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

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.

Copying and pasting code from a PDF can mess up indentation. For this reason large blocks of code will usually have a [View on GitHub] link underneath 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 Basics

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. 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.

For now we're just going to run PHP in the command-line like we did in Week 3 with JavaScript. PHP doesn't have a REPL built-in like Node, so we always need to run a file:

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

1.1 Hello, World!

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

```
<?php

echo "Hello, World!";
```

[View code on GitHub]

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 var_dump

As mentioned above `echo` isn't that useful for values that aren't strings, so we'll be using `var_dump` instead. This outputs not only the value we're interested but also relevant information about it.

For example, running the following:

```
var_dump("Hello");
var_dump(12 + 12);
var_dump(true);
```

Would give us:

```
string(5) "Hello"
int(24)
bool(true)
```

`var_dump` *is* a function, so we call it and pass it values.

`var_dump` isn't brilliant for working with complex values, so we'll switch to using a nicer way of doing it once we've learned about using PHP libraries.

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?
```

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

[View code on GitHub]

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.

> **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.

1.4 Additional Resources

- PHP: `var_dump`
- PHP: Variable Basics
- Wikipedia: PHP
- Wikipedia: Perl

Chapter 2

Basic Types

PHP has the same basic types as JavaScript: numbers, strings, and booleans. 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:

```
var_dump(12 + 12); // int(24)
var_dump(0.1 + 0.2); // float(0.3) - hurrah!
```

[View code on GitHub]

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

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
```

[View code on GitHub]

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 `.`:

```
var_dump($firstName . $lastName); // "CasperSpooky"
var_dump($firstName + $lastName); // PHP Warning - A non-numeric value
↪  encountered
var_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;
}

var_dump($total); // int(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;
}

var_dump($total); // int(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);

var_dump($total); // int(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;
}

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

[View code on GitHub]

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.

The following will cause an error as `$message` is assumed to be in local scope:

```
$message = "Hello";

function sayHello($name) {
    return "{$message} {$name}";
```

```
}

var_dump(sayHello("Jim")); // " Jim"
```

[View code on GitHub]

This means if we want to access a non-local variable we need to use the `global` keyword:

```
$message = "Hello";

function sayHello($name) {
    global $message;
    return "{$message} {$name}";
}

var_dump(sayHello("Jim")); // "Hello Jim"
```

[View code on GitHub]

This can actually be seen as a positive, as it makes it harder to write impure functions by accident.

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

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

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]);

var_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:

```
$message = "Hello";

$say = function ($a) use ($message) {
    return "{$message} {$a}";
};

$result = $say("Wombat");
var_dump($result); // string(12) "Hello Wombat"
```

4.3 Strict Types

As a general rule functions should only accept arguments of a specific type and return arguments of a specific type: e.g. `add` should only accept numbers. We can enforce this using "type declarations"[1].

First we turn on strict typing:

```
<?php

declare(strict_types=1); // always first line after opening tag
```

Next we add type declarations to our function:

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

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

Each parameter (`$a` and `$b`) is given an explicit type. We also add a "return" type - the type of value the function returns - after the parameter brackets.

The parameter types and return types needn't match. Here's a function that takes a string (`$str`) and integer (`$times`) as parameters and returns a string:

```
function repeat(string $str, int $times) : string {
    $output = "";

    for ($i = 0; $i < $times; $i += 1) {
        $output .= $string;
    }

    return $output;
}
```

The possible type declarations are:

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

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

4.3.1 `void`

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

```php
// function doesn't return anything
// so use void return type
function echoNewLine(string $str) : void {
    echo "{$str}\n";
}
```

Generally it's best to write functions that do return useful values, but there are some cases where this is not practical.

4.4 Additional Resources

- PHP: Functions
- PHP: Callable
- PHP: Strict Typing

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 = $values[4]; // last item in this array
var_dump($first); // 1
var_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 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",
];

var_dump($assoc["lastName"]); // string(5) "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:

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

var_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.

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

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

You can also get the key:

```php
$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:

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

```php
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). We'll be looking at Laravel's `Collection` class later, which lets you iterate over an array in a much nicer way.

5.5 Additional Resources

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

Chapter 6

Regex

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 it's 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 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:'
```

6.1.2 Groups & Ranges

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

Range	Description
`[a-z]`	all lowercase letters
`[A-Z]`	all uppercase letters
`[0-9]`	all numbers
`[abc]`	'a', 'b', or 'c'
`[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
`[^abc]`	**not** 'a', 'b', or 'c'

A group is a set of characters wrapped with square brackets. A range, which must always be used *inside* a group, represents a 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
```

6.1.3 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

These can generally be used in regular strings too.

6.1.4 Character Classes

Character classes are shortcuts for specific ranges.

Class	Description
`\s`	whitespace
`\S`	**not** whitespace
`\w`	word (`[A-Za-z0-9_]`)
`\W`	**not** word
`\d`	digit
`\D`	**not** digit

For example:

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

6.1.5 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.

```
/.+@.+/          - an '@' symbol with some number of other characters
↪  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
```

6.1.6 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
```

[View code on GitHub]

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.

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.

[1] "Back references" are particularly useful

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.

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 them.

6.5 Additional Resources

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

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

Chapter 7

Classes in PHP

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:

```
<?php

class Person
{
  // declare class properties at the top
  private $firstName;
  private $lastName;
  private $dob;

  // the constructor method
  public function __construct(string $firstName, string $lastName,
  ↪  string $dob)
  {
    // we can use $this to reference object properties
    $this->firstName = $firstName;
    $this->lastName = $lastName;
    $this->dob = $dob;
  }

  public function getName() : string
  {
    // use $this to read properties
    return "{$this->firstName} {$this->lastName}";
  }
```

```
  public function getAge() : int
  {
    // do some date stuff
    $date = new DateTime($this->dob);
    $now = new DateTime();
    return $now->diff($date)->y;
  }
}
```

[View code on GitHub]

As you can see, it's much the same as a JavaScript class: we have the `class` keyword followed by the name of the class and there are some functions inside the class.

As in JavaScript, we call functions that belong to an object **methods** and the values **properties**.

However, there are some things we've not seen in JavaScript: type declarations, the words `private` and `public`; and we declare our properties outside of the constructor method. We'll look at these in more detail shortly.

Here's how we'd use our class:

```
// as in JavaScript, use the "new" keyword to create
// a new object instance
$jim = new Person("Jim", "Henson", "1936-09-24");

// create another Person
$frank = new Person("Frank", "Oz", "1944-03-25");

// each object has its own properties
// so getAge() will return different values for each one
$jim->getAge();
$frank->getAge();
```

[View code on GitHub]

You can see that where we'd write a dot in JavaScript (`jim.getAge()`), we write an arrow in PHP (`$jim->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 $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.

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

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

  // ...setTown and setCountry as above

  // a getter function
  public function getAddress() : string
```

```
  {
    // you can call methods too
    return implode(", ", $this->assigned());
  }

  private function assigned() : array
  {
    return array_filter([
      $this->street,
      $this->town,
      $this->country,
    ]);
  }
}
```

[View code on GitHub]

7.1.1 Returning `$this`

If your method doesn't have anything to return, for example if it just sets a value, then you can return `$this`: it will give back the current object instance to the user, meaning that they can **chain** such methods together:

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

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

  public function setPostcode(string $postcode)
  {
    // $this represents whichever object we're working on
    $this->postcode = $postcode;
    return $this;
  }
```

```
}


// elsewhere
$address = new Address();

// because setStreet returns $this we can chain methods
$address->setStreet("1630 Revello Drive")->setPostcode("BS3 9BR");
```

[View code on GitHub]

For now we'll leave the return type for these methods blank, but when we look at object-oriented programming later we'll see that we can add useful information here.

