It's common practice in C programming but unknown in PHP.

A header file as a list of definitions for structs and functions, without their actual implementation; something like interfaces for classes. Unfortunately, PHP's FFI doesn't support all modern-day header syntax. So while the original header files are available for whatever library you want to integrate with, chances are you'll need to re-write them specifically for PHP to understand.

As you can see, there's more to FFI programming than writing simple PHP code. On the other hand, even though FFI isn't as performant as a compiled C extension — it has the overhead of having to parse the header files — it can offer a significant performance improvement compared to implementing the same functionality directly in PHP. In summary: FFI has potential but isn't a plug-and-play solution.

That brings us to the question: what use cases are there for FFI? The first ones that come to mind are the CPU-heavy tasks that are way more efficient to do in low-level languages such as C or Rust than in PHP. So I asked around the community who's using FFI, and there are indeed a handful of people using it. One example is to parse large data sets efficiently. There's also the use case of shipping extension-like code with composer.

Another interesting example is changing the PHP runtime itself so that you don't need to add the opening `<?php` tag in your PHP files anymore. It's interesting to see how much becomes possible when PHP can talk to other languages directly. One more: there's a project created by Anthony Ferrara called `php-compiler`. It can compile (a limited set) of PHP code and generate a binary from that code; it's actually written *in* PHP using FFI.

INTERESTED IN MORE?

There's an interesting GitHub repository that lists examples of using FFI in PHP called `gabrielrcouto/awesome-php-ffi`.

Most FFI projects are still very young, though. FFI is supported as of PHP 7.4, so it was only a year old when this book was released. Overall, FFI isn't widely used yet. There's nothing wrong with that: it's a rather niche solution that not many PHP developers have to deal with. Still, FFI has potential, but it probably needs some polishing before it'll be widely adopted. So let's look forward to how it will evolve in the years to come!

CHAPTER 20

INTERNALS

In this book, we've focussed a lot on several syntax-related and core features. Where do these features come from? Who decides what gets added to the language or not? I've mentioned the "internals" group once or twice before — are they calling the shots?

There's indeed a group of core developers who write and maintain all code that makes PHP run. Additionally, there are also extension developers, documentation maintainers, release managers and other PHP project roles. All those people together form the group that decides what gets added to PHP and what doesn't. They are the ones who have voting rights.

Those voting rights are used at the end of every RFC — a "request for comments". An RFC is a document that describes a new feature or change to the language; it's discussed for a period (two weeks being the minimum), and is voted on in the end. If an RFC has a 2/3 majority of votes in favour, it's considered accepted, and the feature is added to the language.

PHP RFC: ATTRIBUTES V2

As an example, here's a (shortened version) of the attributes RFC. You'll notice the old attribute syntax used in this RFC, it was changed only later.

Version: 0.5
Date: 2020-03-09
Author: Benjamin Eberlei (beberlei@php.net), Martin Schröder
Status: Implemented
Target: 8.0
First Published at: http://wiki.php.net/rfc/attributes_v2
Implementation: https://github.com/php/php-src/pull/5394

Large credit for this RFC goes to Dmitry Stogov whose previous work on attributes is the foundation for this RFC and patch.

INTRODUCTION

This RFC proposes Attributes as a form of structured, syntactic metadata to declarations of classes, properties, functions, methods, parameters, and constants. Attributes allow to define configuration directives directly embedded with the declaration of that code.

Similar concepts exist in other languages named Annotations in Java, Attributes in C#, C++, Rust, Hack, and Decorators in Python, Javascript.

So far, PHP only offers an unstructured form of such metadata: doc-comments. But doc-comments are just strings, and to keep some structured information, the @-based pseudo-language was invented inside them by various PHP sub-communities.

On top of userland use-cases, there are many use-cases for attributes in the engine and extensions that could affect compilation, diagnostics, code-generation, runtime behavior, and more. Examples are given below.

The widespread use of userland doc-comment parsing shows that this is a highly demanded feature by the community.

