6

Object-Oriented Programming
with PHP

Certainly not every good program is object-oriented, and not every object-oriented program is good.

- Bjarne Stroustrup, Creator of C++

Contents

1 PHP 6
- 1.1 Hello, World! . . . 7
- 1.2 Composer . . . 8
  - 1.2.1 REPL . . . 8
  - 1.2.2 Taking a Dump . . . 8
- 1.3 Variables . . . 11
- 1.4 Documentation: A Warning . . . 11
- 1.5 Additional Resources . . . 12

2 Basic Types 13
- 2.1 Numbers . . . 13
- 2.2 Strings . . . 14
  - 2.2.1 Concatenation . . . 15
  - 2.2.2 String Functions . . . 15
- 2.3 Booleans . . . 16
  - 2.3.1 Boolean Logic . . . 16
  - 2.3.2 Comparison Operators . . . 17
  - 2.3.3 Falsy Values . . . 17
- 2.4 Additional Resources . . . 17

3 Control Flow 18
- 3.1 Conditionals . . . 18
  - 3.1.1 `if` Statements . . . 18
  - 3.1.2 Ternary Operator . . . 19
  - 3.1.3 `switch` Statements . . . 20
- 3.2 Loops . . . 20
  - 3.2.1 `for` Loops . . . 20
  - 3.2.2 `while` Loops . . . 21
  - 3.2.3 `do-while` Loops . . . 21
- 3.3 Additional Resources . . . 22

4 Functions 23
4.1 Scope . . . 24
4.2 Anonymous Functions . . . 24
4.3 Additional Resources . . . 26

5 Arrays 27
5.1 Numerically Indexed Arrays . . . 27
5.2 Associative Arrays . . . 28
5.3 Iterating Over Arrays . . . 29
5.4 Array Iterator Functions . . . 30
5.5 Additional Resources . . . 33

6 Regular Expressions 34
6.1 Parts . . . 35
6.1.1 Literal Strings . . . 35
6.1.2 Quantifiers . . . 35
6.1.3 Character Sets & Ranges . . . 36
6.1.4 Special Characters . . . 36
6.1.5 Character Classes . . . 37
6.1.6 Dot . . . 37
6.1.7 Anchors . . . 37
6.2 Regex with PHP . . . 38
6.2.1 `preg_match` . . . 38
6.2.2 `preg_split` . . . 39
6.2.3 `preg_replace` . . . 39
6.3 Alternatives to Regex . . . 40
6.4 The Dangers of Regex . . . 41
6.5 Additional Resources . . . 41

7 Classes 42
7.1 Properties . . . 44
7.1.1 Default Values . . . 45
7.2 `$this` . . . 45
7.3 Constructor . . . 46
7.4 Additional Resources . . . 48

8 Autoloading 49
8.1 Autoloading . . . 49
8.2 Namespaces . . . 50
8.3 PSR-4 Autoloading . . . 53
8.4 Additional Resources . . . 54

9 Object-Oriented Programming **55**
9.1 Bootstrapping . . . 56
9.2 Objectification . . . 58
9.3 Chaining . . . 61
9.4 Object-to-Object . . . 62
9.5 The Law of Demeter . . . 65
9.6 Additional Resources . . . 65

10 Encapsulation **66**
10.1 Visibility . . . 66
10.1.1 `public` . . . 66
10.1.2 `private` . . . 67
10.1.3 `protected` . . . 68
10.2 Encapsulation . . . 68
10.3 Writing Safe Code . . . 69
10.4 Type Safety . . . 71
10.4.1 `void` . . . 72
10.5 Type Systems . . . 73
10.6 Additional Resources . . . 74

11 Polymorphism **75**
11.1 Interfaces . . . 75
11.1.1 Message Passing . . . 78
11.2 Inheritance . . . 79
11.2.1 Abstract Classes . . . 81
11.2.2 Overriding . . . 82
11.2.3 `parent` . . . 83
11.2.4 Inheritance Tax . . . 84
11.3 Interfaces plus Inheritance . . . 85
11.4 Composition . . . 86
11.5 Additional Resources . . . 86

12 Static Methods & Properties **87**
12.1 The `static` Keyword . . . 88
12.2 Static Properties . . . 89
12.3 Additional Resources . . . 90

Glossary **91**

How To Use This Document

Bits of text in red are links and should be clicked at every opportunity. Bits of text in `monospaced green` represent code. Most the other text is just text: you should probably read it.

Some sections are marked "Read-Only": these are sections that are intended to be read through in your own time and will not have a corresponding lecture.

Copying and pasting code from a PDF can mess up indentation. For this reason large blocks of code will usually have a View Code link above them. If you want to copy and paste the code you should follow the link and copy the file from GitHub.

Taking Notes

In earlier cohorts I experimented with giving out notes in an editable format. But I found that people would often unintentionally change the notes, which meant that the notes were then wrong. I've switched to using PDFs as they allow for the nicest formatting and are also immune from accidental changes.

Make sure you open the PDF in a PDF viewing app. If you open it in an app that converts it into some other format (e.g. Google Docs) you may well miss out on important formatting, which will make the notes harder to follow.

I make an effort to include all the necessary information in the notes, so you shouldn't need to take any additional notes. However, I know that this doesn't work for everyone. There are various tools that you can use to annotate PDFs:

- Preview (Mac)
- Edge (Windows)
- Hypothes.is
- Google Drive (*not* Google Docs)
- Dropbox

Do not use a word processor to take programming notes! (e.g. Google Docs, Word, Pages). Word processors have the nasty habit of converting double-quotes into "smart-quotes". These can be almost impossible to spot in a text-editor, but will completely break your code.

Chapter 1

PHP

> *I don't know how to stop it, there was never any intent to write a programming language ... I have absolutely no idea how to write a programming language, I just kept adding the next logical step on the way.*
>
> - Rasmus Lerdorf, Creator of PHP

PHP is a general purpose programming language that is frequently used to run the **server-side** code for websites. It was originally created as a simple **templating language**, based loosely on Perl, to allow outputting HTML with repeated elements – PHP originally stood for "Personal Home Page"[1]. But, over the years, it has evolved into a fully **object-oriented programming language**.

PHP should look quite familiar if you've done JavaScript as they are both "C-based" languages, meaning that they share a syntax style: semi-colons, curly braces, and brackets.

We can run PHP in the command-line similarly to how we did in Week 3 with JavaScript:

```
php file.php # run the given file
```

[1] More recently "PHP: Hypertext Preprocessor"

1.1 Hello, World!

As is tradition, we should write a "Hello, World!" program before moving forward:

```
<?php

echo "Hello, World!";
```

We "echo" the string "Hello, World!", this is PHP's equivalent of using `console.log()` – although it's only useful when used with strings. `echo` is *not* a function (or method), but instead a **keyword**. All that really means is that you don't *call* it.

As you can see, it's not that different from JavaScript. The main difference is that on line 1 we have to tell PHP that it's PHP. If we don't do this then PHP has a bit of an identity crisis and doesn't know what's going on.

Make sure you never have anything before the opening `<?php` tag – no spaces, line breaks, or comments – as this will break most modern PHP apps.

Ye Olde PHP

The opening `<?php` tag can seem a bit strange in modern PHP, as it doesn't seem to serve any purpose – surely it knows it's PHP?

When PHP was used mainly as a templating language PHP files would be made up mostly of HTML with only snippets of PHP:

```
<body>
    <h1><?php echo $header ?></h1>
    <div>
        <?php echo $content ?>
    </div>
</body>
```

PHP is still used like this in some PHP frameworks such as WordPress. Nowadays templating languages like Blade and Twig are more commonly used, thus separating the program logic from the templating language.

In JavaScript you could get away without semi-colons at the end of lines. PHP isn't nearly so forgiving: if you forget a semi-colon you will get a syntax error and the code will refuse to run.

From now on code examples won't include the opening tag, but you will need to add it as the first line in all your files.

1.2 Composer

Composer is the PHP package manager. It lets us easily download code that other people have written and use it in our own code. Although it's technically not necessary to use it at this point, we're going to be using Composer to make our PHP experience more pleasant.

1.2.1 REPL

First, let's get a PHP REPL, as there isn't one built in:

```
composer global require psy/psysh
```

This will install PsySH. You only need to run this once on your machine. The `global` bit means that it will install it so that you can use it anywhere on your computer.

You should now be able to run `psysh` from any directory to get a PHP REPL up and running. This can be useful for quickly checking bits of code.

Just type `exit` if you want to get back to the command-line.

1.2.2 Taking a Dump[2]

As mentioned above `echo` is actually pretty useless for anything other than strings[3]. If we want to be able to log out any data type, then we'll need to use something else.

PHP does have the `var_dump()` function, which will log out different data types, but it's still fairly horrible.

[2] Teehee
[3] Try `echo`ing `true` if you don't believe me

```
var_dump(12); // int(12)
var_dump([1, 2]); // array(3) { [0]=> int(1) [1]=> int(2) }
```

We're going to use the `symfony/var-dumper` package[4] to allow us to get behaviour much more similar to `console.log()` in JS. We'll need to install this on a **per-project** basis. To do this, first choose the directory that you want to work from, and then run:

```
composer require symfony/var-dumper
```

This will create `composer.json` and `composer.lock` files and a `vendor` directory.

Composer & Git

The `composer.json` tracks which packages you've installed (as well as other bits of Composer configuration). The `composer.lock` file keeps track of the *exact* versions of packages that have been installed so that another developer could recreate an identical set of packages.

You can recreate your `vendor` directory by running `composer install` after cloning a repository.

As such, your `composer.json` and `composer.lock` files *should* go into version control. Your `vendor` directory should *not* go into version control. So you should immediately create a `.gitignore` file when adding Composer:

.gitignore View Code

```
vendor/
```

[4] Part of the Symfony framework, but written so that it can be used with any PHP code

To use the package (and any other packages that we install) we need to let PHP know to use the Composer files. We do this by adding the following line to the top of any PHP files that need to use it:

```
// load in the Composer configuration
// __DIR__ just means the directory this file is in
require __DIR__ . "/vendor/autoload.php";
```

We now have access to the glamorously named `dump()` function[5]. `dump()` is our `console.log()` equivalent: it outputs things nicely and automatically adds syntax highlighting to the output. Having installed the package, we could rewrite our "Hello, World!" example as follows:

```
<?php

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

dump("Hello, World!");
```

We'll be using `dump()` from now on.