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();
```

However, 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();
```

7.2 Properties

In PHP we declare all of the properties that our class uses at the top of the class. This makes it easy to see which values are available. It also allows us to set default values easily:

```
class DatabaseConnection
{
  // properties with sensible defaults
```

```
  private $engine = "mysql";
  private $host = "127.0.0.1";
  private $port = 3306;

  // properties that don't have sensible defaults
  private $username;
  private $password;

  // ... code to do database stuff
}
```

[View code on GitHub]

Typed Properties

As of PHP 7.4 (released in late 2019) you can add types to class properties:

```
private string $engine = "mysql";
private string $host = "127.0.0.1";
private int $port = 3306;
```

We won't be covering typed properties in any detail, as they're still very new, but you can read more here.

7.3 Visibility

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

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.

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`.

```
class DatabaseConnection
{
```

```
  // properties with sensible defaults
  private $engine = "mysql";
  private $host = "127.0.0.1";
  private $port = 3306;

  // properties that don't have sensible defaults
  private $db;

  // set the database
  public function setDB(string $db)
  {
    $this->db = $db;
    return $this;
  }

  // returns the DB connection
  public function connection(string $username, string $password)
  {
    return new PDO(
      $this->pdoSettings(),
      $username,
      $password
    );
  }

  // this method is private - it's only useful inside the class
  private function pdoSettings() : string
  {
    $parts = implode(";", [
      "host={$this->host}",
      "port={$this->port}",
      "dbname={$this->db}",
    ]);

    return "{$this->engine}:{$parts}";
  }
}
```

[View code on GitHub]

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).

In most instances it's good practice to make all of your properties private and then

use "getter" and "setter" methods if you need to be able to change the values outside the class.

7.4 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 gives us an alphabetic array of last names. We could write a `lastNames()` function, 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.

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

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

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

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

[View code on GitHub]

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

Remember, `private` properties and methods can only be accessed using `$this`, which doesn't have a value in `static` methods. So you'll need to use getters/setters just as you would if the function was written outside of the class.

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.

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:

```
class Person
{
  public static function lastNames(array $people) : array
  {
    // call the getLastNames method of this class
    $lastNames = static::getLastNames($people);

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

  // a private static method
  private static function getLastNames(array $people) : array
  {
    return array_map(function ($person) {
      return $person->lastName();
    }, $people);
  }
```

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

[View code on GitHub]

`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).

You can also have `static` properties. Because they belong to the class they always exist, so you can use them for storing values that you want to have around. 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:

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

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

[View code on GitHub]

Privately declared `static` variables are accessible from inside object instances.

7.5 Additional Resources

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

Chapter 8

Standard Library

8.1 Dates

PHP makes working with dates quite pleasant as long as you use the `DateTime` class (and its relatives).

```
// a date now
$dt = new DateTime();

// output as a string
$dt->format("Y-m-d H:i:s"); // e.g. "2020-02-04 15:13:59"

// a date for a specific time
$dt = new DateTime("5th January 2020, 12:04:03");
$dt->format("d/m/Y @ H.i.s"); // "05/01/2020 @ 12.04.03"
```

[View code on GitHub]

The `format` method takes a string. You can find out all the possible parts on date documentation.

You can also do date maths using the `DateInterval` class:

```
// an interval representing:
// 2 years, 3 months, 5 days, 6 hours, 2 minutes and 6 seconds
$interval = new DateInterval("P2Y3M5DT6H2M6S");

// a date now
```

```php
$dt = new DateTime();

// add an interval to an existing date
// new DateTime, 2 years, 3 months (etc.) in the future
$future = $dt->add($interval); // new DateTime object
$future->format("Y-m-d H:i:s"); // e.g. "2022-05-09 21:24:08"

// subtract interval from a date
$past = $dt->sub($interval); // new DateTime object

// work out the difference between two dates
$birthday = new DateTime("1984-04-16");
$difference = $dt->diff($birthday);

// difference in years
$difference->y; // 35
```

[View code on GitHub]

The period format is pretty strange for `DateInterval`. It starts with a `P` (for "period"), then using `Y`, `M`, and `D` for year, month, and day. Then there's a `T` for "time" followed by `H`, `M`, and `S` for hour, minute, and second.