PROPOSAL

Attribute Syntax

Attributes are a specially formatted text enclosed with "<<" and ">>" by reusing the existing tokens T_SL and T_SR.

attributes may be applied to many things in the language:

functions (including closures and short closures) classes (including anonymous classes), interfaces, traits class constants class properties class methods function/method parameters

Examples:

```php
<<ExampleAttribute>>
class Foo
{
    <<ExampleAttribute>>
    public const FOO = 'foo';

    <<ExampleAttribute>>
    public $x;

    <<ExampleAttribute>>
    public function foo(<<ExampleAttribute>> $bar) { }
}
```

...

In theory, everyone is allowed to make an RFC. You don't need to have voting rights. Every userland developer - you and I - can submit an idea up for discussion. There is an important side note, though: if you manage to get an RFC accepted, it still needs an implementation. In fact, a working implementation before an RFC is voted on is usually an advantage. So while you technically could ask someone to implement your feature after your RFC is accepted, it is an asset being able to code it yourself or work closely with someone who can implement the changes beforehand.

Having described the RFC process in a few paragraphs, it might seem pretty simple and straightforward. Surely, most of the time, sensible RFCs get approved. But there are exceptions where an RFC can become very controversial, with lengthy discussions as a result. A recent example is the attributes RFC, the one I showed an excerpt from earlier. The originally accepted RFC was already the seventh attempt to add annotation-like behaviour in PHP. Those previous attempts happened over several years, the earliest dating back to August 2010!

There are several reasons it took ten years for attributes to be added. First, there's the discussion of what attributes should and shouldn't do: the more functionality they support, the more complex the implementation. There's the discussion on what attributes should look like, which syntax should be used. These kinds of questions lengthen discussion periods and cause a few failed RFCs along the way. Over the years though, the idea can be polished further, and some compromises will probably have to be made.

One such compromise is the attribute's syntax, which took two additional RFCs after attributes were accepted to decide upon. There were options like: `@`, `@@`, `<<>>`, `#[]`, and a few more. In the end, `#[]` was chosen to be the best option, given that `@` wasn't available since it's already the error silencing operator. Months of discussion were needed to arrive at an agreed upon syntax finally. Lots of good arguments have been made in favour of and against all options.

All of these discussions happen in the open. You can follow along just fine by joining the internals mailing list. There's a website called externals.io which exposes the mails on a web page, making them easier to read. I think it's a good thing to follow this list, to know a little bit about the process of how PHP is designed. Even if you never intend to contribute to PHP's core, it's still good to know how the tools you're using are evolving and how userland developers can provide valuable feedback within that process.

We've also seen more and more core contributors listening to userland developers over the past years. Some people cling to the bias that core developers don't care about real-life PHP, but that's definitely not true anymore. There has been lots of interaction between core developers and userland developers on social media, GitHub, and the internals list. Several key figures in PHP's development reach out to userland developers to get a better sense of what's needed in real-life PHP projects.

There's room for improvement, but PHP has been maturing over the past decade. It also shows in the internals process.

CHAPTER 21

TYPE VARIANCE

Earlier in this book, we talked about type systems and their value to a programming language. We also spoke about recent changes to PHP's type system and how it's made more flexible by adding proper variance support. Type safety in programming languages is such an interesting topic that I decided to spend a dedicated chapter on it.

When we talked about static analysis, I showed the example of an `RgbValue` class to represent "integer values between 0 and 255". This type ensures valid input and allows us to remove redundant input validation. It looked like this:

```
class RgbValue extends MinMaxInt
{
    public function __construct(int $value)
    {
        parent::__construct(0, 255, $value);
    }
}
```

Using `RgbValue` as a type, we're promised we'll only be passed input to our function that's within the subset it describes. Subset is the interesting word here. All types can be thought of as a filter on all available input. `RgbValue` represents a subset of positive integers, which represents a subset of all integers, which in turn is a subset of all scalar values (integers, floats, strings, ...), which are a subset of everything. Can you see the mental image of some kind of inheritance chain forming? `RgbValue > positive ints > all ints > scalar values > everything`.

