PHP PANDAS

THE PHP PROGRAMMING LANGUAGE FOR EVERYONE

DAYLE REES

PHP Pandas

The PHP Programming Language for Everyone.

Dayle Rees

This book is for sale at http://leanpub.com/php-pandas

This version was published on 2015-08-09

Tweet This Book!

Please help Dayle Rees by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm learning about PHP and Pandas AT THE SAME TIME. You can too! @ http://leanpub.com/php-pandas #PHPPandas @dayleree s

The suggested hashtag for this book is #PHPPandas.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#PHPPandas

Also By Dayle Rees

Laravel: Code Happy

Laravel: Code Happy (ES)

Laravel: Code Happy (JP)

Laravel: Code Bright

Code Happy (ITA)

Laravel: Code Bright (ES)

Laravel: Code Bright (SR)

Laravel: Code Bright (JP)

Laravel: Code Bright (IT)

Laravel: Code Bright (TR) Türkçe

Laravel: Code Bright (PT-BR)

Laravel: Code Bright (RU)

PHP Pandas (ES)

Contents

Acknowledgements

First of all I would like to thank my girlfriend Emma, for not only putting up with all my nerdy antics, but also for taking the amazing red panda shots for both books! Love you Emma!

Thanks to my parents, who have been supporting my interest in these math boxes for thirty years! Also thanks for buying a billion copies or so of the first book for family members!

I'd also like to thank all of my wonderful colleagues at JustPark for their continued support! You guys rock!

Thank you to everyone who bought my other books Code Happy and Code Bright, and all of the Laravel community. Without your support I'd never have had the confidence to continue writing.

Errata

This may be my third book and my writing may have improved since the last one, but I assure you that there will be many, many errors.

You can help support the title by sending an email with any errors you have found to me@daylerees.com[1] along with the section title.

Errors will be fixed as they are discovered. Fixes will be released within future updates to the book.

[1] mailto:me@daylerees.com

Feedback

Likewise, you can send any feedback you may have about the content of the book or otherwise. You can send an email to me@daylerees.com[2] or tweet to @daylerees.

I will endeavour to reply to all mail that I receive.

[2] mailto:me@daylerees.com

Translations

If you would like to translate PHP Pandas into your language, then please send an email to me@daylerees.com[3] with your intentions. I will offer a 50/50 split of the the profits from the translated copy, which will be the same price as the English copy.

Please note that the book is written in markdown format.

[3] mailto:me@daylerees.com

1. Introduction

Well hello there! Aren't you just the most handsome AND/OR beautiful reader on the planet! Well done you for buying PHP Pandas, and for taking the first step towards your career as a world-famous web developer.

Who am I? Well that's a simple question! My name's Dayle, and I'll be your author for this adventure. I've been writing books for beginners for a few years now, and have taken many other charming readers like yourself on adventures to learning new skills. We'll make new discoveries together, and all along the journey, rest assured that I'll be right by your side.

> Why do you write like a crazy person?

Excuse me? Oh this.. Well you see, this is the only way that I know how to write. If you're looking for a technical book full of science teacher stern-ness (Is that a word? I hope that's a word.) then I'm afraid you've come to the wrong place. I write my books for people. I like to think that we're buddies, sitting in the pub, talking about PHP over a pint of Special Bre… Fosters.

The truth is, the beginners that I've written for tend to like my writing style. They're not looking to gain a maths degree from this book, instead they're looking to learn a thing or two about PHP, and that, I can promise you!

Oh hey, you'll also notice that we're talking right now. You don't get that from other authors do you? You see, I have this magic power that will make you talk to me and ask your questions.

> Wait, how did you do tha…

That would be trade secret. Sorry, we can't share that just yet, but don't you feel glad that you get to be a part of this adventure, and not just an observer?

> I guess so… Sure I'll give it a go.

Excellent.

Well now's about the time where any other book would be telling you about PHP, it's whole history, its application, its author and about a million other things. Well we've already established that I'm not the most traditional author, and I'm not fond

of such chapters. You've bought this book to learn about PHP, so you've already built up a little curiosity about the language. I think this is all you're going to need.

PHP is a programming language that powers most of the sites out there on the big, wide ol' interwebs. It was originally written by a guy called Rasmus Lerdorf, who can often be seen smiling in pretty much any image you find of him on Google. Now, Rasmus is a great guy, and in my own way I thank him each and every day for this language that has given me a trade, but I think that's all you need to know about him. Other PHP books would probably be telling you his favourite cereal about now, but instead, how about we actually jump in and start learning?

This book is for ***absolute*** beginners. This means that if you've never tried programming before in your life then you're in luck my friend! If you've already tried programming, then you'll do just fine. If you're a PHP expert, then now's a time for a refresh of your skills, and maybe you'll pick up a few tips and tricks along the way.

I've been using my girlfriend (no dev experience), my non-technical colleagues, and random people on the street, forcing my book upon them as guinea pigs to see how it goes down with folks that have no prior knowledge of PHP. My little guinea pigs did exceedingly well, so now it's your turn, squeak squeak!

My goal for this book, is for it to become the most fun, factual, and fantastic PHP book that's on the market. I want it to be **the** book that gets recommended when someone is about to become a PHP developer. I've worked really hard to make it accessible to everyone, so if you enjoy this adventure then please tweet about it, blog about it, buy copies for your friends and family, or just print it out and slap people in the face with it as you pass them on the street.

This book is a syntax book for PHP. It's not going to teach you how to make websites (I'm working on the title in the series for this). Instead, it's the first step that will build your foundation knowledge of the language so that when you come to build your first website, you're gonna be @% £^ hot, baby!

If you read the book and you feel like something is missing, that a certain chapter is confusing, or there's anything else bothering you, then please send me an email to `me@daylerees.com` to let me know! I'm incredibly responsive (thanks to all my media queries... haha... programmer joke), and I want this book to be perfect for everyone.

If you read the book and you didn't find anything wrong, well... send me an email to tell me you enjoyed it! I'd love to hear from you.

Right then, let's not waste any more time. You've got some skills to learn! Flip the page, imagine the Jurassic Park theme when they open the gates, and prepare to enter the world of development!

2. Installation

Before we begin working with PHP, we must first install it. You see, PHP is an application like any other. It needs to be installed on our system before it can process PHP code.

The method of installation varies greatly depending on the operating system that we are using. For that reason, I've provided three different guides for installing PHP. The first section will explain how to install PHP on a Linux distribution, namely Ubuntu due to its popularity. The second section will explain how to install PHP on an Apple Mac OSX system. Finally, the third section will explain how to install PHP on the Windows operating system.

We'll only be installing the console version of PHP. We won't be setting up a web server just yet. We'll get to that in a later title. The console version of PHP is all we need to get started with our learning process.

Remember, you only need to read the appropriate section for your computer. Once you have PHP installed, go ahead and skip to the next chapter of the book.

Linux

The best way to install PHP on a unix-based Linux distribution is to use a package manager. The package manager available depends greatly upon the distribution of Linux that you have chosen. I've decided to provide instructions for installing PHP on Ubuntu, one of the more popular distributions of Linux.

Ubuntu uses the `apt` package manager to install its packages. To install the console version of PHP we need to install the `php5-cli` package. Let's do this now. First open a new terminal. You'll need to type the following instruction.

```
1 $ sudo apt-get install php5-cli
```

You don't need to type the dollar sign, that's just the terminal prompt to show you that we're typing it into the console. Once you hit enter, apt will retrieve the PHP application package, and install it for you.

That's it! You're done. well you should be. Let's check, shall we? Simply type...

```
$ php -v
```

This command is used to show the current version of PHP installed. You should see something similar to the following.

```
PHP 5.5.13 (cli) (built: Jun  5 2014 19:13:23)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.5.0, Copyright (c) 1998-2014 Zend Technologies
```

Yours won't be exactly the same, after all, we're all different, aren't we? In the example above, the PHP version is `5.5.13`. Hopefully your PHP version number should be `5.4.0` or greater.

If your version isn't right then you'll have to consult the documentation for your Linux distribution to find out how to install the appropriate version.

Go ahead and skip to the next chapter, you're done!

Mac OSX

On the Macintosh operating system, PHP comes pre-installed. Go ahead, open up the Terminal application and type the following to find the version of PHP you're using.

```
$ php -v
```

Don't type the dollar sign, that's the terminal prompt! You should see something similar to the following, but not exactly the same.

```
PHP 5.4.24 (cli) (built: Jan 19 2014 21:32:15)
Copyright (c) 1997-2013 The PHP Group
Zend Engine v2.4.0, Copyright (c) 1998-2013 Zend Technologies
```

The PHP version in the example above is `5.4.24`. As long as your version of PHP is greater than `5.4` then you're fine, and can move to the next chapter.

If yours isn't, that's okay. We can use a third party package manager for OSX to install a newer version of PHP.

We're going to use a package manager called 'Homebrew' or just 'Brew' for short. To install Homebrew, follow the instructions found on the following site:

brew.sh[1]

I don't want to copy the instructions here, as they often change between different releases. Once you have Homebrew installed, it's time to install a newer version of PHP. I recommend installing version `5.5`. You can do this using the following command.

```
$ brew install php55
```

Next you need to add the location for this version of PHP to your system PATH variable. Don't worry, just type the following.

```
$ PATH=~/usr/local/Cellar/php55/5.5.13/bin:$PATH
```

You may need to update the version number to match the version of PHP that Homebrew has installed on your system. Now let's have another go at checking the version of PHP.

```
$ php -v
```

Hopefully, this time you'll have a version greater than PHP `5.4`. Go ahead and skip to the next chapter.

Windows

Installing PHP on Windows is a little more difficult, at least for me it is. I've tested the instructions below on my Windows 10 machine, but if you have any difficulty replicating these steps, let me know and I'll find someone who's more Windows-savvy to rewrite this section.

First, head over to:

http://windows.php.net/download

Here you'll want to download the latest PHP 5.4 and above zip archive. Once the archive has been downloaded, you'll want to extract it to a sensible location. I chose to extract mine here:

```
C:\Users\Dayle\PHP
```

You're going to need a command prompt to execute the scripts that we write in this book. So here's a great way of running a command prompt on Windows.

Right click on your desktop, or any folder and choose 'Create Shortcut'. In the target box enter:

[1] http://brew.sh/

```
cmd.exe
```

Click next, and name your shortcut “PHP”.

Finally, you’ll want to right click your shortcut and click ‘Properties’. On the ‘Shortcut’ tab, change the ‘Start In’ field to match the location where you extracted the PHP archive. Click ‘OK; when done.

Double click on your PHP shortcut, and you should be greeted with a command prompt. Type…

```
php -v
```

..and you should be greeted with the PHP version information. Confirm that the version is greater than or equal to PHP 5.4, and then move to the next chapter.