∞

Although PHP's date functionality is better than some, it's still fairly common to use a library like `Carbon` to deal with dates and times in PHP.

Whatever you do, avoid the `date()`, `mktime()`, and `strtotime()` functions like the plague!

8.2 Additional Resources

- PHP: The `DateTime` Class
- PHP: The `DateInterval` Class

Chapter 9

Namespaces

So far we've had to put all of our classes into a single file, when ideally we'd like *one class per file.*[1]

9.1 `require_once`

We can import one PHP file into another using the `require_once` keyword:

```
// include the DatabaseConnection class
// assuming it's in the same directory
require_once "DatabaseConnection.php";

// create a new DatabaseConnection object
$db = new DatabaseConnection();

// get the connection
$connection = $db->setDB("tests")->connection("tests", "secret");
```

[View code on GitHub]

We give `require_once` a relative file path and it will be as if the contents of that file are included in place.

[1] It makes your code much more reusable

`require` and `include`

There is also the `require` command: this does the same as `require_once`, except you could accidentally load the same file in more than once - which you almost never want to do.

There is also `include` and `include_once` which do the same as the `require` equivalents, except if it can't find the file it will keep running and just show an error message. However, normally if we're trying to include a file we wouldn't want the code to run at all if it can't find the file, so `require` is preferred.

9.2 Naming Collisions

Large PHP apps can have hundreds (or even thousands) of classes. It's not uncommon for two classes to end up with the same name. For example, in a blog app you might have a `Post` class which deals with the data for each post on the site. But you might also have a `Post` class which posts an update to Slack each time a post is added.

We could, of course, be careful about the naming of each class, calling one `BlogPost` and the other `SlackPost`, but in large apps it can be tricky keeping track of every class name that you've used - and it becomes practically impossible when you have multiple developers working on the same app.

Even if we're really careful naming our classes, we don't have any control over the names of classes in PHP libraries that other people have written. It would be unrealistic to make sure that none of the class names you've used clash with those in any libraries that you might use.

The Bad Old Days

I lied just then. Back in the before-times, when PHP was still trying to find itself, you *did* have to use unique names for every single class - including the ones in libraries (which you had no control over). To get around this issue you would pick an almost definitely unique prefix (like your company name) and add it to the front of every single class in the app: `SmallHadronCollider_BlogPost`, `SmallHadronCollider_SlackPost`. Needless to say, this made the code where the classes were used very messy.

9.3 Namespaces

Namespaces were added to PHP 5.3 to avoid this problem. 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 at the top of the file:

```
namespace Blog\Data;

class Post { ... }
```

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

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

This might not seem any better than the old way of doing things (i.e. using `BlogPost`), but PHP also gives us the `use` keyword.

We can put a `use` statement at the top of a PHP file 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:

```
$class = Services\Slack\Post::class;
var_dump($class); // string(19) "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.

9.4 Autoloading

When I said earlier that PHP apps can contain thousands of classes you might have thought "Well that's going to be an awful lot of `require_once` statements". And, in fact, historically that's exactly what you'd have: a file called something like `load.php` which listed thousands of files. Every time you wanted to add a class you'd need to write it and then make sure you added it to the massive list.

Thankfully, things have moved on since then and PHP supports **autoloading**. This lets us tell PHP where to find a specific class based on its name and namespace. However, writing this code ourselves is unnecessary because we'd be much better using the Composer package manager to do it for us.

9.5 Additional Resources

- PHP Namespaces Explained
- Auto-Loading Classes

Chapter 10

Composer

Composer is PHP's **package manager** (like `npm` is for JavaScript). It lets us easily add code written by other people to our projects.

One of Composer's responsibilities is to set up autoloading of any libraries that it adds, that way you don't have to manually link to all the files that you use in your project. It's also very easy to setup Composer so it will autoload any classes that *you've* created.

10.1 Initialising

First we need to add Composer. Run the following in the project directory that you want to add Composer to:

```
# the -n bit stops it asking you a bunch of questions
composer init -n
```

This will add a `composer.json` file to your project.

10.2 PSR-4 Autoloading

Next we need to tell Composer to load our classes for us. We're going to use the PSR-4 namespace standard. Basically, this means that we pick a directory to be the "root" of our namespace, and everything from that point on is just based on directory names.

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

```
{
    "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.

Next, run `composer dump`: this, somewhat confusingly, generates an autoload file for us[1] in a directory called `vendor` (this is where Composer installs any libraries we might want to use).

We'll also need to create a file in the root of our project that looks like this:

```
require_once __DIR__ . '/vendor/autoload.php';

// ... code that uses the classes
```

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.

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

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`.

10.3 Libraries

As well as handling autoloading for us, Composer's main purpose is to let us use bits of code other people have written. Let's take a look at some useful libraries.

You can search for libraries on Packagist or Libraries.io.

> **Composer & Git**
>
> You don't want to add the `vendor` directory to your Git repositories, as it can be easily recreated by running `composer install`. So make sure you add `vendor/` to your `.gitignore` file.

10.3.1 `symfony/var-dumper`

We install this with `composer require symfony/var-dumper`. We then have access to the `dump()` function, which is a more useful version of `echo`:

```
dump([1, 2, 3, 4]);
/*
    array:4 [
      0 => 1
      1 => 2
      2 => 3
      3 => 4
    ]
*/

dump(new Person("Zazu"));
/*
    App\Person {
      -name: "Zazu"
    }
*/
```

It also adds the `dd()` function (for "debug and die"), which dumps the result and then immediately stops the PHP. This can be useful if you want to check something half-way through a process. Be careful, if you use `dd()` nothing after it will run.

See the VarDumper documentation for more information.

10.3.2 `Illuminate\Support\Collection`

This is part of Laravel, which we'll be covering later. It basically lets us handle arrays in a way which isn't utterly horrible.

We install it by running `composer require illuminate/support`. 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()`.

Generally collection methods return a new collection object. 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`:

```
use Illuminate\Support\Collection;

$numbers = new Collection([1, 2, 3, 4, 5]);

$even = $numbers->filter(function (int $n) : bool {
  return $n % 2 === 0;
});

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

[View code on GitHub]

`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:

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

$squared = $numbers->map(function (int $n) : int {
  return $n * $n;
});

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

[View code on GitHub]

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:

```php
$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(function (int $acc, int $val) : int {
  return $acc + $val;
}, 0);

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

[View code on GitHub]

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` contain just that key/property from each item in the collection:

```php
$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());
```

[View code on GitHub]

10.3.3 nesbot/carbon

The Carbon library makes working with dates in PHP much easier. We install it by running `composer require nesbot/carbon`[2]. Once it's installed we have access to the `Carbon\Carbon` class.

```php
use Carbon\Carbon;

// how old I am
dump(Carbon::createFromDate(1984, 4, 16)->age);

// the next summer olympics
dump((new Carbon("2016-08-05"))->addYears(4));
```

[View code on GitHub]

There are many more features listed in the Carbon documentation.

10.3.4 fzaninotto/Faker

Faker is a library that generates fake data for you. This can be useful for "seeding" databases: generating test data to make sure that everything's working. We install it with `composer require fzaninotto/faker`.