Here's another example: When we talked about PHP's type system, we had a class called `UnknownDate`; it represented a missing date and allowed us to use the null object pattern. `UnknownDate` is a subtype of `Date`, which is a subtype of `object`, which again is a subtype of everything.

Speaking of that example, let's revisit it. Here's the `Invoice` interface:

```php
interface Invoice
{
    public function getPaymentDate(): Date;
}
```

This interface represents the `Invoice` type, and it comes with a rule: it has a `PaymentDate`. This interface *promises* that any object implementing `Invoice` will return a valid `Date` object when calling `getPaymentDate()`. So what about `PendingInvoice`? We decided to make it return an `UnknownDate`. This raises the question: does the promise made by the `Invoice` interface still hold?

```php
class PendingInvoice implements Invoice
{
    public function getPaymentDate(): UnknownDate
    {
        return new UnknownDate();
    }
}
```

It sure does! Since all unknown dates are a *subset* of dates, it means that whenever an `UnknownDate` is returned, we're always sure it's also a `Date`. This is what variance describes: type definitions that change during inheritance, which *still* fulfill the parent's original promise. In the case of return types, we're allowed to further specify them during inheritance, which is called "covariance". In the case of argument types, the opposite is true.

It isn't easy to come up with an example that makes sense for contravariant types — the opposite of covariant types. They are only rarely used in PHP: they make most sense combined with generics, which PHP doesn't support.

Still, there are a few edge cases where contravariance might be useful. Let's consider an example:

```
interface Repository
{
    public function retrieve(int $id): object;

    public function store(object $object): void;
}

interface WithUuid
{
    public function retrieve(string $uuid): object;
}
```

The `Repository` interface describes a simplified repository: a class that can retrieve and store objects from a data store. The repository assumes all IDs will be integers.

There's also a `WithUuid` interface though, one that allows passing textual UUIDs instead of numbers. Next let's implement the `OrderRepository`:

```php
class OrderRepository implements Repository, WithUuid
{
    public function retrieve(int $id): object { /* … */ }

    public function store(object $object): void { /* … */ }
}
```

Here we see a problem: we can't use `int $id` in the `retrieve` method, because it violates the contract specified by `WithUuid`. If we'd use `string $uuid`, there would be the same problem but the other way around.

It's those edge cases that make contravariant types useful: PHP allows us to widen argument types, in order to fulfil both promises: one made by `Repository`, and one made by `WithUuid`. Thanks to PHP 8's union types, we can write `retrieve`'s implementation like so:

```php
class OrderRepository implements Repository, WithUuid
{
    public function retrieve(int|string $id): object { /* … */ }

    // …
}
```

And this code works! Of course, we need to manually deal with managing both strings and integers in our retrieve method now; still, it's good to have the option available when there's no way around the problem.

So return types are covariant, argument types contravariant. What about typed properties? They are invariant, which means you're not allowed to change property types during inheritance. It is explained clearly in the typed properties RFC why that is: *"The reason why property types are invariant is that they can be both read from and written to. The change from `int` to `?int` implies that reads from the property may now also return `null` in addition to integers. The change from `?int` to `int` implies that it is no longer possible to write `null` to the property. As such, neither contravariance nor covariance are applicable to property types."*

Before PHP 7.4, you weren't allowed to widen or narrow types, even though it would be technically correct. And while it might seem like a minor change, it's actually one that makes PHP's type system much more flexible to use.

CHAPTER 22

ASYNC PHP

When discussing the MVC pattern, we looked at why it was a good fit for PHP, especially since PHP has a very clean request/response cycle. Every time a new request arrives, a new process starts and boots our PHP application from scratch. This characteristic makes PHP so easy to start with: you don't need any compiler, you don't need to deal with shared state across requests or worry about memory management; it's super simple.