[5]And the even more glamorously named "dump and die" function `dd()`

1.3 Variables

In PHP we don't *declare* variables, we just start using them. This is possible because variables have to start with a `$` (this is because of PHP's Perl influence).

This makes it much easier to accidentally change the values of existing variables, as there is no difference between creating a new variable and reassigning an existing one. So be very careful when naming things.

```
$name = "Archie";

$age = 4;
$houseNumber = 21;

$name = "Ben"; // changes the value of $name - deliberate?

// using variables
$notUseful = $age + $houseNumber; // 25
```

Variables must start with a dollar, followed by a letter or underscore, followed by any number of letters, numbers, or underscores. Variable names are case-sensitive.

As with JavaScript it is a standard convention to use `$camelCase` for variable names in PHP, although you may see `$snake_case` used in older code.

1.4 Documentation: A Warning

Be careful using the official PHP documentation: *anyone* can submit "User Contributed Notes" and they often contain code samples that are to be avoided. This is often because the notes were added years ago when PHP was a very different language.

As a general rule, don't look at the "User Contributed Notes" section: use Stack Overflow if the documentation hasn't cleared it up for you.

1.5 Additional Resources

- Composer
- PsySH
- Symfony: The VarDumper Component
- PHP: Variable Basics
- Wikipedia: PHP
- Wikipedia: Perl

Chapter 2

Basic Types

PHP has the same basic types as JavaScript: numbers, strings, and booleans (also known as the "Scalar" types[1]). They work in much the same way.

2.1 Numbers

Unlike JavaScript PHP does have a sense of whether a number is an integer (whole number) or a "floating point number" (one with a decimal place). This means that some of the issues we had with JavaScript and decimal numbers don't cause issues in PHP:

```
dump(12 + 12); // 24
dump(0.1 + 0.2); // 0.3 - hurrah!
```

The operators should be familiar:

Operator	Name	Description
`+`	addition	adds two numbers together
`-`	subtraction	subtracts the second number from the first number
`*`	multiplication	multiplies two numbers
`/`	division	divides the first number by the second
`%`	modulus	remainder after dividing the first number by the second

[1] Meaning they represent a single value

We don't have a `Math` object in PHP, instead there are just lots of functions - but the naming should be familiar:

```
// rounding
floor(12.3030); // 12
ceil(12.3030); // 13
round(12.3030); // 12

// powers/roots
pow(2, 3); // 8
sqrt(16); // 4

// random
mt_rand(5, 10); // random integer between 5 and 10 (inclusive)

// trigonometry
cos(1.2); // 0.362... takes an angle *in radians*
asin(0.3); // 0.304...returns value *in radians*
tanh(3); // 0.995... hyperbolic tangent
```

But we can still run into weird issues with numbers in PHP. As a general rule *never trust floating point numbers*

```
floor((0.1 + 0.7) * 10); // 7 - oop!
```

2.2 Strings

Strings are also very similar to JavaScript.

You can use single- or double-quotes:

```
$firstName = "Casper";
$lastName = 'Spoooky';
```

Double-quotes allow you to interpolate values:

```
$fullName = "{$firstName} {$lastName}";
```

We use curly-braces to enter interpolation mode and then use the variable name inside. The `$` here is part of the variable, whereas in JavaScript interpolation it's part of the syntax for interpolation (so the dollar is *outside* the curly-braces).

Because interpolation is such a common thing to do, generally we'll use double-quotes for strings.

2.2.1 Concatenation

PHP uses `.` for concatenation, this avoids the issues that JavaScript had with the overloaded `+` operator. It can, however, lead to many a brain-fart as you try and use `+` when you mean `.`:

```
dump($firstName . $lastName); // "CasperSpooky"
dump($firstName + $lastName); // PHP Warning - A non-numeric value
↪  encountered
dump("1" + "2"); // 3 - coerces to numbers
```

2.2.2 String Functions

Unlike in JS, strings are not objects in PHP. That means they don't have properties or methods. So we have to use functions to work with them:

```
strtolower("Blah"); // "blah"
strtoupper("Blah"); // "BLAH"
trim("   Blah  "); // "Blah"
substr("Fishsticks", 4); // "sticks"
```

There are many other string function in the PHP documentation.

2.3 Booleans

As with JavaScript, PHP has the boolean values `true` and `false`. The only difference in PHP is that they're not case sensitive:

```
$bool = true;
$bool = True; // also valid
$bool = TRUE; // still valid
$bool = TrUe; // totes valid
```

For consistency it's best to stick with the lowercase version.

2.3.1 Boolean Logic

PHP has `&&`, `||`, and `!` which work in the same way as in JavaScript:

```
true && false; // false
true || false; // true
!true; // false
!false; // true
```

PHP also has the written versions `and` and `or`:

```
true and false; // false
true or false; // true
```

If you're coming from JavaScript, you should use the `&&` and `||` versions. The other versions have different "precedence" and will not always work as you expect.

2.3.2 Comparison Operators

All your favourite comparison operators are back:

Operator	Name	Description
`===`	strict equality	`true` if the values are the same
`!==`	non-equality	`false` if the values are the same
`<`	less than	`true` if the first value is less than the second value
`>`	greater than	`true` if the first value is greater than the second value
`<=`	less than or equal to	`true` if the first value is less than or equal to the second value
`>=`	greater than or equal to	`true` if the first value is greater than or equal to the second value

As with JavaScript there are also the type-coercing `==` and `!=` operators, but these are best avoided.

2.3.3 Falsy Values

PHP has all the falsy values that JavaScript has. Empty arrays are also falsy in PHP (which they are *not* in JavaScript):

- `false` itself
- The number zero: `0`, `0.0`, `-0`, `-0.0`
- The empty string: `""`
- An empty array: `[]`
- `NULL`

2.4 Additional Resources

- PHP: Arithmetic Operators
- PHP: Maths Functions
- PHP: Logical Operators
- PHP: Comparison Operators

Chapter 3

Control Flow

Because they're both C-based languages, loops and conditionals in PHP and JavaScript are syntactically identical. Just remember that variables in PHP are not declared and always start with a `$`.

3.1 Conditionals

3.1.1 `if` Statements

A basic `if` statement:

```
if ($x < 10) {
    // do a thing
}
```

With an `else`:

```
if ($x < 10) {
    // do a thing
} else {
    // do the other thing
}
```

An `else if`:

```
if ($x < 10) {
    // do a thing
} else if ($x < 20) {
    // do this thing
} else {
    // do the other thing
}
```

In PHP the space between `else` and `if` can be omitted:

```
if ($x < 10) {
    // do a thing
} elseif ($x < 20) {
    // do this thing
} else {
    // do the other thing
}
```

Although they're not technically necessary for single line blocks, you should *always* use the curly braces around a block.

3.1.2 Ternary Operator

PHP also has the ternary operator. As with JavaScript, a ternary operator is an *expression*:

```
// if $index is less than 0, set it to 5
// otherwise decrement it
$index = $index < 0 ? 5 : $index - 1;
```

3.1.3 `switch` Statements

`switch` statements are also available:

```
switch ($x) {
    case 1:
        $message = "It's One";
        break;
    case 2:
        $message = "It's Two";
        break;
    default:
        $message = "No idea";
}
```

Don't forget to `break` at the end of each `case`; and, remember, `switch` statements are only useful if you want to run different bits of code based on the *same expression*.

3.2 Loops

3.2.1 `for` Loops

`for` loops are much the same as in JavaScript (except we don't declare the counter variable and there are dollars everywhere):

```
$total = 0;

for ($i = 1; $i <= 10; $i += 1) {
    $total += $i;
}

dump($total); // 55
```

Remember the three parts:

1. Setup the counter variable (runs once before the loop starts)
2. Loop condition: run the loop as long as this is `true`
3. Evaluated after each iteration

It's generally best to use a `for` loop if you know how many times the loop needs to run.

3.2.2 `while` Loops

While loops are useful if you don't know how many times the loop needs to run:

```
$i = 0;
$total = 0;

while ($total < 100) {
    $i += 1;
    $total += $i;
}

dump($total); // 105
```

3.2.3 `do-while` Loops

A `do-while` loop is the same as a `while` loop, except that the `do` block will always run *at least once*. They can sometimes be a little easier to work out:

```
$i = 0;
$total = 0;

do {
    $i += 1;
    $total += $i;
} while ($total < 100);

dump($total); // 105
```

∞

As with all loops, be careful not to create an infinite loop!

3.3 Additional Resources

- PHP: `if`
- PHP: `else if`
- PHP: The Ternary Operator
- PHP: `switch`
- PHP: `for` Loops
- PHP: `while` loops
- PHP: `do-while` loops

Chapter 4

Functions

Functions in PHP serve the same purpose as functions in JavaScript: they allow us to reuse bits of code.

They are written in the same way as functions were historically written in most C-style language (including JavaScript):

```
function add($a, $b) {
    return $a + $b;
}

$result = add(12, 34);
dump($result); // 46
```

4.1 Scope

Because we don't declare variables in PHP it takes a very explicit approach to scope: by default variables used inside functions are assumed to be *locally* scoped:[1]

```
$x = 10;

function wtf($z) {
    $x = 20; // different $x, locally scoped
    return $x + $z;
}

$result = wtf(12);
dump($result); // 32
dump($x); // 10
```

4.2 Anonymous Functions

Standard functions in PHP are not values like they are in JavaScript, so we can't just pass them round and assign them to variables in quite the same way. However, passing functions around is such a useful thing to be able to do that recent versions of PHP added **closures** as a way to do this:

```
$add = function ($a, $b) {
    return $a + $b;
};

$result = $add(1, 2);
dump($result); // 3
```

As you can see, because it's stored in a variable we need to use the `$` when calling the function.