[2] If you've already installed `illuminate/support` this isn't necessary, as it's a dependency for that package

```
// use the factory to create a Faker\Generator instance
$faker = Faker\Factory::create();

// generate data by accessing properties
dump($faker->name); // a random name
dump($faker->address); // a random address
dump($faker->text); // some lorem ipsum text
dump($faker->email); // a random email address
```

[View code on GitHub]

Faker supports hundreds of different types of data: IP addresses, credit card numbers, dates, colours, images, &c.

The Case of the Missing Students?

When building the student preparation app for the course we created lots of fake users to make sure everything was working. Then, when real students started signing up, they didn't appear in the admin interface. It took a bit of sleuthing to work out what was going wrong: the query to fetch the students should have been checking for the student "type" field, but in actual fact was checking the student's name for the word "student". And it just so happened that every single test student that we'd added to the database had the name "Test Student <number>"! If we'd been using Faker to generate random names we'd have noticed the issue earlier!

10.4 Additional Resources

- More about PSR-4
- Composer

Chapter 11

Encapsulation

So far, all of the PHP code 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[1] (OOP) is one way to make this easier. The key idea behind OOP is **encapsulation**: we keep functions and variables that are related to each other in one place (an object) and then use visibility to limit which bits of code can access and change them.

An object is effectively a black box: objects can send **messages** to each other (by calling methods), but they need not have any knowledge of the inner workings of other objects.

[1] "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.

> *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

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++

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:

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

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

[View code on GitHub]

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:

```
// setup global variables
$to = null;
$from = null;
$message = null;
$subject = null;
$characterSet = "utf8";

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";
$subject = "A Wombat Welcome";

sendMail();
```

[View code on GitHub]

But this is truly 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:

```
class Mail
{
  private $to;
  private $from;
  private $characterSet = "utf8";

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

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

  public function send(string $subject, string $message)
  {
    // ... 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")
     ->send(
       "A Wombat Welcome",
       "Welcome to the best app for finding wombats near you"
     );
```

[View code on GitHub]

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

11.1 Object-Oriented Programming

In OOP objects use other objects to get things done. Rather than use variables and functions, pretty much everything is an object instance and we use properties and methods. The key skill of OOP is getting the right objects to talk to one another.

For example, say that we have various users on our website and we want to send them our regular mailing list email. In procedural code we'd probably write something like:

```
$userEmails = getUserEmailsFromDatabase();

foreach ($userEmails as $address) {
    sendEmail($address, "Subject", "Message");
}
```

In OOP it might look something like this:

```
$users = Users::all(); // get all the users
$email = new Email("Subject", "Message"); // create a new email
$mailing = new MailingList($users); // create a new mailing list
$mailing->sendEmail($email);
```

Each object represents a specific, *encapsulated*, bit of functionality, dealing with just the things that it needs to and nothing else. We still need to glue the objects together, but this code itself is generally inside other objects.

11.2 (Almost) Pure OO

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.

```
class App
{
  public function start()
  {
    // code that makes the app do stuff
  }
}

// the bootstrap code
$app = new App();
$app->start();
```

[View code on GitHub]

Once we've created our first object the idea of object-oriented programming is that we use objects from that point onwards.

11.3 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. 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.

```
// 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;
```

[View code on GitHub]

11.4 Types

We've already looked at "scalar types" in PHP: integers, floats, strings, booleans, arrays, &c.

In OOP we generally talk of objects as having the type of their class: e.g. a `Person` object instance is of the *type* `Person`.

PHP supports **type declarations** for object instances too. These let us say that the values passed to and returned from a function or method must be instances of a certain type.