Once again, sorry for the roughness of this subchapter. I’ve not used Windows as a development machine for a number of years now. If anyone has a better way of running PHP on Windows, kindly email your instructions to receive your 5 minutes of fame within this chapter!

3. Finding Answers

I know. That's a kinda fluffy title isn't it? You're going to have to trust me when I say that this is important stuff. This chapter is about your confidence as an up and coming developer. Learning is hard, but don't worry; I'm going to help you through this.

Developers are robots.

Why did you decide to pursue development? No, wait! Let me guess. You saw a rockstar PHP developer swagger out of a Limousine into one of New York's hottest night spots, order 5 bottles of Cristal and spend the evening chilling with Jay-Z and the ghost of Tupac.

It's true, a developers life is a glamorous one. I have to write these chapters within my 5 hours of sobriety a day. You've probably seen a developer writing some code and thought…

> Oh man, that dev must be a robot. They know all of those code words and functions and how they all work.

When people without development experience approach developers they assume they are genius types with mathematics honour degrees. Perhaps this is true for some developers, but it's certainly not true for me. I'd like to think that other developers would agree.

The truth is, we're not perfect. We're not even close to perfect. If you think that developers know all of these PHP functions and snippets from memory then you are fooling yourself into thinking that you will never have the capacity to keep up.

It's just not true. We don't memorise everything. In fact, a majority of the code that we use day to day is from reference. We are google warriors. There are functions in PHP that do the most simple things to lines of text, and I look at the PHP documentation almost every week to find the order of the parameters that I pass into them.

When I'm completely stuck I'll try using Google to see if another developer has found a similar challenge. Often I'll find a suitable solution that another developer has discovered, or enough information to point me to a solution. Of course this

works both ways, I'll try and give my solutions back to the community. I'll post answers on Stack Overflow and I'll contribute to forums or discussions. It's important to give back to the community.

So you see, we aren't robots. We don't know everything about the language, and we don't have a solution to every problem. However, we are fantastic researchers. We are opportunists. We are resourceful problem solvers. We are developers.

The art of Googling

When people tell you to Google something, it's easy to take it as an insult. Or perhaps sarcasm? It's not. Google is our homepage for a good reason. Let's learn how we can find answers to common development issues.

We're writing a program, and somewhere we need to reverse a sentence so that 'Pandas rule!' becomes '!elur sadnaP'. We have no idea how to approach this task. We're just getting started with PHP.

We know that in PHP a sequence of text is called a 'string'. We know this because we didn't give up on this crazy book with the Panda examples, and we discovered this fact in a later chapter. Right?

So we know what we want to do. We would like to reverse a string. Let's construct a search query for google.

```
1 reverse string
```

Nope, wait! The problem here is that there are thousands of programming languages. Seriously, computers have been around for a while!

If we search for 'reverse string' then we're going to get answers for C++, ASP.NET, Erlang, you name it. Our focus is on PHP. We don't care about these other languages. We'll have time to play with them later when we become PHP masterminds. Let's fix this problem by adding the language to the search query.

```
1 php reverse string
```

Perfect. Let's take a look at the results that we get back from our Google search. This might be a good time to mention that I don't work for Google, and I'm not working for commission. Feel free to use Bing if you prefer it, but you might end up buying a used horse trailer rather than finding a string reverse function. So where are those results?

Reverse a string - PHP

http://www.php.net/manual/en/function.strrev.php[1]

Reverse a string with php - Stack Overflow

http://stackoverflow.com/questions/11100634/reverse-a-string-with-php[2]

By asking the right question, we receive some useful resources in return. The PHP Manual (sometimes known as the PHP API docs) and Stack Overflow are two of the most useful problem solving resources for PHP available on the internet. I'm not saying they always have the right answer. There are other great sites too, but I'm sure you'll see a pattern in how often your searches result in browsing pages on these two sites.

Right now we're looking for some sort of tool to reverse a string. We're not really looking to solve an abstract problem, we know exactly what we want.

Go ahead and click that first link, we'll be greeted with the lovely PHP manual page for a function called `strrev()`. You don't need to know what a function is yet. Don't worry if this is over your head.

Once you're up to speed with functions you'll see that this PHP manual page offers all that we need to know about using the `strrev()` function, and examples of how to use it.

You see by asking the right questions we received all the help we needed to continue with our work. We had no prior knowledge of the `strrev()` function, but instead we knew the problem that we had to solve. That was enough to lead us to the solution. It doesn't matter if we have to go back to this page later.

Perhaps we don't use the function frequently enough to need to remember its usage pattern. Although, you'll find that if you begin to use the function more and more, and that you frequent the manual page, then before to long you won't need guidance for that problem. You'll instantly think 'Hey I should use that `strrev()` function that I use all the time, and I know exactly how it works!'. It will become muscle memory, and will be part of your toolset.

So the lesson that I'm hoping you have learned from this chapter is that you shouldn't panic. You don't need to remember everything, and it's perfectly natural to ask for help. In fact, it's human to ask for help, and it's human to learn from your experiences.

Congratulations! You're a human, not a robot.

[1] www.php.net/manual/en/function.strrev.php
[2] http://stackoverflow.com/questions/11100634/reverse-a-string-with-php

4. Files

Here's a shocker for you. PHP code is kept in files. I'm sorry, but it's true! You are going to be working with lots and lots of files. Well actually, sometimes one file, but later you'll be working with many, many files!

Now that we have that shocking truth out of the way, isn't it time that you learned how to create a PHP file.

> Dayle, I understand the fundamentals of a computer file system.

Well done buddy! Good for you, but that's not where we're going with this. You see, most PHP files have something in common. I'm talking about the PHP script tag.

Take a close look at this little fella.

Example 01: PHP tag.

```
1 <?php
```

Beautiful isn't she? What a glossy coat. An absolutely fantastic specimen.

> I.. erm..

What? You don't feel the same about her? Trust me, after many years of development in PHP you will find her quite beautiful. You'll see her when you close your eyes to go to sleep at night. She's your best friend. She lets you use PHP.

I always prefer to lead with a practical example, so let's try something together. Create a new file, and call it `test.php`. PHP files usually have the `.php` extension. In honesty, we can execute them without it, but you should stick to it because if you don't then the bigger developers will laugh at you, steal your lunch and make you cry. Just kidding... developers are a friendly bunch, but you really should use the `.php` extension.

First of all, let's write the words...

Example 02: Some text.

```
Pandas rule!
```

...into the file, and save it.

Great, now let's execute the file. We can use this by calling the `php` application at the command line or unix shell, and passing the file name as a parameter. For example, on my Mac I'll be typing the following.

Example 03: Executing a PHP file.

```
php test.php
```

You'll see the words `Pandas rule!` outputted to the screen. This is because everything outside of our beautiful PHP tags are outputted when the application is executed. Let's try something else. We are going to use our first PHP tag.

Let's edit the file so that it reads as follows.

Example 04: PHP segment.

```
<?php

// Pandas are awesome!

?>
Pandas rule!
```

Let's execute the file again. What's the output that we get?

Example 05: Output.

```
Pandas rule!
```

> Hey wait! Where's the rest?

Well spotted, my soon-to-be developer! There's a section of our file missing. This is because everything between our PHP tags is treated as PHP code, and is processed accordingly.

So what are the PHP tags? Well you've met the PHP opening tag already. Do you remember our beautiful friend `<?php`. The `<?php` tag marks the beginning of our PHP code. So when does it end? Well that's where the `?>` tag comes into play.

Now that you know how the PHP tags work, it's easy for us to spot the PHP code in this file. It's the following line.

Example 06: Comment.

```
1 // Pandas are awesome!
```

So what does this line do? Absolutely nothing. It's known as a comment. It helps developers to document their own code. Don't worry. We'll learn more about comments later.

Well that was a nice short chapter, wasn't it? Now it's time for some good news. In the next chapter you'll be writing your first ***real*** lines of PHP code.

Excited? Then why wait! Flip that page.

5. Basic Arithmetic

Now I'm sure you've heard that programming is all math. Right? Well it's time for math. Let's get started.

$$\left|\sum_{i=1}^{n} a_i b_i\right| \le \left(\sum_{i=1}^{n} a_i^2\right)^{1/2} \left(\sum_{i=1}^{n} b_i^2\right)^{1/2}$$

Now solve for X.

Just kidding. Actually, there's no X in that equation. In fact, it's not even an equation, so that was a terrible joke. Hey! They can't all be side-splitters. The truth is, I have no idea what that mess does either. We aren't all math gurus.

Statements

Let's try something that's a little bit closer to my own level of mathematics. You know how to make a PHP file, and you know how to open and close PHP tags. So let's jump straight into a PHP file. We'll call it `math.php`. Here's the content.

Example 01: Addition.

```
1  <?php
2
3  3 + 3;
4
5  ?>
```

Actually, hold on a second. We aren't going to output anything after our PHP code. Why bother with the closing tag? The truth is, most PHP developers omit this tag if there's no content that will follow our PHP code. Let's do that.

Example 02: We don't need a closing tag.

```
1  <?php
2
3  3 + 3;
```

Much better!

Right, just in case your math skills aren't quite as sharp as my own, let me help you out a little. When you add three to three you get six. Okay, now you're ready.

The line `3 + 3;` contains a statement. It's a line of PHP code that will be evaluated by PHP. They normally end with a semi-colon. It looks like this: `;`. You're gonna forget about them all the time at first, but don't worry, soon you'll even be ending your sentences with them;

Given that you now understand basic addition, what do you think will be the output when we execute this file?

> Seven point five.

Well, special reader, let's see if you're right. Go ahead and run `php math.php` to see what happens.

Example 03: Output.

```
1  [nothing here]
```

Woah! Absolutely nothing. This language is stupid. Let's give up. Okay, I'm kidding again. I have a cheesy sense of humour, don't worry, you'll get used to it.

Why didn't we get any output? Well, it's because we didn't tell PHP to output anything. PHP is obedient. Let's go ahead and tell it to give us the answer. We'll use `echo`. It's a PHP language construct that will allow us to see the result of a statement.

Let's alter our statement to include `echo`.

Example 04: The echo statement.

```
1  <?php
2
3  echo 3 + 3;
```

There we go. We place `echo` before the statement that we want to see the result of. Let's try running our application again! Here we go...

Example 05: Output.

```
6
```

Woohoo! Six! **NOT SEVEN POINT FIVE**! Now that's what I'm talking about. We get to see the result of our first statement evaluation with PHP. That's exciting stuff, right?

> I could have done that on a calculator.

I know, I know. It's not exactly rocket science. Rocket science will be covered in a later chapt… Wait, I've already told that joke in another book. I need to get some new material.