[1]The `global` keyword allows us to get around this, but it's unlikely you'll ever need to

Some functions in PHP require a `callable` argument, which just means a function:

```
$result = array_map(function ($value) {
    return $value * $value;
}, [1, 2, 3, 4, 5]);

dump($result); // [1, 4, 9, 16, 25]
```

Closures don't have access to variables declared outside of themselves. In order to use these you use the `use` keyword:

```
$multiplyBy = 10;

$result = array_map(function ($value) use ($multiplyBy) {
    return $value * $multiplyBy;
}, [1, 2, 3]);

dump($result); // [10, 20, 30]
```

In PHP 7.4+ this could be written as:

```
$multiplyBy = 10;

$result = array_map(fn($value) => $value * $multiplyBy, [1, 2, 3]);

dump($result); // [10, 20, 30]
```

This is PHP's new Arrow Function syntax. It avoids the need for `use` as it uses the parent scope. It also automatically returns a value, like with JS's fat arrow functions. However, it can **only be used if the function body is a single expression**: there's no way to have multiple lines.

It's worth noting that in both cases you only get read access to `$multiplyBy`: you cannot change its value inside the closure.

4.3 Additional Resources

- PHP: Functions
- PHP: Callable
- PHP: Arrow Functions

Chapter 5

Arrays

Arrays come in two forms in PHP: numerically indexed and associative.

5.1 Numerically Indexed Arrays

Numerically indexed arrays are exactly the same as in JavaScript.

```
$values = [1, 2, 3, 4, 5];
$first = $values[0]; // first item in array
$last = $values[4]; // last item in this array
dump($first); // 1
dump($last); // 5
```

We can add a value to the end of an array:

```
$values[] = 6;
```

We can also find out how many items are in an array:

```
count($values); // 5
```

Arrays in PHP aren't objects like in JavaScript, so they don't have methods. We have to use built-in functions like `count`.

Old Skool Arrays

If you're working with older PHP you may see arrays written using the older notation:

```
$values = array(1, 2, 3, 4, 5);
```

This isn't actually a function, although it does look like one.

The newer notation was added in PHP 5.4, which has reached end-of-life, so you generally don't need to use the older notation. However, certain coding standards (such as WordPress) discourage the use of the newer style.

5.2 Associative Arrays

Associative arrays are effectively PHP's version of object literals: in PHP "objects" are (almost) always instances of a `class`, but the key-value pairing of object literals is still a useful concept:

```
$assoc = [
    "firstName" => "Ben",
    "lastName" => "Wales",
    "dob" => "2018-08-24",
];

dump($assoc["lastName"]); // "Wales"
```

Notice that the syntax is quite different from JS: square brackets to open, keys need quoting, and fat-arrow between the key and value.

All Arrays Are Associative

Technically speaking, all arrays in PHP are actually associative, they're just automatically given a numerical key if you don't provide one. This does mean you can run into some unusual results if you're not careful:

```
$arr = [];
$arr["key"] = 1; // key provided
$arr[] = 2; // no key, so starts at 0

dump($arr); // ["key" => 1, 0 => 2]
```

You should not deliberately mix numerically indexed and associative arrays as things can get weird.

5.3 Iterating Over Arrays

We can use a `foreach` loop to iterate over every item in an array.

```
$values = [1, 2, 3, 4, 5];

foreach ($values as $value) {
    // $value will be each value in turn
}
```

You can also get the key:

```
$assoc = [
    "firstName" => "Ben",
    "lastName" => "Wales",
    "dob" => "2018-08-24",
];

foreach ($assoc as $key => $value) {
    // $key will be each key in turn
    // $value will be each value in turn
}
```

As numerically indexed arrays are just associative arrays with numerical keys, you can also use this syntax to get the index of a numerical array:

```
$values = [1, 2, 3, 4, 5];

foreach ($values as $index => $value) {
    // $index will be each index in turn
    // $value will be each value in turn
}
```

We called it `$key` in the first case and `$index` in the second, but we can call it whatever we like, as long as it appears before the fat arrow.

5.4 Array Iterator Functions

PHP does have functions for doing `map`, `filter`, and `reduce` but they're inconsistent and not always usable (for example, if you need the current key/index).

However, we can install Laravel's support package to gain access to "Collections", which makes working with arrays much nicer. It's got tonnes of really useful methods, but we'll just look at four of them here: our old friends `map()`, `filter()`, and `reduce()`, as well as a very useful one called `pluck()`.

First we need to install the package:

```
composer require illuminate/support
```

Generally collection methods return a collection. You can turn a collection back into a standard array by calling its `all()` method.

`filter`

Filter is almost identical to JavaScript: we pass it an anonymous function that takes each item in the array and returns a boolean value. It returns a new collection containing all the items for which the function returned `true`:

View Code

```
// use the collect function to create a new collection
$numbers = collect([1, 2, 3, 4, 5]);

$even = $numbers->filter(fn($n) => $n % 2 === 0);

dump($even->all()); // [2, 4]
```

`map`

Map is also very similar to JavaScript: we pass it an anonymous function that takes each item in the array and transforms the value somehow. It returns a new collection where each item has been transformed:

View Code

```
$numbers = collect([1, 2, 3, 4, 5]);

$squared = $numbers->map(fn($n) => $n * $n);

dump($squared->all()); // [1, 4, 9, 16, 25]
```

reduce

Again, reduce is very similar to JavaScript: we pass it an anonymous function that takes the accumulated value and each item in the array. The return value is passed in as the accumulator value for the next iteration. It returns the final accumulated value:

View Code

```
$numbers = collect([1, 2, 3, 4, 5]);

// remember, reduce takes two arguments
// the accumulator and each value in turn
// the initial value for $acc is null
// so make sure you set it
$sum = $numbers->reduce(fn($acc, $val) =>$acc + $val, 0);

dump($sum); // 15
```

Make sure you pass in an initial value for the accumulator, otherwise it will be `null`, which might cause problems.

pluck

We've not come across `pluck` before, but it's very useful. It assumes your collection contains either associative arrays or objects all with the same structure. You pass it a key value and it extracts a new collection containing just that key/property from each item in the collection:

View Code

```
$goodWatchin = collect([
  ["id" => 1, "name" => "Unbreakable Kimmy Schmidt"],
  ["id" => 2, "name" => "The Leftovers"],
  ["id" => 3, "name" => "Game of Thrones"],
]);

// [1, 2, 3]
dump($goodWatchin->pluck("id")->all());

// ["Unbreakable Kimmy Schmidt", "The Leftovers", "Game of Thrones"]
dump($goodWatchin->pluck("name")->all());
```

Under the Hood

The `collect` function is actually creating an instance of the `Collection` class, which wraps the array, and you're calling various methods on it. When you call the `all()` method it gives you back the array it's been working on.

5.5 Additional Resources

- PHP: Arrays
- PHP: Array Functions
- PHP: `foreach`
- Collections

Chapter 6

Regular Expressions

Regular Expressions (or "regex" for short) are a way to check/search a string for a *pattern* as opposed to a specific string.

They're a little bit mad looking to start with. In fact they're a little bit mad looking even after you've been using them for years. But they are very useful as long as you don't get too carried away.

For example, we might want to split a string on a comma followed by *any* number of spaces (not just one). We'd write that using the following regex:

```
/,\s*/
```

Or if we wanted to match any combination of lowercase letters, numbers, underscores, and hyphens between 3 and 16 characters long we could write this with the following regex:

```
/^[a-z0-9_-]{3,16}$/
```

JavaScript also supports regexes (it's actually its own *type* in JS, like numbers and strings).

6.1 Parts

We're not going to get into every aspect of Regexes, but we'll cover enough for them to be useful.

6.1.1 Literal Strings

The first thing to be aware of is that a RegEx of just non-special characters represents that string. So `wombat` is just the string "wombat".

6.1.2 Quantifiers

Quantifiers allow us to specify that a character should appear zero, one, or many times.

Quantifier	Description
`*`	0 or more
`+`	1 or more
`?`	0 or 1
`{3}`	exactly 3
`{3,}`	3 or more
`{3,5}`	3, 4, or 5

For example:

```
/a+/        - would match 'a', 'aa', 'aaa', 'aaaa', etc.
/b*/        - would match '', 'b', 'bb', 'bbb', etc.
/c?/        - would match '' and 'c'
/d{3}/      - would match 'ddd'
/d{3,5}/    - would match 'ddd', 'dddd', and 'ddddd'
/abc*/      - would match 'ab', 'abc', 'abcc', 'abccc', etc.
/(abc)*/    - would match '', 'abc', 'abcabc', 'abcabcabc', etc.
/https?:/   - would match 'http:' and 'https:'
```

Notice that we can use brackets to quantify more than one character.

6.1.3 Character Sets & Ranges

Character sets and ranges allow us to specify a range of characters that we're interested in.

Character Set	Description
`[abc]`	'a', 'b', or 'c'
`[^abc]`	**not** 'a', 'b', or 'c'
`[a-z]`	all lowercase letters
`[A-Z]`	all uppercase letters
`[0-9]`	all numbers
`[0-9afg]`	all numbers plus 'a', 'f', and 'g'
`[a-zA-Z]`	all letters, case-insensitive
`[a-zA-Z0-9]`	alphanumeric characters
`[0-9A-F]`	valid hexadecimal digits

A character set is a set of characters wrapped with square brackets that will match *any* character in the set.

A range, which must always be used *inside* a character set, represents a specified series of characters, e.g. all the letters between `a` and `z`.

For example:

```
/[a-z]+/      - any number of lowercase letters
/[a-z0-9_-]/  - a single lowercase letter, digit, underscore, or hyphen
```

6.1.4 Special Characters

These represent special characters like tab and a new line:

Character	Description
`\n`	a new line
`\r`	a carriage return
`\t`	a tab

`\r` tends to only crop up if you read directly from a file generated on a Windows machine.

These can generally be used in regular strings too.

6.1.5 Character Classes

Character classes are shortcuts for specific ranges.

Class	Description
`\s`	white**s**pace (spaces, tabs, etc.)
`\S`	*not* white**s**pace
`\w`	**w**ord (`[A-Za-z0-9_]`)
`\W`	*not* **w**ord
`\d`	**d**igit
`\D`	*not* **d**igit

For example:

```
/,\s*/      - a comma followed by 0 or more whitespace characters
/\w\s+\w/   - two words separated by 1 or more whitespace characters
```

6.1.6 Dot

In a regex the `.` character has a special meaning: *any* character except for `\n`. You need to be careful using it, particularly with the `*` and `+` quantifiers.

```
/.+@.+/     - '@' symbol with something either side
```

If you want to match an actual full stop you need to "escape" it with a backslash:

```
/\.+@\.+/   - an '@' symbol with some number of '.' either side
```

6.1.7 Anchors

Sometimes *where* the substring appears is important.