For example, say we had a `MailingList` class with a `sendWith()` method. It would be useful to say that we can only pass `Mail` objects into this method[2]:

```
class MailingList
{
  private $emails = [];
  private $subject;
  private $message;
  private $from;

  // ...mailing list code: getters, setters, etc.

  // use the Mail type-declaration to
  // only allow Mail classes
  // returns $this, which is an instance of MailingList
```

[2]This is an example of **dependency injection**.

```
  public function sendWith(Mail $mailer) : MailingList
  {
    // setup the from address
    $mailer->from($this->from);

    // for each email send with the passed in $mailer
    foreach ($this->emails as $email) {
      $mailer->to($email)->send($this->subject, $this->message);
    }

    return $this;
  }
}
```

[View code on GitHub]

Before accepting the `$mailer` parameter, we add the type declaration/hint of `Mail`. Now, if the user of that class tries to pass in something that isn't an instance of `Mail`, PHP will throw an error.

```
// create a new mailing list
$mailinglist = new MailingList();

// ... code to set subject, from, add email addresses

// create a new mail object and pass in
$mail = new Mail();
$mailinglist->sendWith($mail);

// try to send with a Person
// won't work!
$mae = new Person("Mae", "Zimmerman");
$mailinglist->sendWith($mae); // TypeError!
```

[View code on GitHub]

11.5 Why?

Adding type declarations like this lets us find errors in our code quickly.

For example, say that we didn't add the `Mail` type declaration and then passed in an object that didn't have a `send` method. We'd get an error from PHP, but it would say the error was inside `MailingList` when we use the `send` method.

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

But that's not actually the issue, the problem is that we've passed in the *wrong* object type - one without a `send` method. So the error is actually when we pass the wrong type of thing to the `sendWith` method. By adding type declarations to the `sendWith` method we can make sure PHP finds the error at the correct point in our code.

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

11.6 Additional Resources

- Wikipedia: Encapsulation
- What is Encapsulation?: Uses Java for examples, but largely applicable in PHP
- Wikipedia: The Law of Demeter

Chapter 12

Polymorphism

You might look at the `MailingList` example from the previous chapter and think, "What's the point of passing in the `Mail` object?" And you'd be right. If we can only pass in a `Mail` object, then we may as well just create it in the `MailingList` class.

But 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? We can't have a type declaration for two different classes, but if we don't limit the type at all then you could accidentally pass in something that will break your code completely.

This is where the idea of **polymorphism** 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.

12.1 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.

For example, rather than creating a `Mailer` abstract class, we could instead create an interface:

```
interface MailerInterface
{
  public function to(string $address) : MailerInterface;
  public function from(string $address) : MailerInterface;
  public function send(string $sub, string $msg) : MailerInterface;
}
```

[View code on GitHub]

You can see that we list out all of the methods that a class that implements this interface *must* implement. We would use it as follows:

```
// we implement the MailerInterface
class Mail implements MailerInterface
{
  // setup properties
  private $to;
  private $from;
  private $characterSet = "utf8";

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

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

  public function send(string $sub, string $msg) : MailerInterface
  {
    // send using local mail server
  }
}
```

[View code on GitHub]

We would use it in `MailingList` in exactly the same way:

```
class MailingList
{
  // ...etc.

  // use the MailerInteface type-declaration to
  // allows any classes that implement MailerInterface
  public function sendWith(MailerInterface $mailer) : MailingList
  {
    // setup the from address
    $mailer->from($this->from);

    // for each email send with the passed in $mailer
    foreach ($this->emails as $email) {
      $mailer->to($email)->send($this->subject, $this->message);
    }

    return $this;
  }
}
```

[View code on GitHub]

Interfaces are also a type declaration. So, now we could only pass in classes that implement the `MailerInterface` interface.

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

```
class MailChimp implements
  MailerInterface,
  HttpInterface,
  EncryptionInterface,
  ThatOtherInterface
{
  // ...etc.
}
```

[View code on GitHub]

12.1.1 Message Passing

If you look at the `sendWith()` method you'll see that it only uses the `to()`, `from()`, and `send()` methods of the object that gets passed in. Therefore it's *guaranteed* that if the object passed in conforms to the `MailerInterface`, which defines all three of those methods, then it will work. That's not to say the implementation will necessarily work, but the `sendWith()` method has access to all the methods that it requires.