Arithmetic Operators

I know that our `3 + 3` example is simple code, but we'll soon get to bigger and better things. Did you know that there are more mathematical operators? I'm sure some of these ring a bell.

```
+ Addition
- Subtraction
* Multiplication
/ Division
% Modulus
```

Now, I'm sure you'll have seen seen some of these operators before. I know that multiplication and division look a little different to the signs that you may have learned about in school. This is common to most programming languages, and you'll find that the division sign is definitely easier to type on a keyboard. Don't let them worry you, before too long you'll be completely used to them.

If you've not used the 'Modulus' operator before, then it's simple to explain. It can be used to calculate the remainder of a division. For example, the operation '3 % 2' would result in the figure '1'. It's commonly used to determine whether a number is odd or even by dividing by two.

Now let's give PHP something hard to think about, shall we?

Example 06: Harder math.

```
<?php

echo 4 + 3 * 2 / 1;
```

So, what's the result? Well it can be difficult to calculate in our heads because we don't know which order to process the pairs of calculations. Should we add 3 to 2 first? Or maybe divide 2 by 1 first. Hmmm. Tricky!

Of course, in mathematics we learn to use rounded brackets to separate the concerns of an equation. We can do the same with PHP. Let's give it a go.

Example 07: Brackets for separation of concerns.

```
<?php

echo (4 + 3) * (2 / 1);
```

Now we can be sure that `4 + 3` and `2 / 1` are evaluated first, and the resulting values are multiplied. Great, we run our script and get the result...

Example 08: Output.

```
14
```

Awesome, but isn't that cheating? What would we get without the brackets? Let's take them out again.

Example 09: Without brackets.

```
<?php

echo 4 + 3 * 2 / 1;
```

So what's the result? Let's run our script.

Example 10: Output.

```
10
```

That's a totally different figure. Why is that? Well, it's because PHP isn't handling our operators in the same order. Let's take a little time to learn about the order of our operators.

Here's how PHP handles the order of operators.

```
* Multiplication
/ Division
% Modulus
+ Addition
- Subtraction
```

The operator with the highest priority can be found at the top of the table. So this means when PHP examines `4 + 3 * 2 / 1` it will first calculate `3 * 2 = 6`, then `6 / 1 = 6` and finally `4 + 6` to give us the answer `10`.

When I'm writing mathematical lines of code, I like to use brackets to avoid any confusion. I also find that it helps to clarify the intent of the line, causing it to be more readable.

Procedure

PHP code is parsed procedurally. This means that it is read and executed on a statement by statement basis. While it's possible to put more than one statement on a line, this is uncommon amongst PHP developers. This means that we can also approach the code line by line. We can see this in action by adding more statements to our PHP file. Let's try the following.

Example 11: Multiple statements.

```
<?php

echo 2 + 2;
echo 3 + 3;
echo 4 + 4;
echo 5 + 5;
```

Now, let's execute the file...

Example 12: Output.

```
46810
```

> FORTY SIX THOUSAND GIGAWATTS!?

Calm down reader! We only told PHP to output the results, not to place spaces or newlines into the output. This means that PHP has calculated the values correctly. If we space out the result that PHP has given us like so...

Example 13: Output with added clarity.

```
4 6 8 10
```

...then we see that the calculations are in fact correct. It's just that PHP is very obedient and has outputted the values directly after one another.

I've mentioned many times before that PHP is a flexible and lenient language. Let's put that to the test, shall we? Up until now, our statements have a single space between each 'word' (or number). Let's add some extra spaces in an inconsistent format to see what happens. Here's our modified code.

Example 14: White space.

```
<?php

echo    2          +   2;
echo                  3   +3;
echo 4+4;
echo    5+     5        ;
```

While it doesn't look very pretty, if you were to execute the code you'd find that it will work perfectly. PHP doesn't care about the amount of white space between the words within its code. It just deals with it. (Insert dog with shades...)

You'll notice that some of the arithmetic operations, for example `4+4`, don't require a space at all. While this is true, it isn't consistent for all syntax variations. For example, consider the following snippet.

Example 15: No whitespace after echo.

```
<?php

echo5 + 5;
```

If you attempt to execute this script, you'll find that PHP will throw a notice 'Use of undefined constant echo5 - assumed 'echo5'. This is because it doesn't know what the word `echo5` is telling it to do. For this reason, it's always best to place at least one space between your words.

As for statements, if we were masochistic we could choose to put all of the statements on a single line. Here's an example.

Example 16: Multiple statements, single line.

```
1  <?php echo 2 + 2; echo 3 + 3; echo 4 + 4; echo 5 + 5;
```

This is perfectly valid PHP, but you won't find many developers doing it. Having a single statement on each line makes it much easier to read and understand a source file. It also causes problems for version control systems!

We've seen that PHP doesn't care if you use multiple spaces in its source code, but it also considers a newline a whitespace character. This means that the following snippet is completely legal.

Example 17: One statement, multiple lines.

```
1  <?php
2
3  echo
4  2
5  +
6  2
7  ;
```

Don't believe me? Go ahead and try it! While the code functions as intended, it's not exactly the most readable piece of code. If I catch you writing code like this then you're due for a spanking!

There is one practical use to breaking a line, however. If the line is exceedingly long then it also becomes a readability issue. We can resolve this issue by applying a new line at an appropriate reading length. Many developers also apply four spaces (or your current tab setting) to the next line to indicate that it is a continuation. This is similar to how formal works of text use an indented sentence to indicate a new paragraph.

Here's an example of a line break for readability purposes.

Example 18: Clean line-breaking.

```
1  <?php
2
3  echo (3 * 5) / (7 / 12) * (7 * 6) + (7 % 3)
4          + (6 + 7) * (12 / 3);
```

That's some serious math, but hopefully you will find it much easier to read.

It's also worth noting that you can also place empty lines within your code to add clarity. Here's an example.

Example 19: Extra line breaks for clarity.

```
1  <?php
2
3  echo 3 + 2;
4
5  echo 7 * 7;
6
7  echo 5;
```

So you see, PHP can be extremely flexible, but don't forget to add that end of line semi colon because it will never forgive you.

EVER;

6. Variables & Assignment

Now we're getting to the meat and potatoes! Variables are an extremely useful and well abused part of the developers toolkit. Let's get started, shall we?

Tiny Boxes

I'd like you to think of variables as tiny little boxes that we keep things in. Variables are words that start with a dollar `$` sign. Let's take a look at an example.

Example 01: Basic assignment.

```php
<?php

$three = 3;
```

If you think of the variable `$three` as a little box, then we've put the value `3` into it. That's what the equals sign does. In math we use the equals sign to indicate the result of an equation, however, in PHP it's an entirely different story.

In PHP the equals `=` sign is known as the assignment operator. It's used to ***set*** something. We are telling PHP to ***set*** the variable `$three` to the number `3`.

If you execute the script we have created above, you'll find that PHP doesn't output anything at all. This is because assignment is purely assignment. We aren't telling PHP to output anything. However, now that we have set the variable `$three` to the value `3`, we can use the `echo` construct on it.

Example 02: Echoing a value.

```php
<?php

// Set our variable to the value three.
$three = 3;

// Output the value of our variable.
echo $three;
```

First we set our variable, and then we use the `echo` construct to output the value that it's holding. If we execute our code, then we receive `3` as output.

This is great because it means we can give nicknames to things. You know, just like those mean kids at school. For example, the number '3.14159265359' is a very beautiful number to lovers of circles, but it's awfully hard to remember, isn't it? Let's give it a nickname. We'll call it Pete. No wait, I have a better idea.

Example 03: An appropriate variable name.

```
1  <?php
2
3  $pi = 3.14159265359;
```

There, now we have created a new variable called `$pi` that holds the value `3.14159265359`. This means that we can use the variable anywhere in our code to perform calculations. Here are some examples.

Example 04: Using variables in statements.

```
1  <?php
2
3  // Assign pi to a variable.
4  $pi = 3.14159265359;
5
6  // Perform circumference calculations.
7  echo $pi * 5;
8  echo $pi * 3;
```

After setting `$pi`, we can use it in other statements to perform calculations.

We can declare and assign as many variables as we like, but there are a number of rules that we need to follow when choosing names. Variable names can contain numbers, letters and underscores. However, they ***must*** start with either a letter or underscore, never a number! They are case sensitive, which means that `$panda` is different to `$pAnda`. Here are a few examples.

Example 05: Naming variables.

```
<?php

$panda = 1;      // Legal
$Panda = 1;      // Legal
$_panda = 1;     // Legal
$pan_da = 1;     // Legal
$pan_d4 = 1;     // Legal
$pan-da = 1;     // Illegal
$4panda = 1;     // Illegal
```

While variable names can contain underscores and start with capitals, it's a common practice to use a naming format known as `camelCasing`. Don't worry, it doesn't require a camel.

camelCased names start with a lowercase character. Variables that are to be named with multiple words will have the first character of subsequent words capitalized. Here are some examples.

Example 06: camelCased variable names.

```
<?php

$earthWormJim = 1;
$powerRangers = 1;
$spongeBobSquarePants = 1;
```

Do you remember how our statements return a value? Well our assignments are also statements. Can you guess what this means? That's right, they also return a value. We can prove this by using our good ol' friend the `echo` construct.

Example 07: Statements return a value.

```
<?php

echo $panda = 1337;
```

We receive the number `1337` as the output. This is because the assignment of the `$panda` variable is performed before it is outputted. This process allows us to use a clever trick. It's not something you're going to use very often, but I think it's a pretty cool trick to know. Go ahead and take a look at this example.

Example 08: Multiple assignment.

```
<?php

$firstPanda = $secondPanda = $thirdPanda = 1337;
```

The snippet above might look a little crazy, but it makes more sense if you read it from right to left. The `$thirdPanda` is assigned the value `1337`, next the `$secondPanda` is assigned the value of `$thirdPanda`, and finally the `$firstPanda` is set to the value of the `$secondPanda`. This means that all variables are set to the final value. Neat, right?

Just my type

Until now we have been working with numbers. It would be boring if those were the only types of values that we can use, right? I think it's about time we examined the other possibilities. Here are some of the common values used within PHP applications.

- integer
- float
- boolean
- string
- null
- array

There are a few more, but let's not complicate matters right away. We need to learn little by little. You don't want knowledge overload!

Let's take a look at these types one by one. First we have integers. These are whole numbers, we've been using these in our previous examples.

Example 09: Integers.

```
<?php

$panda = 2;
$redPanda = -23;
```

Floats are floating point numbers. They have decimal points, and thus contain fractions. They can be used in a similar fashion to integers. In fact, we've used one already. Do you remember our friend `$pi`? That was a float. Let's move on to something new shall we?

Example 10: Floats.

```php
<?php

$panda = 2.34;
$redPanda = -23.43;
```

Booleans are binary data types. No don't panic! We aren't going to do any binary arithmetic. It's just a way of expressing that they can be one of two values. A boolean can either be `true` or `false`. Later on, we'll take a look at how boolean values can be used to change the flow of our application.