Anchor	Description
`^`	beginning of the string
`$`	end of the string

For example:

```
/^abc/          - would match 'abc' but not '0abc'
/abc$/          - would match 'abc' but not 'abc0'
```

6.2 Regex with PHP

We can use regexes for all sorts of string manipulations. The three most common are:

- Searching a string
- Splitting a string
- Replacing a string

6.2.1 `preg_match`

The `preg_match` function can be used to check if a string matches a regular expression:

```
// does the string contain one or more 'l' characters
preg_match("/l+/", "Hello"); // 1

// does the string *start* with one or more 'l' characters
preg_match("/^l+/", "Hello"); // 0

// does the string contain two words, separated by a space
preg_match("/\w\s+\w/", "Hello There World"); // 1

// does the string consist of *just* two words, separated by a space
preg_match("/^\w\s+\w$/", "Hello There World"); // 0
preg_match("/^\w\s+\w$/", "Hello Mum"); // 1
```

It returns `1` if a match is found and `0` if it is not. Make sure you always use `===` when checking the result, as it returns `false` if an error occurs – which might get confused for `0` if you use `==`:

```
if (preg_match("/l+/", "Hello") === 1) {
    // matches one or more 'l' characters
}
```

6.2.2 preg_split

`preg_split` can be used to split a string on a certain regex:

```
$csv = "first, second,   third,fourth";

// split on a comma followed by 0 or more spaces
$result = preg_split("/,\s*/", $csv);

// [
//    [0] => "first",
//    [1] => "second",
//    [2] => "third",
//    [3] => "fourth"
// ]
```

We pass it a regex and a string and it gives us back an array of strings where the original string has been split on the regex.

6.2.3 preg_replace

The `preg_replace` function can be used to replace part of a string that matches a regex with something else:

```
$str = 'blah      blah   blah';

// replace one or more space with a single space
$tidied = preg_replace("/\s+/", " ", $str);

// "blah blah blah"
```

There is a lot more to `preg_replace` than this basic example,[1] but we'll keep it simple for now.

[1] "Capture groups" and "back references" are particularly useful

Flags

In PHP we can add various **flags** to the end of the regex. These go after the last forward-slash, e.g. `"/[a-z]*/i"`.

There are three particularly useful ones in PHP:

Flag	Name	Description
`i`	case insensitive	pattern will match upper and lower case
`m`	multi-line	separate lines count as separate strings for anchors
`s`	dot all	the `.` character should include new lines

6.3 Alternatives to Regex

For basic validation you are often better using PHP's `filter_var` function:

```
$email = "penny@hello.horse";
$valid = filter_var($email, FILTER_VALIDATE_EMAIL);

if ($valid) {
    // valid email address
}
```

The `filter_var` function takes a string and a filter type. It then returns the filtered string if it is valid or `false` otherwise.

Here are some particularly useful filters:[2]

- `FILTER_VALIDATE_EMAIL`
- `FILTER_VALIDATE_DOMAIN`
- `FILTER_VALIDATE_URL`

There's a full list on the PHP docs.

[2]These big shouty looking things are constant variables defined by PHP – in C-based languages constants are often written in uppercase with underscores

6.4 The Dangers of Regex

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*
>
> - Jamie Zawinski

It's not uncommon for people new to programming to try and solve complex string manipulations using regular expressions. This can lead to hard to read and inefficient code. There are many problems that require a **parser**: a much more clever sort of algorithm that can elegantly cope with things like matching start/end tags.

As a general rule, if your regular expression isn't easy to understand in one glance, then you probably shouldn't be using it.

Fine:

```
/,\s*/
```

Fuck off:

```
/^(?:(?:https?|ftp)://)(?:\S+(?::\S*)?@|\d{1,3}(?:\.\d{1,3}){3}|(?:(?:[a-z\d\x{00a1}-\x{ffff}]+-?)*[a-z\d\x{00a1}-\x{ffff}]+)(?:\.(?:[a-z\d\x{00a1}-
```

6.5 Additional Resources

- Regexr: an online Regex testing tool - make sure you set it to use "PCRE"
- RegExOne: Learn Regular Expressions with simple, interactive exercises
- PHP: `preg_match`
- PHP: `preg_matchall`
- PHP: `preg_replace`
- PHP: `preg_split`
- freeCodeCamp: RegEx Exercises with JavaScript
- Regular Expressions: Now You Have Two Problems
- RegEx Crossword
- Stack Overflow: RegEx match open tags except XHTML self-contained tags

Chapter 7

Classes

In PHP (and all object-oriented languages), an **object** is a way of storing some values (**properties**) and functionality (**methods**) that are related in some way.

It's very common to need many objects that have the same methods and property names, but where the *values* of the properties are specific to each object.

For example we might have two objects, that both represent people: both have a `firstName` property, a `lastName` property, and a `birthdate` property as well as a `getAge()` method, which returns their age based on the date of birth and the current date. The property names and methods will be identical for both objects, however the *values* for the properties will be different: a "Mark" object would have `"Mark"`, `"Wales"`, and `"1984-04-16"` for the respective values, whereas a "Ben" object would have `"Ben"`, `"Wales"`, and `"2018-08-24"`.

`firstName`	"Mark"	"Ben"
`lastName`	"Wales"	"Wales"
`birthdate`	"1984-04-16"	"2018-08-24"
`getAge()`	36	1

Two different objects with the same properties, but different values

A **class** is an abstract representation of an object that you want to create. For example, you might have a class `Person` that allows you to create lots of object **instances** representing different people.

Here's a class that represents a person:

View Code

```
class Person
{
    private $firstName;
    private $lastName;
    private $birthdate;

    public function __construct($firstName, $lastName, $birthdate)
    {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
        $this->birthdate = $birthdate;
    }

    public function getAge()
    {
        $date = new DateTime($this->birthdate);
        $now = new DateTime();
        return $now->diff($date)->y;
    }
}
```

Here's how we'd use our class:

View Code

```
// create new Person object "instances"
// passing in the unique values when we create it
$mark = new Person("Mark", "Wales", "1984-04-16");
$ben = new Person("Ben", "Wales", "2018-08-24");

// we can call the getAge() method
$mark->getAge(); // 36
$ben->getAge(); // 1
```

You can see that where we'd write a dot in JavaScript (`ben.getAge()`), we write an arrow in PHP (`$ben->getAge()`), but otherwise it's almost identical in usage.

PSR-2: Coding Style Guide

You've possibly noticed that in all the examples above the opening curly brace ({) for classes and methods is on its own line. This is part of the PSR-2: Coding Style Guide spec.

If you do an `if` statement (or other control structure) then the opening curly brace, obviously, goes on *the same* line.

You're probably thinking that this doesn't make the slightest bit of sense. And you'd be right. PSR-2 was created by sending round a questionnaire about coding style to 30 or so of the most prolific PHP programmers and they just went with whatever the majority said for each point.

But it's the style that everyone uses now. You'll get used to it.

7.1 Properties

In most OOP languages we can declare the properties that our object has[1]. We normally declare all of our properties at the top of the class: this makes it easy to see in one glance what properties the object instance will have.

View Code

```
class Person
{
    private $firstName;
    private $lastName;
    private $birthdate;

    // ...rest of code
}
```

We declare properties by using the `private`[2] keyword followed by the name of the property – with a dollar at the front, just like with variables. However, properties are `not` variables, as they belong to a specific object instance: each object instance with its own set of values.

[1] JavaScript is still relatively immature as a "classical" OOP language, but support for this is coming
[2] More on this later

You should always get and set properties using methods ("getters" and "setters"). We'll go into why this is later in the week.

7.1.1 Default Values

We can assign properties default values when we declare them. For example, we could add an `$arms` property to the `Person` class and set it to `2` by default:

View Code

```
class Person
{
    private $firstName;
    private $lastName;
    private $birthdate;

    // add an $arms property with the value of 2
    private $arms = 2;

    // ...rest of code
}
```

You can always use a method to change the value later.

If you declare a property but don't assign it a value it will have the value `null`.

7.2 `$this`

Inside our classes we can use the `$this` keyword to access properties and methods that belong to the current object instance. It works in much the same way as JavaScript except that it's much more reliable: `$this` in PHP *always* refers to the current object and has no meaning elsewhere.

View Code

```
class Address
{
  private $street;
  private $town;
  private $postcode;
  private $country;
```

```
  // a setter function
  public function setStreet($street)
  {
    // $this represents whichever object we're working on
    $this->street = $street;
    return $this;
  }

  // ...setTown, setCountry, setPostcode as above

  // a getter function
  public function getFullAddress()
  {
    $parts = array_filter([
      $this->street,
      $this->town,
      $this->postcode,
      $this->country,
    ]);

    // you can call methods too
    return implode(", ", $parts);
  }
}
```

Notice that it's the `$this` bit that has the dollar in front of it, the property name does not, even though they *do* have a dollar in front of them when we declare them at the top of the class[3].

7.3 Constructor

When we create a new object instance with `new` any values that we pass as arguments will be given to the `__construct()` method, just as if we'd called it ourselves. So the number of arguments that we pass with `new` should always match the number of parameters that the `__construct()` method accepts:

View Code

```
class Person
{
    // ...properties
```

[3] For weird historical reasons

```
    // takes three parameters
    public function __construct($firstName, $lastName, $birthdate)
    {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
        $this->birthdate = $birthdate;
    }

    // ...other methods
}

// we pass in three arguments
$mark = new Person("Mark", "Wales", "1984-04-16");
```

We need to make sure we store the values passed in as properties on the newly created instance, otherwise they'll cease to exist once the `__construct()` method has completed[4]. It's fairly standard to name the parameters the same thing as the properties that they'll be assigned to as in the above example.

If we've given a property a default value we don't need to set it up in `__construct()` as well, it will automatically be given the default value when the object instance is created. If *all* of your properties have default values then a `__construct()` method might not even be necessary[5].

[4] Remember, parameters only exist inside the function they belong to

[5] It's not uncommon in full OOP code to accept no parameters in the constructor and to use setters for everything.