This is 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 also why we try to only use `private` properties: if a property is `public` then we need to know about how the internals of the class work.

12.2 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, say that we want to create a `Mail` and a `MailChimp` class. Both of these will be responsible for sending an email, so they will have some things in common, but their inner workings will be different. We could create a parent `Mailer` class:

```
class Mailer
{
  protected $to;
  protected $from;
  protected $characterSet = "utf8";

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

```
    return $this;
  }

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

[View code on GitHub]

We move all of the shared code into the `Mailer` class. We don't put the `send()` 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`:

```
// extends Mailer
// so it has the same properties and methods
class Mail extends Mailer
{
  public function send(string $subject, string $message) : Mailer
  {
    // send using local mail server
  }
}

// extends Mailer
// so it has the same properties and methods
class MailChimp extends Mailer
{
  public function send(string $subject, string $message) : Mailer
  {
    // send using mail chimp
  }
}
```

[View code on GitHub]

Now, in the `MailingList` class we can use `Mailer` as the type declaration. That means that depending on our mood,[1] we can send messages with either the local mail server or with MailChimp:

[1] Or possibly something more concrete

```
class MailingList
{
  // ...etc.

  // use the Mailer type-declaration to
  // allows all children of Mailer
  public function sendWith(Mailer $mailer) : MailingList
  {
    // setup the from address
    $mailer->from($this->from);

    // for each email send with the passed in $mailer
    foreach ($this->emails as $email) {
      $mailer->to($email)->send($this->subject, $this->message);
    }

    return $this;
  }
}

// elsewhere
$mailinglist = new MailingList();
$mailinglist->sendWith(new Mail());

// or maybe
$mailinglist->sendWith(new MailChimp());
```

[View code on GitHub]

12.2.1 Abstract Classes

But you can perhaps see a few issues with this. Firstly, you could create an instance of `Mailer` and pass that into `sendWith()`. This would cause an issue because the `Mailer` class doesn't have a `send()` method, so you'd get an error. Secondly, there's no guarantee that a child of `Mailer` has a `send()` 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 sendWith()
// tries to call it
$mailinglist->sendWith(new Mailer());
```

[View code on GitHub]

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 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 `send()` method:

```
// make the class abstract
// it cannot be instanced now, only extended
abstract class Mailer
{
  protected $to;
  protected $from;
  protected $characterSet = "utf8";

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

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

  // create an abstract method signature
  // all children of this class must implement the send method
  abstract public function send(string $subject, string $message) :
  ↪  Mailer;
}
```

[View code on GitHub]

12.2.2 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*[2] (i.e. excepts the same parameters), this will work:

```
class MailChimp extends Mailer
{
  // override the to method from Mailer
  public function to(string $address) : Mailer
  {
    // same as before
    $this->to = $address;

    // additional code
    $this->doWeirdAPIThing();
    return $this;
  }

  // ...etc.
}
```

[View code on GitHub]

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

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

  // not sure why you'd need to do this
```

[2] 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.

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

[View code on GitHub]

12.2.3 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:

```
class MailChimp extends Mailer
{
  public function to(string $address) : Mailer
  {
    // call the to method of the parent
    // passing in the address argument
    // has same method signature
    parent::to($address);

    // additional code
    $this->doWeirdAPIThing();
    return $this;
  }

  // ...etc.
}
```

[View code on GitHub]

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

12.3 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 the same bit of functionality to multiple classes. And it will almost always lead to much more complicated code.

That's not to say it isn't useful. It's totally fine to inherit 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.[3] But if it's code that you've written, it's always worthwhile thinking "Do I really need to use inheritance?"

Sandi Metz[4] 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.

12.4 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.

---

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

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

12.5 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.

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.
- **Instance**: An object is an instance of a specific class with its own set of properties
- **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.
- **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

∞

May 1, 2020