Example 11: Booleans.

```php
<?php

$panda = false;
$redPanda = true;
```

Next up we have the 'string' value. Strings are used to store a word, a character, or a sequence of text. Strings are special, so I've decided to dedicate a short chapter to them. We'll come back to this!

Example 12: Strings.

```php
<?php

$panda = 'Normal Panda';
$redPanda = "Red Panda";
```

Null is a special value. It is nothing. Nil. Zero. Well actually it's not zero. Zero is numeric, and we can use integer for that. Nulls are exactly nothing. Null is the value that a variable has before assignment has been performed. It's a really useful value, and you're going to see a lot of it in the future.

Example 13: Null values.

```php
<?php

$noPanda = null;
```

Arrays are another special type of value. In fact, this is my favourite one of all. So much that I've decided to dedicate a full chapter to them. For now, all you need to know is that it's a value that holds a collection of other values. Woah! Inception stuff, right?

Example 14: Arrays.

```php
<?php

$countThePandas = [1, 2, 3];
$morePandas = array(5, 6, 7, 8);
```

Advanced Assignment

In a previous chapter we discovered the operators that we can use on variables, and we've mastered the assignment operator. So what happens when we put them both together? Will it create a new black hole and consume the entire universe? I'm feeling a little daring, shall we find out?

Example 15: Assignment with addition.

```php
<?php

// Set a value.
$panda = 3;

// Attempt to create a black hole.
$panda += 1;

// Universe withstanding, dump the value.
var_dump($panda);
```

First we set a variable to the integer value of three. Next, we've plopped the addition operator onto the front of the assignment operator and supplied another integer value of one.

We can use the function `var_dump()` (more on functions later!) to interrogate not only the value held within a variable, but also its type!

What did we get back from the dump?

Example 16: Output.

```
int(4)
```

Awesome! The universe is saved. It looks like we have a four? Well, I suppose that makes sense. We know that `$a + $b` returns a value without setting it, and we know

that the assignment operator is used to set the value of variables. This does both. We're telling PHP to set the value of `$panda` to its current value plus one.

You can use this syntax with any of the operators that we've discovered so far. There's only one catch. Don't place the operator on the other side of the equals sign. Trust me, I tried it. A portal opened to a dark underworld, half dinosaur, half human creatures broke through and began to terrorise Cardiff. Only with the aid of a home made flamethrower (powered by PHP) was I able to fend off the vicious creatures. I'd hate to see it happen to you. Please be careful!

Next up, we have the incremental operator. Actually, let's not forget the decremental operator too. She tends to get a little less attention. In fact, let's showcase her abilities.

I prefer to lead with an example. Consider the following snippet.

Example 17: – After.

```
<?php

// Set a value.
$panda = 3;

// Decrease the value.
$panda--;

// Dump the value.
var_dump($panda);
```

There in the middle, do you see her? The beautiful decremental operator. We simply place two minus signs after the variable. What does it do? Well, here's the result of the code snippet.

Example 18: Output.

```
int(2)
```

As we can see, the value of `$panda` has been decreased by one. It's a quick shortcut to decreasing a value. Likewise, using a `++` can be used to increase a value. Those are the only two operators that work, though. Don't you be cheeky and try use the multiplication operator. It just won't work as you expect it to!

I wonder what will happen if we put the operator before the value? Let's have a go, shall we?

Example 19: – Before.

```php
<?php

// Set a value.
$panda = 3;

// Decrease the value.
--$panda;

// Dump the value.
var_dump($panda);
```

What's the answer? Aren't you excited?

Example 20: Output.

```
int(2)
```

Oh, it's the same. Well that was rather boring, wasn't it? Actually, I know a little secret. It's not the same. Sure, the value we received back looks identical, but my example doesn't do it credit.

Let's craft a different example. We'll show the state of a value **before** the operator is used. We'll examine the result of the statement **when** the operation is used, and finally, we'll examine the value **after** the operator is used. We don't expect the after-value to be any different.

Example 21: The stages of –.

```php
<?php

// Set a value.
$panda = 3;

// Dump BEFORE.
var_dump($panda);

// Dump DURING.
var_dump(--$panda);

// Dump AFTER.
var_dump($panda);
```

Let's execute the code. What are the three values we receive?

Example 22: Output.

```
int(3)
int(2)
int(2)
```

The first value is three. We must have expected that, I mean, all we did was set it, right? The result of the statement using the decremental operator is equal to two. The resulting value is also two. That means that the value is decreased on the second line.

Let's move the operator to the other side of the value, shall we? Like this:

Example 23: The stages of – part two.

```
<?php

// Set a value.
$panda = 3;

// Dump BEFORE.
var_dump($panda);

// Dump DURING.
var_dump($panda--);

// Dump AFTER.
var_dump($panda);
```

Look very closely to spot the difference. Let's take another look at the result.

Example 24: Output.

```
int(3)
int(3)
int(2)
```

Hey!? That middle value is different! Why hasn't it been decreased? Well, by swapping the operator, we've told PHP to decrease the value **AFTER** the current line. The result of the operation line is the same value as it was initially.

Let me summarise.

```
$value-- - Change value *after* current line.
--$value - Change the value on the current line.
```

Why is this useful? Well, here's a use for you. I'm sure if you're creative you will find more. Using the operator that changes **after** the current line, we can set another variable to its value, and decrease the original value on the same line. Like this:

Example 25: Assign and increase.

```
<?php

// Set a value.
$panda = 3;

// Assign, and then increase.
$pandaFriend = $panda++;
```

What we've done here, is saved a line. It's a bit of a shortcut. Here's how it would look if not for the incremental operator.

Example 26: Incrementing explained.

```
<?php

// Set.
$panda = 3;

// Assign.
$pandaFriend = $panda;

// Increase.
$panda = $panda + 1;
```

In a later chapter about loops you'll find another use for this operator. In the next chapter we'll be taking a closer look at strings.

7. Strings

We've encountered strings already, haven't we? Well, I think they are a really interesting data type. That's why I've chosen to dedicate a short chapter to them.

Interpolation

Strings are the name that programmers have given to sequences of text held within the variables of our applications. Let's assign a string to a variable for some catch-up.

Example 01: Different quotes for different folks.

```php
<?php

// String with single quotes.
$panda = 'Pandas rule!';

// String with double quotes.
$panda = "Pandas rule!";

// Output.
echo $panda;
```

Okay, so I know the second line is pointless. I left it in there to remind you of the two different types of quotes that can be used to enclose string values.

Right then, if we have two options, then what's the difference? It can't just be for clarity, right? I mean, they both look equally clean to me.

Well, there is actually a difference. String values that are enclosed in double quotes are a little smarter. They have a trick that their single-quoted brethren just aren't able to do. It's called String interpolation.

That's a horrible term, isn't it? It's very confusing. It simply means that we can embed values within a string. Let's take a look at this in action.

Example 02: String interpolation.

```php
<?php

// Set a string.
$value = 'pandas';

// Single quotes.
$first = 'We love $value!';

// Double quotes.
$second = "We love $value!";

// Output.
var_dump($first);
var_dump($second);
```

We'll try inserting a variable by name into both types of strings. Let's take a look at the result of dumping out both types of strings.

Example 03: Output.

```
string(15) "We love $value!"
string(15) "We love pandas!"
```

As you can see, with single quotes the name of the variable is presented as if it were simply part of the string. However, within the double quoted string, our variable name has been replaced with the value of the variable.

Woah! That's super useful. Isn't it?

I want to offer you a little advice. It's best to wrap interpolated values within `{` curly braces `}`. It makes the interpolation much cleaner, and you'll find it will work better when you start using arrays.

Here's an example.

Example 04: Neater string interpolation.

```php
<?php

// Set a string.
$value = 'pandas';

// Insert value.
$result = "We love {$value}!";

// Output.
var_dump($result);
```

Much cleaner, isn't it? Stick with the curly braces. Trust me, you will grow to love them.

Concatenation

Concatenation is the process of joining two strings end to end. Like creating a bead necklace, or the human centipede... oh my. Let's take a look at an example of this process in action.

Example 05: String concatenation.

```php
<?php

// First value.
$first = 'Pandas are';

// Second value.
$second = ' awesome!';

// Concatenate.
var_dump($first . $second);
```

First we create two string values, and then we output the result of concatenation. In PHP we use the period `.` character to perform concatenation. Let's take a look at the result, shall we?

Example 06: Output.

```
Pandas are awesome!
```

Great! Our strings have been stuck together. We can concatenate as many values as we like. Here's an example.

Example 07: More values!.

```php
<?php

// First value.
$first = 'Pandas';

// Second value.
$second = ' are';

// Third value.
$third = ' completely';

// Fourth value.
$fourth = ' awesome!';

// Concatenate.
var_dump($first . $second . $third . $fourth);
```

We can also concatenate different types of variables together; PHP will simply treat them as strings. To prove this, let's concatenate a string and a float value together.

Example 08: Concatenating different types.

```php
<?php

// First value.
$first = 'Value of: ';

// Second value.
$second = 27.325;

// Concatenate.
var_dump($first . $second);
```

It's a similar example to what we've seen previously. So, what's the result?

Example 09: Output.

```
string(16) "Value of: 27.325"
```

Just as we expected! The float value has been converted to a string, and stuck to the end of the other value. The same would happen for all basic data types.

I've got a question. Actually, I know the answer... it's just that sometimes I find asking questions is a great way to get that brain of yours ticking!

Why can't we use the addition operator to concatenate strings? I mean, we're just adding two strings together, aren't we?

In other languages, Javascript for example, you can use `+` to concatenate strings. In fact, it's the most accepted way to do so. In PHP, the addition operator is purely mathematical. Let's see what happens if we attempt to perform addition on two strings.

Example 10: Addition on strings.

```
<?php

// First value.
$first = 'Pandas are ';

// Second value.
$second = 'awesome!';

// Concatenate.
var_dump($first + $second);
```

What's the result? It's *nothing* impressive!

Example 11: Output.

```
int(0)
```

PHP understands that our strings are complicated, and so, it treats them as the integer value of zero. Two zeroes equal zero.

If our strings represented real numbers in string format, then PHP would be able to do something a little more sensible with them. Let's try it!

Example 12: Addition on numeric strings.

```php
<?php

// First value.
$first = '3';

// Second value.
$second = '5';

// Concatenate.
var_dump($first + $second);
```

This time, the result makes more sense to us.

Example 13: Output.

```
int(8)
```

We call the process of transforming from one type to another 'casting'. The casting of strings to numeric values within mathematical operations is either convenient or dangerous, depending on your circumstance. It's convenient that we don't have to cast these values ourselves, but sometimes casting doesn't give us the value we expect. For example, `three` won't cast to `3`, it will be `0`. Be sure to use caution when using strings within mathematical operations.