Simplicity isn't always preferred, though. There's a community within PHP that has started to grow in popularity over the years: the async community. It's worth mentioning them in this book because there are more and more use cases for async PHP lately. Modern PHP isn't only used to power blogs or small company websites; it's also used to build large web applications at scale, and there are cases where async adds significant value.

To be clear upfront: I won't be able to explain async programming in depth in this chapter. Explaining async requires a book on its own. The goal of this chapter is to let you know about the options out there for PHP. So I'll assume you know some async related vocabulary. No worries if that's not the case though, it'll still be an interesting read, but you probably will need to do some follow-up research if you're interested in the topic. In that case, I'd advise you to look at the JavaScript community, which has embraced async programming like no other language and has great resources to learn the async mindset.

WHAT'S ASYNC?

With "async PHP" we mean that one PHP process can handle several things at once. You can use threads or create child processes, each dedicated to a single task. The parent process, in turn, will monitor and manage all those children. For example, you could create a child process for every request that comes in and share the loaded and booted framework from the parent process instead of booting it from scratch on every request.

Async is more than managing child processes, though. Think of dealing with I/O, such as reading files or communicating with external services. Instead of performing a database query and waiting for the result before executing the next one, you could execute all your queries in parallel. Say you have ten queries that each take a second to execute; you could potentially run all of them in one second instead of ten.

In short, async programming can boost performance significantly if used correctly.

Most MVC frameworks aren't optimised for the async mindset though: they assume they'll have to boot from scratch every time a new request comes in. In turn, they try to optimise parts of that process with caching and by compiling configuration into an optimised production-ready version. You might instinctively think that if you're able to skip the "boot phase" of your framework, you'd see massive performance improvements; that's not entirely the case, though.

You see, having a "shared application state" and handling requests in parallel within PHP wouldn't make a significant difference: your webserver is already running several processes to serve requests in parallel. On top of that, you could use preloading to have the application booted and ready in memory. The biggest performance win comes from handling I/O asynchronously, which most frameworks aren't optimised to do. They aren't built with the async mentality. That's fine since async PHP adds a layer of complexity that's overkill for many projects.

Given that, you shouldn't expect groundbreaking performance gains by simply combining an MVC framework and an async framework like Amp or ReactPHP - there's more to it than that. There are more interesting use cases for async than the average MVC application.

Think about complex APIs that need to do lots of realtime heavy calculations. When I asked around the community for people who had real-life use cases for async PHP, someone told me they used Swoole to run their API. Swoole enabled them to cache data statically across processes to reduce database I/O and do complex computations in parallel. All in all, they were able to serve 8000 requests per second instead of 600.

ASYNC FRAMEWORKS

I mentioned Swoole just now and also talked about Amp (Amphp in full) and ReactPHP. Those are the three most popular async frameworks used today. Amp and ReactPHP are implemented in PHP itself. You can download them in your project with composer, and you're set. Swoole, on the other hand, is a PHP extension you'll have to install on your system. While ReactPHP describes itself as a framework for "Event-driven, non-blocking I/O with PHP"; Swoole is a "Coroutine based Async PHP Programming Framework".

Besides full-blown async frameworks, there's also the popular Guzzle HTTP library, which can perform HTTP requests in parallel. This is a much simpler and isolated approach to async programming, but there are several use cases for it as well. Say you're loading data from a paginated API; you could potentially load several pages at once instead of loading them one by one.

Each framework has its target audience with its own needs; all are popular choices and used in production today.

Another real-life example that I came across was a platform that regularly sent out 100,000 push notifications using AWS Lambda. Instead of spawning 100,000 Lambdas, they chunked the data into groups of 500 and sent each group in parallel to 200 Lambdas in total. They used Guzzle to perform 200 HTTP requests in parallel and waited for all of them to finish. Each Lambda in turn, would send the 500 push notifications in parallel as well and finished in under a minute. All in all, it took 2 minutes to send out 100,000 push notifications, thanks to async programming.