Using `new` Objects

Unlike in JavaScript you can't immediately use a created object:

```
// won't work
new Person("Jim", "Henson", "1936-09-24")->getAge();
```

If you really want to, you can get around this with a pair of brackets around the `new` statement:

```
// will work
// but you don't have a reference to the object anymore
(new Person("Jim", "Henson", "1936-09-24"))->getAge();
```

However, this isn't something you're likely to need to do all that often.

7.4 Additional Resources

- PHP Apprentice: Classes
- Laracasts: Classes 101
- Laracasts: Classes

Chapter 8

Autoloading

We only know how to use PHP when everything is in one `.php` file. As we start to build more complex code with multiple classes it would be good to keep things separate. Ideally we'd have one `class` per file in an easy to find location.

8.1 Autoloading

Currently, if you put one of your classes in a separate PHP file and then try and use it from another PHP file you'll get an error:

```
<?php

$app = new App(); // PHP Fatal error: Class 'App' not found
```

We could require the file this is in manually:

```
<?php

require_once __DIR__ . "/classes/App.php";

$app = new App(); // success!
```

But this will only work for a few files before it becomes tedious.

Thankfully, modern PHP supports **autoloading**: we can programmatically tell PHP how to find a class that it doesn't know about yet. Technically we could do this in whatever way we like:

View Code

```
// registers an autoloader
// if a class name is used that PHP doesn't recognise
// it will call this function and pass in the class name
spl_autoload_register(function ($class) {
  // the function loads in a file
  // using the class name that was passed in
  require_once __DIR__ . "classes/{$class}.php";
});
```

But it's generally better if everyone does things in the same way, which is where the PSR-4 Autoloader standard comes in.

But before we can understand that, we need to know about **namespaces**.

8.2 Namespaces

Namespaces solve the problem of needing to work with two or more classes that happen to have the same name. For example, in a large app you might have a `Post` class that represents a blog post, but you might also have included a Slack package that includes a `Post` class that posts messages to Slack. It would be cumbersome to make sure that every single class used in a large app had a unique class name.

The most everyday use of namespaces is the file system on your computer: you can have two files called exactly the same thing *as long as they're in separate directories.*

Namespacing in PHP is much the same idea. We assign each class to a namespace and then we can have two classes with the same name, *as long as they're in separate namespaces*. This means that when we use the class we need to tell PHP which namespace we are talking about.

We assign a namespace by adding a `namespace` declaration as the very first line of PHP:

```
<?php

namespace Blog\Data;

class Post { ... }
```

Now, when we want to use this class we'll need to use the namespace:

```
new Blog\Data\Post();
```

If we use a class many times inside another file, we can put a `use` statement at the top to tell it to always use a specific namespaced class:

```
use Blog\Data\Post;

// further down the file
new Post(); // actually new Blog\Data\Post()

// we can use the other namespaced Post class
// we just need to use the full namespace
new Services\Slack\Post();
```

Generally we'll use the same class multiple times inside a file, so this saves a lot of typing.

If the class you want to use is in the *same* namespace as the current class you don't even need a `use` statement.

You can **alias** a class to give it a different name in the file you're working in. This can be particularly useful if you have two classes which share the same class name but are in different namespaces:

```
use Blog\Data\Post;
use Services\Slack\Post as SlackPost;

new Post(); // actually new Blog\Data\Post()
new SlackPost(); // actually new Services\Slack\Post()
```

The static `class` property

Sometimes it's useful to get the fully namespaced name of a class as a string (e.g. in a configuration file). All classes have a static `class` property:

```
$class = Services\Slack\Post::class;
dump($class); // "Services\Slack\Post"
```

This is particularly useful as backslashes need escaping in strings, meaning if you were to write the string out by hand you'd have to write:

```
$class = "Services\\Slack\\Post"; // need to escape every backslash
```

It also means you'll get an error if you try and use a class that doesn't exist.

8.3 PSR-4 Autoloading

The PSR-4 autoloading spec basically ties namespaces to a directory structure within a specified directory. Composer can do all the work of autoloading for us, but we do need to set it up.

First, create a directory called `app`. Then edit the `composer.json` so that it looks like this:

composer.json View Code

```
{
    "autoload": {
       "psr-4": {"App\\": "app/"}
    },
    "require": {}
}
```

Here we've told Composer that any namespace starting with the root `App` should look for files in the `app` directory. We could use anything as the root namespace or directory name, but this is the convention.

Next, run `composer dump`: this regenerates the Composer autoload file for us[1].

Now, as long as we stick to the following rules, we won't need to require any other files manually:

1. One class per file, where the file name is the same as the class name (case sensitive)
2. Put all of our classes in the `App` root namespace
3. If we add directories inside the `app` directory (for extra organisation), they add an extra level to the namespace

Because namespaces and classes in PHP are usually capitalised and, with PSR-4, the directory and filenames match the namespace/class naming, all the files and directories inside `app` will also be capitalised.

For example, if we had a class called `Post` that just sat inside the `app` directory, it should be in a file called `Post.php` and have the namespace `App\Post`. If we had a class `Post` that did something with Slack we could create a directory `app/Slack` and then put the file `Post.php` in it with the namespace `App\Slack\Post`.

8.4 Additional Resources

- PHP: Namespaces Overview
- hrefhttps://www.php-fig.org/psr/psr-4/PSR-4: Autoloader Spec

[1]Technically, it gets rid of the existing autoload file and then creates a new one – hence the name

Chapter 9

Object-Oriented Programming

So far, all of the PHP code[1] you've written has been "procedural": start at the top of a file, run through it, maybe call a few functions as you go, and then finish at the end. This is fine for simple programs or when we're just working inside an existing system (e.g. WordPress), but it doesn't really scale to larger applications.

The problem comes because we need to manage **state**: keeping track of all the values in our code. For a large app you could easily have thousands of values that need storing. Naming and keeping track of all these variables would become a nightmare if they were all in the same global scope.

Object-Oriented Programming[2] (OOP) is one way to make this easier. We keep functions and variables that are related to each other in an object. We then get different objects to talk to one another, so that rather than having lots of global variables and functions, all of our data lives inside relevant objects. This is a very different way of thinking about code, but can be very elegant if you can wrap your head around some of the intricacies we'll be covering over the next few days.

[1] And most of the JavaScript too – except for event handlers
[2] "Oriented" not "Orientated".

The Unusual History of PHP

PHP has a long and complicated history. The first version of PHP wasn't even a programming language, it was just simple templating language that allowed you to re-use the same HTML code in multiple files.

Over the years PHP morphed into a simple programming language and then into a modern object-oriented programming language. However, it wasn't really until 2009, with the release of PHP 5.3, that PHP could truly be considered a fully object-oriented language.

Because of this gradual change, older PHP frameworks and systems (such as WordPress) were originally written using non-OO code, which is why they still contain a large amount of procedural code.

PHP gets a lot of flack for not being a very good programming language and a few years ago that was perhaps a valid criticism. But in recent years, particularly with the release of PHP 7, it's just not true anymore. It certainly still has some issues, but nothing that a few libraries can't get around.

> *There are only two kinds of languages: the ones people complain about and the ones nobody uses*
>
> - Bjarne Stroustrup, Creator of C++

9.1 Bootstrapping

Many object oriented languages *only* use objects. For example in Java everything lives inside a class and you specify which class your app should create first when you run it.

Because of PHP's history we always need a little bit of procedural code to get our objects up and running. This is often called the **bootstrap** file:

bootstrap.php View Code

```
// include Composer for packages/autoloading
require_once __DIR__ . "/vendor/autoload.php";

// bootstrap code
```

```
// uses the Application class in the App namespace
$app = new App\Application();
$app->start();
```

Once we've created our first object the idea of object-oriented programming is that we **use objects from that point onwards**. Each object might use a few objects itself, which in turn might use a few more objects, creating a complex cascade of behaviours:

```
Application->start()
    Router->parseURL()
        URLParser->splitURL()
            String->separateWith()
    Request->generate()
        Request->getHeaders()
        Request->getBody()
        Request->parseBody()
            JSON->toObject()
    Application->callController()
        Controller->index()
            Post->all()
                Model->query()
                    DB->select()
                    DB->toArray()
            View->render()
                Blade->loadTemplate()
                Blade->parseTemplate()
                Blade->interpolate()
            Response->generate()
                Response->body()
                    JSON->fromObject()
```

9.2 Objectification

Say that our app includes some code to send an email. If we were using procedural code we would probably have a function called `sendMail` that we can pass various values to:

View Code

```
// a sendMail function
function sendMail($to, $from, $message) {
  // ... send email
}

// elsewhere
sendMail(
  "bob@bob.com",
  "hello@wombat.io",
  "Welcome to the best app for finding wombats near you"
);
```

But we might want to be able to customise more than just the to, from, and message parts of the email. Which means we'd either need to have a lot of optional arguments (which becomes unwieldy quickly) or rely on global variables:

View Code

```
// setup global variables
$to = null;
$from = null;
$message = null;
$bcc = [];

function sendMail() {
  // ... use the global variables
}

// elsewhere
$to = "bob@bob.com";
$from = "hello@wombat.io";
$message = "Welcome to the best app for finding wombats near you";
$bcc = ["ada@lovelace.dev", "donald@knuth.horse"];

sendMail();
```

But this is horrible: we have no way of preventing other parts of our code from changing these values and we would start having to use long variable names to avoid ambiguity in bigger apps.