PHP comes with a number of functions, little machines that do work for us, that can be used to perform a number of operations on strings. We can reverse them, replace sections of them, extract sub-strings, calculate length and much, much more. There are more string functions than I can count. Don't worry. We'll be looking at functions in a later chapter. For now, let's take a look at the 'Array' data type in the next chapter.

8. Arrays

It's time for my favourite data type. I absolutely love arrays! They are fantastically fun, and really useful. Arrays are great for when one value isn't enough, and for some reason the collection of values are related.

Indexed

Let's say that we want to store the names of all of our pandas within our application. They are all pandas. They are related. It doesn't make sense to create separate variables for each of them.

Let's store them in an array, shall we? What?! Don't tell me you didn't see that coming. You saw the title, didn't you?

Example 01: Creating arrays.

```php
<?php

// Create an array.
$pandas = array('Lushui', 'Jasmina', 'Pali');

// Create an array as well.
$pandas = ['Lushui', 'Jasmina', 'Pali'];
```

Here we have two examples of arrays. They both work out the same, but the second example was a syntax that was added in PHP version 5.4. This means that it won't work on versions before 5.4. However, I highly recommend using the second format, since the PHP world is moving forward, and we'll soon all be using version 5.6.

Some people like to use the first type of array in code that they intend to share with others, so that it remains compatible with *all* versions of PHP. We'll be using the *new* array syntax for most of the examples in this book. You're free to use the older ones if you choose to do so.

Array values are held between either an `array(` and a `)` or between an `[` and a `]` (square brackets). Each element of an array must be separated by a comma: `,`. The last option in an array can also take an optional comma, but I prefer to leave it out. In the example above, we've added three string values to an array. Arrays can hold strings, other data types, or even other variables!

Here are some more examples.

Example 02: Further examples.

```php
<?php

// Create an array of strings.
$pandas = array('Lushui', 'Jasmina', 'Pali');

// Create an array of integers.
$integers = [3, 6, 9, 12];

// Create an array of floats.
$floats = [1.30, 2.60, 3.90, 4.120];

// Set some variables.
$one    = 1;
$two    = 2;
$three  = 3;

// Create an array of variables.
$variables = array($one, $two, $three);
```

Arrays are extremely flexible too. They can hold a mix of different types of values. Here's another example.

Example 03: Creating arrays with mixed types.

```php
<?php

$one = 1;

// Create an array of mixed values.
$mixed = array('Lushui', $one, 5, 23.54);
```

Great! They're extremely flexible, but how can we use them? We want to be able to retrieve these values that we've collected together, right? Don't worry, we can get to them!

Let's assume that we're working with the first example. Here it is once again to refresh your memory.

Example 04: Refresher.

```php
<?php

// Create an array of pandas.
$pandas = ['Lushui', 'Jasmina', 'Pali'];
```

Let's try to access 'Lushui'. This is an indexed array so we can access each individual element by their position in the array. Lushui is the first item, so let's go ahead and attempt to access it.

Example 05: Accessing array values by index.

```php
<?php

// Create an array of pandas.
$pandas = ['Lushui', 'Jasmina', 'Pali'];

// Get the first item from the array.
$lushui = $pandas[1];

// Echo the result.
echo $lushui;
```

We can access our array items by using square brackets at the end of the array variable and providing an integer value for position between them. Let's go ahead and run the code. I'm eager to meet Lushui!

Example 06: Output.

```
Jasmina
```

Wait, what!? Jasmina? Well, I can't be mad at Jasmina. She's a beautiful red panda that belongs at Bristol zoo, but I was expecting Lushui! Why are we graced with Jasmina's presence? It's because we're programmers.

> Actually, I'm still learning.

Shh you! You've got this far, and you haven't quit on me. You're practically a programmer already.

Here's a secret. Programmers count from zero. That means that position `1` is in fact the second element in our array. This is why Jasmina is chewing on our trouser legs.

Let's fetch Lushui using position zero so that Jasmina has someone to play with!

Example 07: Always start from zero.

```php
<?php

// Create an of pandas.
$pandas = ['Lushui', 'Jasmina', 'Pali'];

// Get the first item from the array.
$lushui = $pandas[0];

// Echo the result.
echo $lushui;
```

Now, with a little bit of luck...

Example 08: Output.

```
Lushui
```

Hurray! Here's our lovely little Lushui.

We can provide any index that we want to retrieve our pandas from the array. Here's another example.

Example 09: Accessing other indexes.

```php
<?php

// Create an of pandas.
$pandas = ['Lushui', 'Jasmina', 'Pali'];

// Fetch our pandas into separate variables.
$lushui     = $pandas[0];
$jasmina    = $pandas[1];
$pali       = $pandas[2];
```

Let's see what happens if we try to retrieve value `3` from the array. Since our arrays are zero-based, that means that nothing should exist at position three, right? Let's find out. First we'll need a snippet to test.

Example 10: Indexes that don't exist.

```php
<?php

// Create an of pandas.
$pandas = ['Lushui', 'Jasmina', 'Pali'];

// Fetch a panda that doesn't exist.
$fakePanda = $pandas[3];

// Dump the result.
echo $fakePanda;
```

There, that should do the trick! Right then, let's execute the code and see what happens.

Example 11: Output.

```
PHP Notice:  Undefined offset: 3 in <FILENAME> on line 7
```

Oh dear! We've got a notice. It's not quite an error, but it's not what we want. When you've learned about functions you'll be able to count the number of elements in an array which will help to avoid this error. For now, you'll have to be careful!

It's time to take a look at another type of array.

Associative

Associative arrays are ones with user defined keys. In some languages, these are known as maps, hashes or dictionaries. In the previous section the keys to our arrays were integers that were provided automatically. Why don't we try providing our own? Let's try to have a little more control over our arrays.

The keys for our our associative arrays must be strings. So let's create a map of number names to their integer values.

Example 12: Associative arrays.

```
<?php

// Create an associative array.
$numbers = [
    'one'       => 1,
    'two'       => 2,
    'three'     => 3,
    'four'      => 4,
    'five'      => 5,
    'six'       => 6
];
```

Our array keys and values are separated by an equals = and greater-than > symbol stuck together. We call this the array assignment operator. The keys can be found on the left, and the values on the right. Otherwise, the array takes a similar format to an indexed one.

Let's try and retrieve the value at position zero.

Example 13: Accessing an associative array by index.

```
<?php

// Create an associative array.
$numbers = [
    'one'       => 1,
    'two'       => 2,
    'three'     => 3,
    'four'      => 4,
    'five'      => 5,
    'six'       => 6
];

// Dump a value.
echo $numbers[0];
```

Let's go ahead and run our file again.

Example 14: Output.

```
PHP Notice:  Undefined offset: 0 in <FILENAME> on line 14
```

Oh no! There's that error again. Wait, I know! It's because we provided our own keys. PHP didn't have to provide numeric keys for us, so using integer values isn't going to work. Let's try using one of our keys to retrieve a value instead.

Example 15: Accessing array values by name.

```
<?php

// Create an associative array.
$numbers = [
    'one'       => 1,
    'two'       => 2,
    'three'     => 3,
    'four'      => 4,
    'five'      => 5,
    'six'       => 6
];

// Dump a value.
echo $numbers['three'];
```

Let's try running our application again. Fingers crossed.

Example 16: Output.

```
3
```

Great! That's just what we wanted. Now we've learned how we can create our own keys to make our arrays more manageable. There's one final trick to arrays that I'd like to share with you.

Multi-dimensional

Now, we know that red pandas are multi-dimensional. They open portals through the fabric of space and time to travel to people who need cuteness and happiness. They also use these portals to travel to areas with vast quantities of fresh fruit,

so they aren't entirely selfless. Well as it turns out, arrays can also be multi-dimensional.

Remember when I told you that arrays can hold any data type? Well it turns out that arrays are a data type. Can you think of any reason why an array shouldn't be able to hold other arrays? No? Well neither can I. Let's try it.

Example 17: Multi-dimensional arrays.

```
<?php

// Create a multi-dimensional array.
$numbers = [
    'prime'         => [2, 3, 5, 7, 11],
    'fibonacci'     => [1, 1, 2, 3, 5],
    'triangular'    => [1, 3, 6, 10, 15]
];
```

Here we have a multi dimensional array of popular mathematical patterns. The outer array is associative, with keys representing the pattern names. The internal arrays are indexed ones. They could be associative as well if we had wanted. I just thought that this might keep it simple.

We can use a concept of 'depth' to describe an array of this kind. This array is two layers deep. It has an outer layer (the associative array) and and several secondary layers (the indexed arrays). Arrays can be as deep as we like, once again I've kept it to two layers for the sake of simplicity.

If we wanted to retrieve the third value from the Prime sequence array, we've discovered that we can use a snippet similar to the following.

Example 18: Retrieving values from multi-dimensional arrays.

```
<?php

// Create a multi-dimensional array.
$numbers = [
    'prime'         => [2, 3, 5, 7, 11],
    'fibonacci'     => [1, 1, 2, 3, 5],
    'triangular'    => [1, 3, 6, 10, 15]
];

// First get the prime numbers array.
$primes = $numbers['prime'];

```

```
13 // Next get the third (second, zero-based) number.
14 echo $primes[2];
```

Of course, we receive the value '5'.

We know that this works, but I've got a better way. We can shorten this. Shall I share it with you? Well, I suppose that you have bought the book, so that means we have a legally binding contract, right? I have to share with you. Fine! Let me explain through the medium of interpretive dance. Actually, hold on. A code snippet might make more sense.

Example 19: Accessing multi-dimensional array values directly.

```
1  <?php
2
3  // Create a multi-dimensional array.
4  $numbers = [
5      'prime'          => [2, 3, 5, 7, 11],
6      'fibonacci'      => [1, 1, 2, 3, 5],
7      'triangular'     => [1, 3, 6, 10, 15]
8  ];
9
10 // Access our prime number directly.
11 echo $numbers['prime'][2];
```

Using additional sets of brackets, we can step deeper into our multi-dimensional array to access nested values directly. We can provide as many sets of brackets as we need to. We can even mix numeric and string based keys.

It's worth noting that if any of the indexes in the chain are missing we will receive our old friend the 'Undefined index' notice.

Multi-dimensional arrays are a great way of expressing grid based data, or even 2D/3D coordinates. Most of the time, however, they are simply used to express complex data structures.

In the next chapter, we'll take a look at how we can cast the values into different data types.

9. Casting

Things don't always go the way you planned, do they? Variables aren't always the right type. Fortunately, with PHP we can use a method known as 'casting' to force one data type to be another.

Basic Usage

Let's say we have a string with the following content.

Example 01: Nothing new here.

```
1  <?php
2
3  $panda = '3';
```

Sure, it's a 3, but internally PHP is holding it as a string, and treating it as a string. We can see this by dumping the value. If we use `var_dump()` on the `$panda` variable, we'll receive the following output

Example 02: Output.

```
1  string(1) "3"
```

See? PHP knows. Sure, when you perform arithmetic on the above string, PHP will cast the value to a numeric one internally, but what if we want to do it ourselves? What if we want to ensure that the variable is holding an integer instead. This is where casting is useful.