People sometimes compare PHP to NodeJS and mention its lack of async capabilities, but the opposite is true. There are lots of options for async programming in PHP. They are used in production today for a variety of use cases. PHP differs with JavaScript in that it has no built-in support for async programming such as `async`, `await` or promises. There has been interest to add `libuv` in PHP's core, which is the same library that powers Node's asynchronous capabilities. There haven't been any attempts to integrate it, but chances are PHP will have a built-in async engine one day.

Until then, there are already great and battle-tested solutions out there that can be used today.

CHAPTER 23

EVENT DRIVEN DEVELOPMENT

Event sourcing, CQRS, event-driven; these mystical terms can seem daunting if you've never worked in an event-driven system before. And even if you have, there are lots of opinions, theories, and patterns. It still might seem like a very complex matter overall. Event-driven systems aren't the solution to all problems. While they add a layer of flexibility, they also remove simplicity and explicitness.

At its core, event-driven development isn't all that difficult. It's the patterns that build upon a simple concept that make it more difficult, but also more powerful. This power is often needed in complex and large applications, and indeed: PHP is often used to build such applications. I wanted to dedicate a chapter on the topic since you'll probably have to deal with some form of event-driven systems in your career. It's important to have some background information.

So let's start with the basics.

The idea of an event-driven system is that you step away from micro-managing the program flow and instead allow individual components to react whenever something happens. An example: instead of having a single function or service that manages the "creation of an invoice", there can be several small services, each handling a part of the invoice creation process. The starting point is the invoice being created; next there's a service that generates a PDF and saves it on the filesystem; and one that sends an email to the customer notifying them about a pending invoice.

SERVICE? OBJECT? FUNCTION?

I've used three terms in one paragraph to describe "a piece of code that reacts when something happens". You might even think about calling them "micro services". For now, we won't focus on the technicalities of how these services communicate with each other and assume they are, in fact, simple objects in the same codebase, running on the same server; I'll call them plain old "services" from here on out.

By the way, have you noticed how Alan Kay's vision of what "objects" are perfectly fits this model? I like it when things come together!

These services don't necessarily need to know about each other: they react whenever something happens in the system. This "something" is called an "event".

From a technical point of view, all an event-driven system needs is an event bus: it's the central place that knows about all services listening for events; we could also call them "event subscribers". Whenever an event happens, the event bus is notified, which in turn will notify all relevant subscribers. It's not difficult at all to program a simple event bus yourself. For example, here's an implementation in only a few lines of code:

```php
interface Subscriber
{
    public function handles(object $event): bool;

    public function handle(object $event): void;
}
```

```php
class EventBus
{
    private array $subscribers = [];

    public function addSubscriber(Subscriber $subscriber): self
    {
        $this->subscribers[] = $subscriber;
    }

    public function dispatch(object $event): void
    {
        foreach ($this->subscribers as $subscriber) {
            if (! $subscriber->handles($event)) {
               continue;
            }

            $subscriber->handle($event);
        }
    }
}
```

You can come up with lots of niceties and additions, but at its very core, this is all you need: a list of subscribers that can be notified whenever an event is dispatched.

ASYNCHRONOUS

Nothing is stopping you from making such an event bus asynchronous. In fact, an event-driven system is often preferred when doing async programming: it's a model that fits the parallel and async mindset extremely well.

For our examples, we'll assume the event bus always processes events synchronously: it's much simpler to reason about the event flow that way and eliminate many technicalities we'd have to deal with otherwise.