So, we want to store the variables and the functionality together in one place and in such a way that values can't be accidentally changed. This is where objects come in:

View Code

```
class Mail
{
  private $to;
  private $from;
  private $subject;
  private $message;

  public function to($to)
  {
    $this->to = $to;
    return $this;
  }

  public function from($from)
  {
    $this->from = $from;
    return $this;
  }

  public function subject($subject)
  {
    $this->subject = $subject;
    return $this;
  }

  public function message($message)
  {
    $this->message = $message;
    return $this;
  }

  public function mail()
  {
    // ... code to send mail
    // we can use $this to access the values
```

```
  }
}


// elsewhere
$mail = new Mail();
$mail->to("bob@bob.com")
     ->from("hello@wombat.io")
     ->subject("A Wombat Welcome")
     ->message("Welcome to the best app for finding wombats near you")
     ->mail();
```

Now if we need to add additional fields, we can just add a property and setter method:

View Code

```
class Mail
{
  // ...other properties
  private $bcc = [];

  // ...other methods

  public function bcc($bcc)
  {
    $this->bcc = $bcc;
    return $this;
  }

  public function mail()
  {
    // updated to use $this->bcc as well
  }
}


// elsewhere
// can now add bcc as an option
$mail = new Mail();
$mail->to("bob@bob.com")
     ->from("hello@wombat.io")
     ->subject("A Wombat Welcome")
     ->message("Welcome to the best app for finding wombats near you")
     ->bcc([
```

```
        "ada@lovelace.dev",
        "donald@knuth.horse",
    ])->mail();
```

Now we can add an array of `bcc` addresses. And because we've set it to an empty array by default it wouldn't matter if we didn't set any – the code should still work as before.

This is a much nicer way of working with code.

9.3 Chaining

You'll notice that all the methods that *set* a value return `$this`. As a general rule of thumb, if a method doesn't have anything sensible to return, for example if it just sets a value, then you can return `$this`. `$this` represents the current object instance, which allows you to **chain** methods together:

```
$mail->to("bob@bob.com")->from("hello@wombat.io")
     ->bcc([
       "ada@lovelace.dev",
       "donald@knuth.horse",
     ])->send(
       "A Wombat Welcome",
       "Welcome to the best app for finding wombats near you"
     );
```

If we don't return anything from a method we get `null` back, which isn't very useful:

```
$result = null;
$result->doThing(); // PHP Fatal error: Call to a member function
↪  doThing() on null
```

By returning `$this` we get back the object instance (in the above case, the thing stored in `$mail`) each time, meaning we can call another method straight away.

If the methods hadn't returned `$this` we'd have had to write it using the `$mail` variable each time:

```
$mail->to("bob@bob.com");
$mail->from("hello@wombat.io");
$mail->bcc([
    "ada@lovelace.dev",
    "donald@knuth.horse",
]);
$mail->send(
    "A Wombat Welcome",
    "Welcome to the best app for finding wombats near you"
);
```

You can only use chaining on "setter" methods: ones that change a property on the object but don't have an obvious return value. The whole purpose of "getter" methods is to return a specific value, so it wouldn't make sense to use chaining on those.

9.4 Object-to-Object

In OOP objects use other objects to get things done. The key skill of OOP is getting the right objects to talk to one another.

For example, say that you wanted to send some emails to a mailing list. Rather than having the mailing list object do all of the work, we could instead have one object that deals with keeping track of who is in the mailing list and another that deals with sending the email. This is known as the "single responsibility principle": objects shouldn't try to do more than one sort of thing.

For example, we've already got a `Mail` class that just deals with creating and sending an email:

View Code

```
class Mail
{
  private $to;
  private $from;
  private $message;
  private $subject;

  public function to($address)
  {
```

```
    $this->to = $address;
    return $this;
  }

  public function from($address)
  {
    $this->from = $address;
    return $this;
  }

  public function subject($subject)
  {
    $this->subject = $subject;
    return $this;
  }

  public function message($message)
  {
    $this->message = $message;
    return $this;
  }

  public function mail()
  {
    // ... sends the mail
    dump("Sending using local mail server: \"{$this->subject}\" to
    ↪  {$this->to}, from {$this->from}, saying {$this->message}");
  }
}
```

It has various methods for adding the necessary email fields and another for actually sending the mail (well, pretending to).

The mailing list class can be relatively simple too – it just keeps track of all the subscribers and has a method for sending an email to each of them:

View Code

```
class MailingList
{
  private $subscribers = [];

  public function __construct($subscribers)
  {
```

```
    $this->subscribers = $subscribers;
  }

  // the send method takes a mail object as an argument
  public function send($mail)
  {
    // go through each subscriber one at a time
    foreach ($this->subscribers as $subscriber) {
      // use the passed in Mail object
      // update just the to field each time
      // then send the mail
      $mail->to($subscriber)->mail();
    }
  }
}
```

But notice that the actual sending of the mail is done by calling methods on a `Mail` object that has been passed in.

We could write something like the following to get it working together:

View Code

```
// create the mailing list
$mailinglist = new MailingList([
  "a@b.com",
  "c@d.com",
  "e@f.com"
]);

// setup the message
$mail = new Mail();
$mail->from("fish@flap.com")
     ->subject("Faces")
     ->message("Hi, I like your face");

// send the message to everyone on the mailing list
$mailinglist->send($mail);
```

By passing the `Mail` code *into* the mailing list class we can **compose** their behaviour to create a more complex behaviour[3].

[3] We could probably split up the `Mail` class up even further so that the sending code was separate

9.5 The Law of Demeter

The "Law of Demeter" is a guideline for OOP about how objects should use other objects. Expressed succinctly:

> *Each object should only talk to its friends; don't talk to strangers*

In practice, this means that an object should only call methods on either itself or objects that it has been given/created. You should avoid calling a method which returns an object and then calling a method on that object: it requires too much knowledge about other objects.

View Code

```
// this is fine
$this->doThing();

// this is fine
$this->mailer->send("Hello");

// this is not good
// the object needs to know how libraries
// and how books work
// this is *not* chaining
// get back a different object each call
$this->library->firstBook()->author();

// even worse
// using properties directly
// should stick to method calling
$this->library->books[0]->author;
```

You should be careful using chaining (returning `$this` from a method). If the method returns a *different* type of object then it's easy to break the Law of Demeter without realising it.

9.6 Additional Resources

- PHP Enthusiast: Chaining Methods and Properties
- Wikipedia: The Law of Demeter
- The Laravel Bootstrap File

Chapter 10

Encapsulation

Once our apps get past a certain size it becomes impractical to try and remember every single bit of code you've written. You'll remember that when we were working with functional programming in JavaScript we got around this by **composing** small bits of functionality to create more complex behaviours and we've already seen how we can use objects to achieve something similar. However, we need to be more careful when using objects, as they are more complicated than individual functions. This is where the concept of **encapsulation** comes in.

Encapsulation is one of the key ideas behind OOP: we keep functions and variables that are related to each other in an object and then use **visibility** to limit which bits of code can access and change them.

In this way an object becomes a black box: objects can send **messages** to each other (by calling methods), but they shouldn't have any knowledge of the inner workings of other objects.

10.1 Visibility

Methods and properties in PHP can have three levels of visibility: **public**, **private**, and **protected**.

10.1.1 `public`

A **public** method can be called anywhere in the PHP code.

A **public** property can be read and changed from anywhere in the PHP code.

```php
class Open
{
    public $name = "Blah";

    public function doThing()
    {
        // ...does thing
    }
}

// elsewhere
$open = new Open();
$open->doThing(); // does thing

dump($open->name); // "Blah"
$open->name = "Pickle Rick";
dump($open->name); // "Pickle Rick"
```

10.1.2 private

A **private** method can only be called within the class it is declared in using `$this`.

A **private** property can only be read and changed within the class it belongs to by using `$this`.

```php
class Closed
{
    private $name = "Blah";

    private function doThing()
    {
        // ...does thing
    }

    public function doOtherThing()
    {
        // can access private property with $this
        $name = $this->name;

        // can call private method with $this
        $thing = $this->doThing();
```

```
        // ...do other thing
    }
}

// elsewhere
$closed = new Closed();

// can't call doThing without $this — i.e. outside class
$closed->doThing(); // PHP Fatal error: Call to private method
↪  Closed::doThing()
$closed->doOtherThing(); // does other thing

dump($closed->name); // PHP Fatal error: Cannot access private property
↪  Closed::$name
$closed->name = "Pickle Rick"; // PHP Fatal error: Cannot access
↪  private property Closed::$name
```

10.1.3 `protected`

A **protected** property/method can only be used within the class it is declared in and any class that inherits from it. We'll look at inheritance later.

10.2 Encapsulation

We can think of `public` properties and methods as the surface-area of a class and the `private` properties and methods as the insides. The smaller the surface-area, the less we have to remember to use the class. Remember, if all our insides are outside then shit goes everywhere.

As a general rule we want to make as much `private` as possible. Many of your methods will be `public` (definitely the getters and setters), but if the method is only useful internally then you should make it `private`.

In particular there is generally no reason to make properties `public`: we should write getter and setter methods to access these values:

```
class Closed
{
    private $name = "Blah";

    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }

    public function getName()
    {
        return $this->name;
    }
}

// elsewhere
$closed = new Closed();
dump($closed->getName()); // "Blah"
$closed->setName("Pickle Rick");
dump($closed->getName()); // "Pickle Rick"
```

This probably seems like a lot of extra code, but it will help us write "safe" code.

10.3 Writing Safe Code

Another aspect of encapsulation is making sure that when other bits of code use our classes we do everything we can to stop things going wrong *inside* our class. If the person using our class has to look inside the class to see what's gone wrong, then we've broken encapsulation.

For example, going back to our `Mail` class, if the `$to` property was `public` it would be easy to set it to *any* value:

```
$mail = new Mail();
$mail->to = 12;
```

This would then cause issues when we call the `mail()` method later on: it will try to send an email to a number and probably throw an error. It will look like something has gone wrong *inside* `Mail`, when really it was the usage of `Mail` – setting the `$to` property to a number – that caused the issue.

If all the properties are `private` we can protect against this:

View Code

```
class Mail
{
  private $to;
  // ... other properties

  public function to($address)
  {
    if (filter_var($address, FILTER_VALIDATE_EMAIL) === $address) {
      $this->to === $address;
      return $this;
    }

    throw new InvalidArgumentException("Invalid email address");
  }

  // ...other methods
}
```

Now the user of the class *has* to use the `to()` method – as the property is `private` – and if they pass in something that is not a valid email address they will get en error message **that points to where they called `to()`**. In other words, the error will not be *inside* the `Mail` class and encapsulation is preserved.

Exceptions

In the above example we **throw** an **Exception**. An Exception is just an error that will stop the code from running, just as if we'd written some invalid syntax or called a method that doesn't exist.

PHP has various types of custom exception that you can throw yourself.

If you're expecting an error you can also **catch** exceptions: this allows the code to continue running when things go wrong, but in a controlled fashion.