In PHP, there are a number of functions to help cast, but as we've not started on functions yet, let's use the shortcuts that are available to us. We can cast a value into another data type by specifying the new data type during the assignment process. It sounds complicated, but it's actually quite simple. Let's take a look at an example in code.

Example 03: Casting to integer.

```
<?php

// Set a string value.
$panda = '3';

// Cast to integer.
$panda = (int) $panda;

// Dump result.
var_dump($panda);
```

We set a string representation of the number three on the first line. There's nothing new there. On the next line we use rounded `(` brackets `)` containing a new data type to cast the provided value (in this case our variable) to whatever data type was placed between the brackets. We've decided that it should be an integer, so we've provided the type `int` to our casting brackets. We could also have provided the type `integer`, they are interchangeable, but I find `int` is much faster to type, and much neater!

Let's take a look at the result.

Example 04: Output.

```
int(3)
```

Fantastic! Our panda is now an integer instead of a string. Shall we try casting the string to a float? Can you guess the syntax? It's simple!

Example 05: Casting to float.

```
<?php

// Set a string value.
$panda = '3';

// Cast to float.
$panda = (float) $panda;

// Dump result.
var_dump($panda);
```

That's right, we just use the type `float` instead of `int`. Let's take a look at the result.

Example 06: Output.

```
float(3)
```

Great! Our value is now a float.

Our casts can also work in reverse. If we want a float value represented as a string, we could use the following example.

Example 07: Casting to string.

```
<?php

// Set a float value.
$panda = 3.4567;

// Cast to string.
$panda = (string) $panda;

// Dump result.
var_dump($panda);
```

If we dump out the result, we'll find that our float value for `$panda` is now a string.

Example 08: Output.

```
string(6) "3.4567"
```

It's clear what these casting examples achieve. I think that it's what we expected of it. Arrays are a little more difficult, though. What do you think will happen if we try to cast a string to an array?

When we have a question such as this, it always makes sense to write a little snippet to experiment with. Sometimes in my day-to-day work I forget what will happen with complicated casts or arithmetic, and I will construct a small demo snippet just to confirm my own understanding.

Example 09: Casting to array.

```php
<?php

// Set a string value.
$panda = 'Lushui';

// Cast to array.
$panda = (array) $panda;

// Dump result.
var_dump($panda);
```

We'll use the `array` type to cast to an array. So, what happened?

Example 10: Output.

```
array(1) {
    [0]=> string(6) "Lushui"
}
```

Well that's interesting! It looks like PHP has wrapped our string within an indexed array. In other languages, a statement such as this might have split our string up into an array of characters (letters), but PHP doesn't have the concept of a character type, and so it handles the statement a little differently.

This wrapping of a string is actually fairly useful. If a piece of our code demands that the variable it uses is an array, then we can simply cast to an array to have the value wrapped.

What happens if we try to cast a float value to an array? I suppose it will wrap that too, shall we check?

Example 11: Casting float to array.

```php
<?php

// Set a float value.
$panda = 1.98765;

// Cast to array.
$panda = (array) $panda;

// Dump result.
var_dump($panda);
```

Let's run it!

Example 12: Output.

```
array(1) {
    [0]=> float(1.98765)
}
```

Great! As we had expected, our float has been wrapped in an array too.

Don't be afraid to use casting to bully your values into the right data type. They are here to work for you, after all!

In the next chapter we'll be learning more about comments.

10. Comments

We've been using single line comments up until now to help explain what's happening within our code snippets. It's time to take a look at the other types of comments available to us, so that we have better methods of learning for when our snippets become a little more complicated.

Single Line

Single line comments are the ones that we've been using so far. They begin with a double `//` slash, and can occupy a single line. Like this.

Example 01: Single line comments.

```
1  <?php
2
3  // This is a single line comment.
```

Whitespace doesn't effect them; they begin at the double forward slash, and terminate at the end of the line. This means that we could place them after our code if we prefer. Here's an example.

Example 02: Another single line comment.

```
1  <?php
2
3  $panda = 'Awesome'; // Set panda to awesome.
```

This works just fine. However, I prefer to place my comments on the line above my code. This makes the line much shorter and easier to read. Within the examples in this book, I'll be placing my comments on the line above the code that they are describing. You're welcome to do the same, or to choose your own position.

If we want for our comment to span multiple lines, we can quite happily provide more single line comments, like this.

Example 03: Many single line comments.

```
<?php

// Pandas are awesome.
// It's because they are kind of like cats.
// Except more fluffy, and they kind of look like
// how cats look in anime.
```

There's a much better way though.

It's worth noting that a hash `#` character can also be used in the place of a `double slash` to begin a comment. However, the double slash is preferred, and the hash comments are simply a relic of a bygone era!

Multi-Line

If we want for a comment to span multiple lines, we can use the multi-line syntax instead. Let's take a closer look, shall we?

Example 04: Multi-line comment.

```
<?php

/* Pandas are awesome.
   It's because they are kind of like cats.
   Except more fluffy, and they kind of look like
   how cats look in anime.
*/
$pandas = 'Awesome!';
```

Everything between the `/*` and the `*/` will count as a comment. Even if it spans multiple lines. The comment looks a little unbalanced though, so what many developers like to do is use additional stars on each line for alignment. Here's an example.

Example 05: Multi-line comment with asterisks for alignment.

```
<?php

/* Pandas are awesome.
 * It's because they are kind of like cats.
 * Except more fluffy, and they kind of look like
 * how cats look in anime.
 */
$pandas = 'Awesome!';
```

As you can see, all of the star `*` characters of the comment are in alignment, and the comment looks much, much neater.

These types of comments can be really useful using a process known as 'commenting out code' to aid the debugging process. Here's an example.

Example 06: Some code.

```
<?php

// Set a string.
$pandas = 'awesome!';

// Set a string.
$pandas = 'fantastic!';

// Output.
echo 'Pandas are ' . $pandas;
```

We know that this piece of code will output `Pandas are fantastic!`. Let's imagine that we want to try and see how it looks if pandas are awesome, instead. We might want to go back to them being fantastic for later, so this test is only temporary. Let's comment out the middle bit.

Example 07: Commented out code.

```php
<?php

// Set a string.
$pandas = 'awesome!';

/*
// Set a string.
$pandas = 'fantastic!';
*/

// Output.
echo 'Pandas are ' . $pandas;
```

There, we've used the multi-line comment symbols to skip the middle statement so that PHP ignores the code and pandas are awesome again. If we'd like them to be fantastic once more, we simply remove the symbols.

There's one final form of comment that we need to take a look at. It's a little advanced, but I think you can take it! You're a smart, beautiful AND / OR handsome reader, after all.

Doc Blocks

Doc blocks are a special type of comment. Other programs known as documentation generators can read these special types of comments, and use the information provided within to generate documentation for your project.

Doc blocks look similar to multi-line comments, except that the opening tag uses two stars instead of one. Here's an example.

Example 08: Doc block.

```php
<?php

/**
 * Description of pandas.
 */
$pandas = 'awesome!';
```

When our automated documentation system runs, our `$pandas` variable will have a description of what it does. If we want to, we can provide additional information that might be useful to our documenter. Let's have another example, shall we?

Example 09: Doc block with metadata.

```
<?php

/**
 * Description of pandas.
 *
 * @var string
 */
$pandas = 'awesome!';
```

We can provide extra metadata to our doc blocks using properties that are prefixed with an `@` symbol. Parameters associated with these properties are placed on the same line, and multiple parameters can be separated with spaces.

With the provided `@var` property, our documentation system knows that the `$panda` variable contains a string value, and can display this information to help others learn more about the code.

Don't worry too much about these doc blocks, we'll stick to the other comments for now. In a later chapter I'll start to introduce them. I just wanted you to be aware of them, just in case you happen to see them somewhere else.

In the next chapter, we'll be taking a look at forks, so you better go and raid your cutlery drawer!

11. Forks

Right now your application follows the sequence of lines that you've placed in your source files. Life isn't like that, though, is it? We don't just blindly follow orders. We make decisions based on the information around us.

In PHP, our information is held within variables. To make decisions based on the value of these variables we can use forks. They fork the flow of our application, changing its path depending on the result of a comparison. In PHP, we call this an `if` statement.

If

Let's take a look at a basic if statement. I think it might be best if we lead with an example, don't you?

Example 01: The if statement.

```
1  <?php
2
3  if (true) {
4      echo 'Yey, panda time!';
5  }
```

An if statement always begins with an `if`. Well, I'm sure you'd worked that out. Next we have two sections. The code within the `(` rounded `)` brackets is the condition. If the condition evaluates to a `true` value, then the code within the `{` curly `}` braces will be executed.

Let's execute this snippet to see what happens.

Example 02: Output.

```
1  Yey, panda time!
```

Awesome! It's always great when it's panda time. The echo statement within the curly braces is executed because the condition evaluates to `true`. Well, the condition **is** true, but that still evaluates to a boolean `true`.

Let's see what happens if we change the condition to something that evaluates to a boolean false. How about this?

Example 03: Another if statement.

```
<?php

if (false) {
    echo 'Yey, panda time!';
}
```

Let's go ahead and execute the code one more time.

Example 04: Output.

```
(nothing)
```

There we go, nothing happened. The code within our curly braces isn't executed when our condition evaluates to `false`.

You might be asking yourself why this is useful? Why can't we just add or remove the line of code when we need it? Well, it's because providing simply a boolean doesn't make the if statement very useful. It was just a clean way of explaining how conditions are evaluated as booleans.

Consider the following variation. There's a slight difference, can you spot it?

Example 05: Comparing strings.

```
<?php

$panda = 'Lushui';

if ($panda == 'Lushui') {
    echo 'Yey, panda time!';
}
```

This time, we're comparing the value of a string variable to another string. Once again, the echo statement will be executed. This is because `$panda` does equal 'Lushui'. This means that the condition once again evaluates to `true`.

Let's change the value of `$panda` to see what happens.

Example 06: Comparing different strings.

```
<?php

$panda = 'Pali';

if ($panda == 'Lushui') {
    echo 'Yey, panda time!';
}
```

We'll execute the code once more.

Example 07: Output.

```
(nothing)
```

There we go. This time, `$panda` does not equal 'Lushui'. This means that the condition will equate to `false`, and the code within the curly braces (actually, we call this a 'block') will not be executed.

Right now, we're controlling the value of `$panda`, but what if `$panda` was set as the result of user input? If we set `$panda` to whatever our user typed into our application. Perhaps it could be data acquired from another part of a big application. This data would be able to change the flow of our application, allowing sections of code to be executed only under certain conditions. This is when programming gets exciting.