In our invoice example, we'd have two services that subscribe to the `InvoiceCreatedEvent` event:

```php
class InvoicePdfService implements Subscriber
{
    public function handles(object $event): bool
    {
        return $event instanceof InvoiceCreatedEvent;
    }

    public function handle(object $event): void
    {
        // Generate invoice PDF and save it on the filesystem
    }
}

class InvoiceMailService implements Subscriber
{
    public function handles(object $event): bool
    {
        return $event instanceof InvoiceCreatedEvent;
    }
    public function handle(object $event): void
    {
        // Send mail with a link to the customer's invoice page
    }
}
```

Our implementation could do with a little polishing, though. By adding some reflection capabilities to our event bus, we can determine whether a subscriber should handle an event based on its method signature instead. This not only makes our code more concise, but also allows us to know exactly what kind of event we're dealing with in our subscribers. Let's imagine we've refactored our event bus, and can now write subscribers like so:

```php
class InvoicePdfService
{
   public function handle(InvoiceCreatedEvent $event): void
   {
        // Generate invoice PDF and save it on the filesystem
   }
}
```

Let's continue exploring our event-driven system in depth. There's one big caveat with it - in fact, it's the major characteristic of all event-driven systems: indirectness.

Imagine the invoice creation the `InvoiceCreatedEvent` being triggered after the invoice was created:

```php
public function createInvoice()
{
    $invoice = /* … */;

    $event = new InvoiceCreatedEvent($invoice);

    $this->eventBus->dispatch($event);
}
```

While event-driven development promises flexibility — you can hook up as many subscribers as you want — it also causes a form of indirect coupling. We don't know what exactly will happen when we're triggering this event; we need to trust that the right subscribers will handle it without us seeing. This layer of indirectness can make debugging the program flow much harder, even when events are handled synchronously. On top of that: you can't have any direct return value after dispatching an event, since an infinite amount of subscribers can handle that event. However, you could introduce some polling layer to observe the results in, for example, a database. Still, there are many complications you have to deal with: flexibility at the cost of simplicity.

Did you notice that we haven't written any "event sourcing" or "CQRS" code, yet still, we're already working with an event-driven program?. You don't need event sourcing, a command bus, CQRS, or micro services in its simplest form. You only need events. Given enough time, though, any developer working in such an event-driven system would encounter problems with this simplified approach.

There could be a wide range of performance issues: degraded developer experience, scaling issues, or problems with managing consistency and state. You would naturally try to solve those problems by applying patterns and principles, and that's exactly what the event-driven community has been doing for years now. They have come up with patterns to help tackle the more difficult problems, the ones you'd encounter in real-life, complex systems.

Martin Fowler writes and talks about several such patterns, and how the community discovered them. In the next part of this chapter, I will briefly discuss four of those patterns, all of which are often used. The relevant links are listed at the end of this chapter.

EVENT NOTIFICATION

The simplest form of an event-driven system called "Event Notification": events are merely used to notify about something that has happened. Our example with the `InvoiceCreatedEvent` is already more complex than this since we're using the request data and sending it along with the event.

With event notifications, the event only signifies that something happened. It's up to the services themselves to reach out to the database, a third-party service, or outer state and determine what data they want to use. This is also the weakest form of event-driven development: everything is still coupled together. The only difference is that you're using the flexibility of events to hook several services up to one event.

EVENT-CARRIED STATE TRANSFER

The second pattern is what we have applied in our example: we've captured the relevant data when the event occurred and sent it together with the event. All services handling that event are only allowed to use the data that's encapsulated by that event.

This approach ensures that we can have multiple services listening to the same event and don't have to worry about which order they are handled in. We're always sure our services won't rely on the external state, so the event becomes "a source of truth".

EVENT SOURCING

Built upon the idea of events carrying all necessary state comes event sourcing. Instead of saving, for example, an invoice to the database, what would happen if we'd save the events themselves? Would that be beneficial?

If events become the source of truth and are saved to the database, there's a layer of extra information available to us at all times. Instead of knowing how the end result looks (the invoice), we now also know what the steps were that constituted that result (the events).

Take a look at this list of events, also called an "event stream":

```
[
    InvoiceDraftCreated::class,
    InvoiceSent::class,
    InvoicePaid::class,
]
```

If we used events only to trigger a service to do something, we'd lose the event's data after it was handled. Traditional applications often deal with these kinds of problems, which is why they keep track of state changes in the database: columns like `created_at` or `payment_date` are added on the invoice and have to be carefully managed from there on out.