10.4 Type Safety

PHP can do some of the work for us when it comes to checking that the right sorts of values are being passed to our methods.

As a general rule methods should only accept arguments of a specific type and return arguments of a specific type. We can enforce this using "type declarations"[1].

For example, we can add type declarations to the `to()` method:

```
public function to(string $address) : Mail
{
    // ...code as above
}
```

The parameter (`$address`) is given an explicit type (`string`). We also add a "return" type – the type of value the method returns – after the parameter brackets. In this case the `to()` method returns `$this`, which is an instance of a `Mail` object, so we use the class name as the return type.

The possible type declarations are:

- **A class name**: an instance of that class
- `int`: for numbers that have to be whole (e.g. a limit of a `for` loop)
- `float`: for any numbers (an integer passed to `float` *will* work)

[1] In previous versions of PHP these were called "type hints"

- `string`
- `bool`
- `array`
- `callable`: a closure

Now if you try and call a method and pass in/return the wrong type of value PHP will throw an error.

10.4.1 `void`

If a method doesn't return anything, we can use the special `void` return type:

```
// method doesn't return anything
// so use void return type
public function mail() : void {
    // ...mail code
}
```

But generally it's better to return `$this` if there's no obvious choice.

Strict Typing

By default PHP will coerce scalar values to the appropriate type if it sees a type declaration. For example passing `12` (a number) to the `to()` method would convert the number into a string (`"12"`). We can enable "strict" typing to disable this behaviour. To do this we add a declaration to the file that *calls* the relevant method:

```
<?php

declare(strict_types=1);
```

This should go as the very first line after the PHP opening tag (before any `namespace` declarations).

Once strict typing is turned PHP will throw an error if the types do not match.

10.5 Type Systems

Being able to create our own "types" using classes lets us create **type systems**: we can start to make guarantees that our code will always work.

Consider the `send()` method of the `MailingList` class:

```
public function send($mail)
{
  // go through each subscriber one at a time
  foreach ($this->subscribers as $subscriber) {
    // use the passed in Mail object
    // update just the to field each time
    // then send the mail
    $mail->to($subscriber)->mail();
  }
}
```

It currently goes over each subscriber and then calls the `to()` and `mail()` methods on the `$mail` object that has been passed in. As is, we could pass in *any* type of object to `send()`:

```
$person = new Person("Mark", "Wales", "1984-04-16");

// oops - that's not going to work
$mailinglist->send($person);
```

If we ran the above code we'd get an error inside the `MailingList` class when we try to call `to()` on it:

```
Call to undefined method Person::to()
```

However, with one simple change we can restore encapsulation:

```
// add the Mail type declaration
public function send(Mail $mail)
{
  foreach ($this->subscribers as $subscriber) {
```

```
    // has to be a Mail object
    // so this is guaranteed to work
    $mail->to($subscriber)->mail();
  }
}
```

Now we've guaranteed that the thing being passed in is of the type `Mail`. And we know that `Mail` objects have a `to()` method that takes a `string` and returns itself. And we know that a `Mail` object has a `mail()` method that doesn't take any arguments. Now if the user tries to pass in the wrong sort of thing:

```
TypeError: Argument 1 passed to MailingList::send() must be an instance
↪  of Mail
```

Encapsulation has been restored. Everything is good again[2].

10.6 Additional Resources

- PHP: Strict Typing
- Wikipedia: Information Hiding

[2] Everything is *not* good again

Chapter 11

Polymorphism

What if we wanted to be able to sometimes send our mailing list emails using the server's built-in mail program and sometimes using MailChimp?[1] We can't have a type declaration for two different classes, but if we don't limit the type at all then we loose encapsulation.

This is where the idea of **polymorphism**[2] comes in. Polymorphism is when two *different* types of object share enough in common that they can take each other's place in a specific context.

There are two ways to enforce this in most OO languages:

- **Interfaces**: when an object implements a defined set of methods.
- **Inheritance**: when an object can inherit methods/properties from another object, creating a hierarchy of object types.

11.1 Interfaces

One way we can take advantage of polymorphism is to use "interfaces". An **interface** is a list of **method signatures** that a class can say it conforms to. It is a contract: if a class implements an interface then we are guaranteed that it has a certain set of methods taking a specified set of arguments.

[1] We're very deliberately focussing on more abstract classes. A lot of books on OOP focus on classes representing concrete "things" in the real world, where there is a strong mapping between what the code does and what the things can physically do. However, it is easy to end up thinking this is the *only* way classes should be used, which inevitably means folks ignore all the OOP principles the second there's a more abstract bit of functionality.

[2] Oh, a heart so true our courage will pull us through you teach me, and I'll teach you Pokemon! Gotta catch 'em all!

Let's have a look at how `MailingList` uses the object that's passed in:

```
$mail->to($subscriber)->mail();
```

So it needs to be an object that has a `to()` method that takes a string and returns something that has a `mail()` method that takes no arguments and doesn't return a value.

We could create an interface for this:

MailInterface.php View Code

```
interface MailInterface
{
  public function to(string $address) : MailInterface;
  public function mail() : void;
}
```

The interface consists of **method signatures**: definitions of the methods that anything implementing the interface must stick to. We should add as much typing information to these as possible: if we didn't include the return type on `to()` then we couldn't guarantee that it returns something with a `mail()` method.

We can then say that `Mail` **implements** this interface:

Mail.php View Code

```
class Mail implements MailInterface
{
  // ...as before plus types
}
```

Our class already has matching methods, so this should work without needing to change anything. If we created a class that doesn't fully implement an interface then your code won't even run: you'll get an error straight away.

Finally, we need to update `MailingList`:

```
public function send(MailerInterface $mail)
{
    //...as before
}
```

Rather than using the `Mail` type declaration we've used the *interface* type instead. Our code should work as before.

But now we could also pass in *any* class that implements `MailInterface`. Which means we could create a `MailChimp` class:

MailChimp.php View Code

```
class MailChimp implements MailInterface
{
  // ...to, from, subject, message as before

  // completely different mail method
  public function mail() : void
  {
    // ... sends the mail
    dump("Sending using MailChimp: \"{$this->subject}\" to {$this->to},
    ↪  from {$this->from}, saying {$this->message}");
  }
}
```

And use it instead:

View Code

```
// setup the message
$chimp = new MailChimp();
$chimp->from("fish@flap.com")
      ->subject("Faces")
      ->message("Hi, I like your face");

// send the message to everyone on the mailing list
$mailinglist->send($chimp);
```

The `Mail` and `MailChimp` classes are **polymorphic** in respect to how they can be used inside `MailingList`. Either class in guaranteed to work because they both implement `MailerInterface`, which guarantees that the code in `MailingList` that uses the passed-in object will work.

It's worth noting that we *only* declare method signatures that are used inside `MailingList`: we could add method signatures for the `from()`, `subject()`, and `message()` methods too, but that would just unnecessarily limit the sorts of things that we can pass in. After all, we only use the `to()` and `mail()` methods, so those are the only ones we need to guarantee encapsulation.

A class can `implement` as many interfaces as it likes:

```
class MailChimp implements
    MailInterface,
    HttpInterface,
    InterspaceInterface
{
    //... code here
}
```

11.1.1 Message Passing

Interfaces are a core idea in OOP. But one that is often not talked about with beginner level books on OOP. Although it's called *Object*-Oriented Programming, it's not actually the objects that should get the focus, it's the **messages** that they can send to one another: the methods and their parameters.

The reason interfaces are such a powerful idea is because they focus solely on the messages and don't tell you anything about the implementation. This is important because of encapsulation. If we have to worry about how a specific object does something, then we can't treat it as a black box.

This is why we only declare `public` *methods* in an interface: `private` methods and properties are about how the class is implemented, whereas interfaces are all about how the classes talk to one another.

> *I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging" ... The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviours should be*
>
> - Alan Kay, Father of OOP

11.2 Inheritance

Inheritance lets us create a hierarchy of object types, where the **children** types inherit all of the methods and properties of the **parent** classes. This allows reuse of methods and properties but allowing for different behaviours.

For example, both `Mail` and `MailChimp` will be responsible for sending an email, so they will likely have some things behaviours in common (the `to()`, `from()`, `message()`, and `subject()` methods for example).

We could create a parent `Mailer` class that contains all of this repeated functionality:

Mailer.php View Code

```
class Mailer
{
  protected $to;
  protected $from;
  protected $message;
  protected $subject;

  public function to(string $address) : Mailer
  {
    $this->to = $address;
    return $this;
  }

  public function from(string $address) : Mailer
  {
    $this->from = $address;
    return $this;
  }
```

```
  public function subject(string $subject) : Mailer
  {
    $this->subject = $subject;
    return $this;
  }

  public function message(string $message) : Mailer
  {
    $this->message = $message;
    return $this;
  }
}
```

Notice that the properties are all `protected`: if they were `private` they would not be accessible inside any of the children classes.

We don't put the `mail()` method into the `Mailer` class, as it will be different for each implementation. We can then **extend** the `Mailer` class to copy its behaviour into `Mail` and `MailChimp`:

Mail.php View Code

```
// extends Mailer
// so no need to write to, from, subject, message
class Mail extends Mailer
{
  public function mail() : void
  {
    // ... sends the mail
    dump("Sending using local mail server: \"{$this->subject}\" to
    ↪  {$this->to}, from {$this->from}, saying {$this->message}");
  }
}
```

Now, in the `MailingList` class we can use `Mailer` as the type declaration. That means that depending on our mood,[3] we can send messages with either the local mail server or with MailChimp:

```
public function send(Mailer $mail)
{
    //...as before
}
```

A child class inherits all of the types of its parents, so a child of `Mailer` will pass this type check.

Again, we have polymorphism: different classes with some shared functionality that means they can be used interchangeably in a specific context.

11.2.1 Abstract Classes

But you can perhaps see a few issues with this. Firstly, you could create an instance of `Mailer` and pass that into `send()`. This would cause an issue because the `Mailer` class doesn't have a `mail()` method, so you'd get an error. Secondly, there's no guarantee that a child of `Mailer` has a `mail()` method: we could easily create a child of `Mailer` but forget to add it. This would lead to the same issue:

```
$mailinglist = new MailingList();

// oops, won't work
// Mailer doesn't have a send method
// so we'll get an error when send()
// tries to call it
$mailinglist->send(new Mailer());
```

This is where **abstract classes** come in. These are classes that are not meant to be used to create object instances directly, but instead are designed to be the parent of other classes. We can setup properties and methods in them, but they can't actually be constructed, only extended.