Any statement within the condition will be evaluated. Go ahead and experiment with this basic if statement for a while before moving to the next section. It will be a very important part of your new career in programming.

Else

ELSE!? Else what!? Sorry. It just sounded like one of those dramatic moments in a soap opera. You, know. When someone threatens someone else with a 'or else!'. Actually, `else` is just another extension to the `if` statement. Shall we take a closer look?

Example 08: If with an else.

```php
<?php

$panda = 'Pali';

if ($panda == 'Lushui') {
    echo 'Yey, Lushui time!';
} else {
    echo 'Oh hey there Pali!';
}
```

The `else` portion of our if statement functions in a similar fashion to the `if` portion. The only difference is that `else` doesn't require a condition. The code within the `else` curly brackets will be executed only if the condition within the `if` statement equates to `false`.

This means that our `if` statement can now react to both circumstances, whether the condition evaluates to `true` or `false`. This is a definite fork. Two different parts of the application can execute depending upon a condition.

Well that was simple, wasn't it? Let's get to the final component of the `if` statement.

Elseif

Woah! That's both of those words combined! Now things are getting really serious. Right then, you know how this works by now. Let's get started with an example.

Example 09: The elseif statement.

```php
<?php

$panda = 'Pali';

if ($panda == 'Lushui') {
    echo 'Yey, Lushui time!';
} elseif ($panda == 'Pali') {
    echo 'Oh hey there Pali!';
} elseif ($panda == 'Jasmina') {
    echo 'Looking pretty Jasmina!';
}
```

Hey look! More paths! Using the `elseif` blocks we are able to provide alternate conditions for our fork. PHP will work its way down the list of conditions, and execute the code block for the first one that evaluates to `true`.

In the above example, we can see that if we were to alter the `$panda` variable, there are four possible outcomes to our snippet of code. Let's list them for clarity.

- `$panda` is **Lushui** - *Yey, Lushui time!*
- `$panda` is **Pali** - *Oh hey there Pali!*
- `$panda` is **Jasmina** - *Looking pretty Jasmina!*
- `$panda` is **something else**. - *(nothing)*

If we want to, we can also use `else` alongside `elseif` to provide a default execution path for when none of the other conditions match. Let's take a look at an example.

Example 10: The else statement.

```php
<?php

$panda = _SOMETHING_;

if ($panda == 'Lushui') {
    echo 'Yey, Lushui time!';
} elseif ($panda == 'Pali') {
    echo 'Oh hey there Pali!';
} elseif ($panda == 'Jasmina') {
    echo 'Looking pretty Jasmina!';
} else {
    echo 'Sorry, who are you?';
}
```

Experiment with the different combinations of `if`, `elseif` and `else` to see how different outcomes of conditions effect the flow of execution within your code. We're starting to become real developers now, can you feel it?

Switch

When you find yourself using many `elseif` statements to compare the same variable, you're probably going to want to use a switch statement instead. It's a much cleaner and more efficient way of reacting to different values in different ways.

Let's take a look at an example switch statement. We'll start small. First, let's take a look at the switch block.

Example 11: An empty switch block.

```
<?php

switch ($panda) {

}
```

The `switch` statement takes a similar shape to that of the `if` statement. Except this time, it's not a condition that is contained within the curly braces. Instead it's the value that we wish to compare.

Let's add some case statements to the example so we can see how it works.

Example 12: A switch block with a case.

```
<?php

$panda = _SOMETHING_;

switch ($panda) {

    case 'Lushui':
        echo 'Yey, Lushui time!';

}
```

Within the curly braces of our `switch` statement, we provide our `case`'s. Case's let us compare the value of the variable provided to the switch, and execute lines of code if there is a match.

In the above example, we create a line beginning with the word `case`. Next, we provide the comparison value followed by a `:` colon. On the next line(s) we provide the code that we wish to be executed if there is a match.

We can add as many case statements as we like. Here's another example with multiple comparisons.

Example 13: A switch block with multiple cases.

```php
<?php

$panda = _SOMETHING_;

switch ($panda) {

    case 'Lushui':
        echo 'Yey, Lushui time!';

    case 'Pali':
        echo 'Oh hey there Pali!';

    case 'Jasmina':
        echo 'Looking pretty Jasmina!';

}
```

Here we can see multiple `case` statements. Be careful, though! There's a gotcha in the snippet! You see, whichever case statement matches, the code from the other `case` statements below will also be executed.

This means that if the value of `$panda` is `Lushui` then all three echo statements will be executed. This could be really useful if we wanted them to be executed, but right now, we don't! So how can we make sure that only a single echo statement is executed? Simple! We'll take a `break` or three!

Example 14: The break statement.

```php
<?php

$panda = _SOMETHING_;

switch ($panda) {

    case 'Lushui':
        echo 'Yey, Lushui time!';
        break;

    case 'Pali':
        echo 'Oh hey there Pali!';
        break;
```

```
14
15      case 'Jasmina':
16          echo 'Looking pretty Jasmina!';
17          break;
18  }
```

If we place a `break` statement on the line after our echo statements, we can stop the execution of the switch statement at that point. The flow of execution will break out of the switch and continue after the final `}` closing curly brace.

These `break` statements will also have the same effect on `if`, `elseif` and `else` statements. Break is a keyword that will break the execution of the current fork. It's a useful one to remember!

There's one final thing that's missing from our switch statement. Right now, it's similar to how our `elseif` statement functioned without the `else` on the end. So, how do we replicate the functionality with the `else`? Well it's simple, we must simply provide a `default`!

Let's change our example, shall we?

Example 15: Default.

```
1   <?php
2
3   $panda = _SOMETHING_;
4
5   switch ($panda) {
6
7       case 'Lushui':
8           echo 'Yey, Lushui time!';
9           break;
10
11      case 'Pali':
12          echo 'Oh hey there Pali!';
13          break;
14
15      case 'Jasmina':
16          echo 'Looking pretty Jasmina!';
17          break;
18
19      default:
20          echo 'Sorry, who are you?';
21
22  }
```

The `default` statement works in a similar fashion to the `case` statement, except that you don't need to provide a comparison value. The code below the `default` will be executed if none of the other `case` statements have matched.

The `default` statement doesn't require the `break` because it's the last part of the switch statement. However, if you'd like to add it for clarity, then it won't make a functional difference.

In the next chapter we'll be taking a look at iterators. Get excited! Go on, I dare you!

12. Loops

Iterators are useful when you have a collection of values. Sometimes, the word 'loop' is a better way to describe an iterator. We'll use these techniques to allow a piece of code to run repeatedly until (or while) a condition is met.

Some loops are like saying "Knock on a door until someone answers.", and others are more similar to "Count every body buried in my basement.". Let's take a closer look at how these structures might be implemented in PHP.

While

The while loop will continue to run a snippet of code while the provided condition is true. Let's construct an example using a counter.

Example 01: The while loop.

```php
<?php

// Set the panda counter to zero.
$pandas = 0;

// Iterate while $pandas less than 50.
while ($pandas < 50) {

    // Output the number of panda butlers.
    echo "We have {$pandas} panda butlers!\n";

    // Increment the counter.
    $pandas++;
}
```

First, we'll set a counter called `$pandas` to zero. Now we can begin creating our first loop. Hurray! We need to begin our while loop with the word `while` followed by a condition within the `(` rounded `)` brackets, and finally a block of code held within `{` curly `}` braces.

The `while` loop takes a similar form to the `if` statement. The difference is that the code within the `{` block `}` portion will continue to execute, over and over again, while the condition is equal to `true`.

Let's execute the code and see what happens, I'm excited to meet the pandas.

Example 02: Output.

```
We have 0 panda butlers!
We have 1 panda butlers!
We have 2 panda butlers!
We have 3 panda butlers!
We have 4 panda butlers!
We have 5 panda butlers!
... lots more! ...
We have 41 panda butlers!
We have 42 panda butlers!
We have 43 panda butlers!
We have 44 panda butlers!
We have 45 panda butlers!
We have 46 panda butlers!
We have 47 panda butlers!
We have 48 panda butlers!
We have 49 panda butlers!
```

Woah! That's a lot of pandas. The loop begins and checks that the value of `$pandas` is less than 50. It is less than 50, in fact, it's set to 0, so the code within the block portion of the loop is executed. This means that the count of pandas is echoed out, and the `$pandas` counter is incremented.

The loop begins once again, this time with a `$panda` value of one. It's still less than 50, so the block is executed again. This process happens over and over again, until the 51st iteration of the loop, when the value of `$pandas` is finally greater than 50. The condition is now evaluated as false, the block isn't executed this time, and our code continues to the next line after the while loop.

While loops are a fantastic way of waiting for something. Let's try a more applicable solution that will demonstrate this loop in action.

Example 03: Waiting.

```
<?php

// Create an empty database connection.
$database = null;

// Iterate while the database isn't set.
while ($database == null) {

    // Attempt to connect to the database.
```

```
10      $database = createDatabaseConnection();
11
12  }
```

For now, ignore the `createDatabaseConnection()`, we haven't taken a look at functions yet. Just go ahead and assume that it will attempt to set the `$database` variable to an instance of a database connection.

We begin with a `$database` variable set to `null`, and since the loop checks that the value is `null` it will certainly be executed once. The loop block will attempt to create a new database connection. If it does, then the code will continue. If the creation of the database connection fails, then the `$database` variable will still equal `null`, and the loop will execute once more.

This means that the code will continue to loop until we have a database connection. We could put our database code after the loop, and feel confident that we will have an active database connection when PHP gets to that part of our application.

Do While

The do while loop is very similar to the while loop, except for one minor difference. Let's take a look at a syntax example. Why don't you try to spot the difference?

Example 04: The do-while loop.

```
1   <?php
2
3   // Set the panda counter to zero.
4   $pandas = 0;
5
6   // Iterate...
7   do {
8
9       // Output the number of panda butlers.
10      echo "We have {$pandas} panda butlers!\n";
11
12      // Increment the counter.
13      $pandas++;
14
15  // ... while $pandas less than 50.
16  } while ($pandas < 50);
```

Well, did you work it out? First let's take a look at the syntax. We begin with a `do` keyword. Unlike the `while` loop, next we have the block section of our loop container within curly braces. Next, we have the `while` keyword, followed by our condition.

The condition works in an identical way to the `while` loop. The only difference between these two loops is that the do while loop will *always* execute the code block at least once. It will execute the block before it checks the condition for the first time.

For

It's time to take a look at a `for` loop. This one will be a little tricky to explain, so we had best lead with an example.

Example 05: The for loop.

```
<?php

// Iterate for 50 repetitions.
for ($i = 0; $i < 50; $i++) {

    // Output the number of panda butlers.
    echo "We have {$i} panda butlers!\n";
}
```

In the above example we've recreated a similar loop to our original `while` loop, except that we don't need to manage the counter variable ourselves.