If we're saving the events directly, though, we can retrieve them from a store (a database, filesystem, or something else), and rebuild our application state dynamically from scratch. For example, we can rebuild the invoice that would be the result of these events - at least as long as those stored events carry all the relevant data with them.

That's the power of event sourcing: being able to rebuild the whole application state, only by using events. It opens doors for interesting use cases. We could, for example, start generating reports based on the historical data that's available within these events. We could generate a report that analyses the average time it takes for customers to pay their invoice after it was sent to them, without rewriting our data model. That's right: all the data we need is already stored as events, we just need to interpret them in a new way.

With event sourcing come lots of other problems, though. The most pressing one: performance. A production application will store millions upon millions of events over time; surely, we can't rebuild our whole application state from scratch every time a request comes in. That's why there are other patterns helping us with those kinds of problems: projections and snapshots are often used to build a cached and reusable state, instead of always rebuilding it from scratch. A practical example could be an invoice projection: a table that stores the end result of all those invoice events and where we can easily read data from.

Another abstraction that often comes with event sourcing is the difference between the intention of making a change, and the change itself. When we directly triggered an `InvoiceCreatedEvent`, it felt a little off: the invoice itself wasn't created *yet*. Instead it would make more sense to have the intent called `CreateInvoice` and the actual result to store in the database `InvoiceCreated`. The first one is often called the "command", while the second one is called the "event".

A lot of complexity appears when we're trying to apply event sourcing properly. That's because of the same reasons our most basic implementation also added complexity: it's the cost we pay for a more flexible and scalable system. Keep this in mind: an event-driven architecture isn't always the right solution for a problem. It might very well be that a simpler approach is not only faster but also better.

A wise developer, Frank De Jonge, once said: *"Event Sourcing makes simple problems hard, and hard problems simple"*. Make sure you've weighed the pros and cons before adding event sourcing to your project.

CQRS

CQRS — command query responsibility segregation — is the fourth and final pattern I want to touch on. Martin Fowler describes it like this: *"At its heart is the notion that you can use a different model to update information than the model you use to read information. For some situations, this separation can be valuable [...] The rationale is that for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that does neither well."*

In other words: CQRS aims to separate the concerns of writing data and reading data. It again allows for more flexibility. Keep in mind that it's a pattern for very complex systems. Martin Fowler even warns not to use CQRS too quickly: *"Despite these benefits, you should be very cautious about using CQRS. Many information systems fit well with the notion of an information base that is updated in the same way that it's read, adding CQRS to such a system can add significant complexity."*

There's lots more to tell about event-driven systems, but it's outside the scope of this book. Several PHP frameworks facilitate an event-driven system, but most important is the event-driven mindset instead of a bunch of technical tools. I hope you've got a bit of an idea of what that mindset consists of, and I recommend checking out some more resources on the topic if you're interested in it.

SOME MORE RESOURCES

Martin Fowler's introduction to event-driven architecture:
https://www.youtube.com/watch?v=STKCRSUsyP0

Greg Young sharing insights on DDD, CQRS and Event Sourcing:
https://www.youtube.com/watch?v=LDW0QWie21s

IN CLOSING

I admit that finishing this book is somewhat of an emotional moment for me. I've been working on this project for over four years now; first individually on my blog, then together with my awesome colleagues at Spatie.

I feel that I said what I had to say. Even though there's much more to teach about PHP development, this book has laid a strong foundation. It feels like all those separate topics floating around my head finally got connected in one place.

My hope is that your learning process doesn't stop with this book. Even though proper learning material is hard to come by, I encourage you to keep challenging yourselves to grow as professional developers. That's what I've been doing for years now: keep learning and keep growing.

So in closing, thanks for reading, and I truly hope I was able to help you in your never-ending process of learning and growing.

Thanks,
Brent

PS: If you want a quick summary of all the new and fancy PHP syntax, you can head over to https://front-line-php.com/cheat-sheet and browse our handy cheat-sheet.