We can also create `abstract` method signatures. These allow us to say that a child class *has* to implement a method with the given name and parameters. We'll get

[3] Or possibly something more concrete

an error if we don't.

This gets round both issues: we won't be able to instance `Mailer` and we can make sure any of its children have a `mail()` method:

Mailer.php View Code

```
// declare the class abstract
// can't be instanced itself now
abstract class Mailer
{
  protected $to;
  // ...other properties

  public function to(string $address) : Mailer
  {
    $this->to = $address;
    return $this;
  }

  // ...other methods

  // abstract method
  abstract public function mail(string $subject, string $message) :
  ↪  void;
}
```

11.2.2 Overriding

Children classes can **override** methods and properties from parent classes. That means if we wanted to make it so that the `to()` method in the `MailChimp` class did something slightly different, then we could write a different `to()` method. As long as it has *the same method signature*[4] (i.e. excepts the same parameters), this will work:

[4]The `__construct()` method is an exception to this rule. As it is unique to the specific class it can have a different set of parameters.

MailChimp.php View Code

```
class MailChimp extends Mailer
{
  // override the to method from Mailer
  public function to(string $address) : Mailer
  {
    // additional code
    $this->doWeirdAPIThing();

    // same as before
    $this->to = $address;

    return $this;
  }

  // ...etc.
}
```

If necessary it's possible to stop a child class from overriding a method by adding the `final` keyword in front of it:

Mailer.php View Code

```
abstract class Mailer
{
  // ...etc.

  // not sure why you'd need to do this
  // but now a child class couldn't change this method
  final public function to(string $address) : Mailer
  {
    $this->to = $address;
    return $this;
  }
}
```

11.2.3 `parent`

Sometimes when we're overriding a method it can be useful to still have access to the parent object's version. For example, we might want to override the `to()` method of `Mailer`, but only to add a bit of functionality. We can do this by calling

the method name on `parent`. Make sure you pass along the arguments when you do this:

MailChimp.php View Code

```
class MailChimp extends Mailer
{
  public function to(string $address) : Mailer
  {
    $this->doWeirdAPIThing();

    // call the to method of the parent
    // passing in the address argument
    // has same method signature
    return parent::to($address);
  }

  // ...etc.
}
```

You can call the parent's constructor method with `parent::__construct()`.

11.2.4 Inheritance Tax

> *The object-oriented version of "Spaghetti code" is, of course, "Lasagna code": too many layers*
>
> - Roberto Waltman

If you read any books about OOP they'll focus a lot of their time on inheritance. While inheritance can be very useful, all of this attention means that it's often the technique that programmers reach for when they need to add similar functionality to multiple classes. And it will almost always lead to much more complicated code.

If you think about it, inheritance can easily break encapsulation. With poorly written inheritance structures it can sometimes be necessary to understand the inner-workings of all a class's parent classes. It can lead to situations where you become scared to change a line of code in the parent class in case it inadvertently affects a piece of code in one of the children classes. And the last thing you want to be writing is code that you're worried about changing.[5]

[5]This wisdom courtesy of many such experiences.

That's not to say inheritance isn't useful. It's totally fine to inherit well-documented code that frameworks or libraries provide, as in these cases you're generally adding one tiny bit of functionality to something that's much more complicated under-the-hood.[6] But if it's code that you've written, it's always worthwhile thinking "Do I really need to use inheritance?"

Sandi Metz[7] suggests not using inheritance until you have *at least* three classes that are *definitely* all using exactly the same methods. You should never start out by writing an abstract class: write the actual use-cases first and only write an abstract class if you definitely need one. If you do use inheritance then try not to create layers and layers of it: maybe have a rule that you'll only ever inherit through one layer.

11.3 Interfaces plus Inheritance

Any class can implement an interface, including parent classes. So our `Mailer` class could implement the `MailInterface`:

```
abstract class Mailer implements MailInterface
{
    // ...as before
}
```

Then the `MailingList` class could still use `MailInterface` as the type declaration:

```
public function send(MailerInterface $mail)
{
    //...as before
}
```

In general, interfaces are more flexible than inheritance, so they should be preferred for type declarations.

[6] Although some purists would say even in this case there are better alternatives: see the Active Record vs Data Mapper debate

[7] She's been doing OOP since it was invented in the 70s, so she probably knows what she's talking about

11.4 Composition

> *Prefer composition over inheritance*
>
> - The Gang of Four, *Design Patterns*

The idea of *composition over inheritance* is that rather than sharing behaviour with inheritance, we share it using interfaces and shared classes. If a lot of classes share the same implementation of a method, rather than using inheritance consider moving it into a separate class that they can all share. It might require a bit more code to get working, but it is much easier to make changes to.

For example, in the case of `Mail` and `MailChimp` we could pull the email specific code out in an `Email` class, which both classes can use. This separates the *generating* of an email from the *sending* of an email, which is probably no bad thing.

11.5 Additional Resources

- PHP Apprentice: Interfaces
- PHP Apprentice: Inheritance
- Wikipedia: Composition over Inheritance
- Practical Object-Oriented Design in Ruby: An intermediate book about OOP. In Ruby, but all the key concepts work in PHP too.
- Wikipedia: The SOLID Principles of OOP: Quite advanced, but incredibly useful if you can get your head round it.
- YouTube: Sandi Metz Videos: If you're doing OOP in a few years time, watch all of these. There's a lifetime of experience in these talks.

Read-Only: Chapter 12

Static Methods & Properties

Classes are primarily used for creating object instances. But sometimes it's useful to write some functionality about the object type instead of object instances.

For example, if we have a `Person` class we might want to write a bit of functionality that given an array of `Person` objects gives us back an array of just last names. We could write a `lastNames()` function in global scope, but then it's not associated with the `Person` class.

Instead, we will write a `static` method: a method that belongs to the class itself rather than to an object instance.

View Code

```
class Person
{
  // static methods (and properties) at top
  public static function lastNames($people)
  {
    return array_map(function ($person) {
      // best to use methods to get values
      return $person->lastName();
    }, $people);
  }

  // ...non-static methods and properties at bottom
}

// elsewhere, pass in an array of Person objects
Person::lastNames([$oli, $pete, $nicola, $tom]);
```

Now it is clear that the `lastNames()` method has something to do with `Person` objects.

Paamayim Nekudotayim

The `::` symbol is also known as the "Paamayim Nekudotayim", which is Hebrew for "double colon". This can lead to the somewhat mystifying error:

```
PHP expects T_PAAMAYIM_NEKUDOTAYIM
```

All it's saying is you need a `::` somewhere.

The Zend Engine, which was behind PHP 3.0 and all subsequent releases, was originally developed at the Israel Institute of Technology.

12.1 The `static` Keyword

Sometimes it's useful to be able to refer to the class you're currently working in. We can use the `static` keyword to do this.

For example, we might want a separate `static` method that returns the last names alphabetically – but it would be repetitive to rewrite the method we've already got:

View Code

```
class Person
{
  // a method to returns just last names
  public static function lastNames($people)
  {
    return array_map(function ($person) {
      return $person->lastName();
    }, $people);
  }

  // a method to get the last names sorted
  // uses the existing static lastNames method
  public static function sortedLastNames($people)
  {
    // call the lastNames method of this class
```

```
    $lastNames = static::lastNames($people);

    // sort alphabetically then return
    sort($lastNames);
    return $lastNames;
  }

  // ...non-static methods and properties at bottom
}
```

`static` vs `self`

You will sometimes see `self` instead of `static` to reference the current class. Using `static` in this way was only added in PHP 5.3, so a lot of older code uses `self`.

If you're not using inheritance, then it doesn't make any difference which one you use. If you do, then `self` refers to class that it is *written* in and `static` refers to the class it is *called* in (which might be different from where it was written if you're using inheritance).

12.2 Static Properties

You can also have `static` properties. Because they belong to the class they exist for as long as your code is running, so you can use them for storing values that you want to keep around – like global variables, but with a dedicated home. This is very useful for caching values.

Say we needed to create a `$renderer` object that all our object instances can use to render... something. We could use a `static` property to store it, so that we only create it one time:

View Code

```
class Post
{
  // lets us store the value for the life of the app
  private static $renderer;

  public static function setup()
  {
```

```
    // only set value if it's not been set yet
    if (!static::$renderer) {
      static::$renderer = doComplexThingThatTakesLotsOfEffort();
    }
  }

  public function __construct()
  {
    // run setup everytime object created, will only do something the
    ↪  first time
    static::setup();
  }

  public function render()
  {
    // can use
    return static::$renderer->render();
  }
}
```

`static` properties can be very useful for things like caching, although there are often other alternatives.

12.3 Additional Resources

- PHP Apprentice: Static

Glossary

- **Class**: An abstract representation of an object instance
- **Composer** PHP's package management system
- **Dependency Injection** Rather than hard-coding a dependency on a specific class, taking advantage of polymorphism and passing in a class that implements a specific interface.
- **Encapsulation** Keeping the inner-workings of a class private so that they can only be accessed by sending appropriate messages (by calling methods)
- **Inheritance** A way to share code between different classes. Should be used with caution as it breaks aspects of encapsulation.
- **Instance**: An object is an instance of a specific class with its own set of properties
- **Interface**: A list of method signatures that tell us how to talk to an object that implements it
- **Namespace**: A set of classes where each class has a unique name. We can have two classes with the same name as long as they are in different namespaces.
- **Polymorphism**: When two different classes can be used interchangeably in a specific context because they share the relevant method signatures
- **Single Responsibility Principle** The principle that an object/class should only do one sort of thing. We get more complex behaviour by composing different objects – similar to function composition in functional programming languages.
- **Static**: A property or method that belongs to a class rather than an object instance

Colophon

Created using TEX

Fonts

- Feijoa by Klim Type Foundry
- Avenir Next by Adrian Frutiger, Akira Kobayashi & Akaki Razmadze
- Fira Mono by Carrois Apostrophe

Written by Mark Wales

∞

October 27, 2020