We're more than familiar with the code block held between the curly braces, so let's take a close look at the first line.

Example 06: The first line.

```
<?php

for ($i = 0; $i < 50; $i++)
```

Of course, our `for` loop begins with a `for` keyword. Next, we have a set of `(` rounded `)` brackets. These are normally used to hold a strict condition, but the `for` loop is a little different. Within the brackets we find three short statements.

The first statement is used to set the counter for our loop to its initial value.

Example 07: Initialize a counter.

```
1  <?php
2
3  $i = 0;
```

In this instance we're setting a variable named `$i` to zero. Normally, I'd encourage you to use descriptive names for your variables, but `$i` is somewhat of a tradition. In fact, most of the variables used during the construction of a for loop are single letters, and `$i` appears to be the most common of all. I'm honestly not sure why this is, but it's a tradition that has been taught throughout the ages. If anyone reading happens to know where this tradition came from, then please do let me know! I'll update the book. My best guess would be that `$i` stands for iteration counter!

Next, we have our condition.

Example 08: Set a condition.

```
1  <?php
2
3  $i < 50;
```

This is the part of the loop we are familiar with from the previous sections. We're telling PHP to continue to execute our code block while the value of `$i` is less than fifty.

Finally, we have a statement to adjust our value.

Example 09: Modify the counter.

```
1  <?php
2
3  $i++;
```

We can use this to increment our counter variable. This statement will be executed with each iteration of our loop. This means that the first line of our for loop could be expressed as the following format.

Example 10: The syntax.

```
<?php

for (_BEFORE_; _CONDITION_; _AFTER_EACH_)
```

The first section is executed **before** our loop is executed. The second statement, or the condition, is evaluated **every time** the loop is executed. Finally, the third statement is executed after each loop iteration.

As you can see from the fewer amount of lines in the example, a for loop is a great way of implementing a loop when some form of counter is involved.

Don't forget! You're not limited to purely incrementing counters. You could increase the counter by a greater amount for a larger step, or even multiply it!

Foreach

Foreach loops are useful when used in combination with iterable types such as arrays. The number of iterations of this loop is determined by the number of items within our array. Let's take a look at an example.

Example 11: The foreach loop.

```
<?php

// Create an array of pandas.
$pandas = ['Lushui', 'Pali', 'Jasmina'];

// Iterate our panda array.
foreach ($pandas as $panda) {
    echo "Hello there {$panda}!\n";
}
```

We've created a small array of pandas. We're masters of arrays already, aren't we? This is no problem for us at all. Let's focus on the next line instead.

Example 12: Iterating an array.

```
<?php

foreach ($pandas as $panda)
```

The foreach loop starts with a `foreach` keyword. Sorry, spoiler alert! Am I right? Next we have a set of `(` rounded `)` brackets. Within the brackets we have two variables separated by the keyword `as`. We're telling PHP to loop through all of the elements in the array, and for each iteration set `$panda` to the current array element.

The final part of the loop is of course the code block within curly brackets. We're more than familiar with these code blocks by now. They contain the code that is executed with each iteration of the loop.

For the first iteration of our loop `$panda` will be set to 'Lushui', for the next iteration `$panda` will be set to 'Pali', and for the final iteration `$panda` will be set to 'Jasmina'.

This is my most used loop. I'd say I use it roughly 23,543 times a day. Something like that! It's because I enjoy working with arrays.

In a previous chapter, we discovered how to provide our own keys for arrays. You might want access to the array key, as well as its value within your for loop. Well don't worry buddy! PHP can do that for you. Let's take a look at the syntax.

Example 13: Foreach loop with keys.

```
<?php

// Create an array of pandas.
$pandas = [
    'first'     => 'Lushui',
    'second'    => 'Pali',
    'third'     => 'Jasmina'
];

// Iterate our panda array.
foreach ($pandas as $position => $panda) {
    echo "You're the {$position} panda, {$panda}!\n";
}
```

This time, we have an array with custom keys. We're using strings for both keys and values, but they could be anything. Instead of simply assigning a new variable for each loop iteration, we provide a temporary key and value placeholder separated by a `=>` symbol.

Let's take a look at the values of both variables with each loop iteration.

- **First Iteration** ($position = 'first') ($panda = 'Lushui')
- **Second Iteration** ($position = 'second') ($panda = 'Pali')
- **Third Iteration** ($position = 'third') ($panda = 'Jasmina')

We can use these temporary variables to accomplish any of our sick little master plans. They are completely at our disposal.

It's worth noting that a *copy* of your array elements are passed into the `foreach` loop block. This means that any changes to the variable within the loop, will not be reflected once the loop has completed.

If you'd like to change the value of the element during the foreach loop, simply use the `key => value` format, and access the property by key through the original array. For example `$pandas[$position] = 'Foo!;.`

This was the final loop that I have to share in this chapter, but before we call it quits, let's take a look at some of the control keywords available to us for use within our loops.

Control

In the forks chapter, we discovered how to use `break` to exit out of a fork and continue with the execution of our code. We can also use `break` inside a loop to exit out and continue processing our application. It's a handy statement like that. Here's a quick example.

Example 14: Breaking a loop.

```
<?php

// Create an array of pandas.
$pandas = [
    'first'     => 'Lushui',
    'second'    => 'Pali',
    'third'     => 'Jasmina'
];

// Iterate our panda array.
foreach ($pandas as $position => $panda) {
    echo "You're the {$position} panda, {$panda}!\n";
    break;
}
```

This loop will only output for the first iteration of the loop. As soon as PHP reaches the `break` statement, it will exit the loop and continue processing the application.

The next control statement is `continue`. Using `continue` in the code block will instruct PHP to complete the current iteration at that point, and continue to the next iteration. It's useful for 'skipping' loop iterations.

Let's take a look at an example. Why don't we throw an `if` statement into the mix? We're a master of those now, aren't we?

Example 15: Continuing iteration.

```
<?php

// Create an array of pandas.
$pandas = [
    'first'     => 'Lushui',
    'second'    => 'Pali',
    'third'     => 'Jasmina'
];

// Iterate our panda array.
foreach ($pandas as $position => $panda) {

    // If the position variable equals 'second'.
    if ($position == 'second') {

        // Break the iteration.
        continue;
    }

    // Output.
    echo "You are the {$position} panda, {$panda}!\n";
}
```

Take a look inside the loop block. We have an if statement that will `continue` to the next iteration if the `$position` temporary array key variable is equal to the string 'second'. Let's take a look at the result, shall we?

Example 16: Output.

```
You are the first panda, Lushui!
You are the third panda, Jasmina!
```

As you can see, the second iteration of the loop hit the `continue` keyword, skipping the `echo` statement and continuing to the third iteration.

In the next chapter, we're going to be taking a look at functions. They'll help to give our code some structure.

13. Functions

Functions are a great way to organise your applications. They can be used to avoid repetition in your codebase. When you find a similar piece of code duplicated many times within your application, it's a good indicator that you should be using a function instead.

You can think of functions as little factory machines. Your variables (little boxes) go into a slot on one side of the machine, some cogs spin, crunching noises happen and something else comes out of the other side.

Basic Usage

Let's take a look at a function. You've been with me for many chapters, I'm sure you know how this works!

Example 01: My first function.

```php
<?php

function echoSong() {
    echo "So take the photographs, and still frames in your mind.\n";
    echo "Hang it on a shelf in good health and good time.\n";
    echo "Tattoos and memories and dead skin on trial.\n";
    echo "For what it's worth, it was worth all the while.\n";
}

echoSong();
```

Right, this is all new! Before we begin the explanation, let's go ahead and execute the code. What's the output that we receive?

Example 02: Output.

```
So take the photographs, and still frames in your mind.
Hang it on a shelf in good health and good time.
Tattoos and memories and dead skin on trial.
For what it's worth, it was worth all the while.
```

We're presented with some lyrics from a classic song. There's bonus points in it if you can name the artist and song title. Answers on a postcard folks! Actually, you can just tweet me!

We should take a look at the function definition. Before a function (little machine) can be used, we must tell PHP what it does. This is our definition. Here's the section that I'm talking about.

Example 03: Function definition.

```php
<?php

function echoSong() {
    echo "So take the photographs, and still frames in your mind.\n";
    echo "Hang it on a shelf in good health and good time.\n";
    echo "Tattoos and memories and dead skin on trial.\n";
    echo "For what it's worth, it was worth all the while.\n";
}
```

We know what the echo lines do, so let's strip out the body of the function definition for clarity.

Example 04: Function outline.

```php
<?php

function echoSong() {
    // Do stuff here.
}
```

The first line in the example is called the method signature. It describes the method, and its shape. With a function this simple, all we really have is a name. We start the function definition with the keyword `function` followed by the name of the function that we are defining. The function name has similar naming rules to variables, so we're going to stick with camelCased names for consistency.

Next, we have a pair of rounded brackets. Later we'll learn how we can give our function parameters, and these brackets will become more interesting, but for now, leave them empty! Finally, we have a block in curly brackets. This is similar to the loops and forks that we have seen in the previous chapters.

Our code snippet is letting PHP know that when we call our `echoSong()` function, it should echo out the lyrics to a classic song which you're kicking yourself because you've forgotten the name of.

Now that we've told PHP about our function, we are able to make use of it. That's what this line does.

Example 05: Calling a function.

```
1  <?php
2
3  echoSong();
```

We can execute a function by providing its name, and a set of rounded brackets. The block within the function we defined will then be executed and our lyrics will be displayed.

We can execute our function as many times as we like. This means that the snippet of code we have created is now re-usable across the whole of our application. We're being efficient.

Here's an example of calling the function multiple times.

Example 06: Calling a function multiple times.

```
1  <?php
2
3  function echoSong() {
4      echo "So take the photographs, and still frames in your mind.\n";
5      echo "Hang it on a shelf in good health and good time.\n";
6      echo "Tattoos and memories and dead skin on trial.\n";
7      echo "For what it's worth, it was worth all the while.\n";
8  }
9
10 echoSong();
11 echoSong();
12 echoSong();
```

Let's execute the code snippet to see the output.

Example 07: Output.

```
So take the photographs, and still frames in your mind.
Hang it on a shelf in good health and good time.
Tattoos and memories and dead skin on trial.
For what it's worth, it was worth all the while.
So take the photographs, and still frames in your mind.
Hang it on a shelf in good health and good time.
Tattoos and memories and dead skin on trial.
For what it's worth, it was worth all the while.
So take the photographs, and still frames in your mind.
Hang it on a shelf in good health and good time.
Tattoos and memories and dead skin on trial.
For what it's worth, it was worth all the while.
```

I'm sure that song was meant to have a chorus! Oh well, our code has executed as it should, calling our function three times. We should create functions for snippets of code that are used often within our application. They will serve to reduce the size of the codebase, and allow for us to change the common functionality in one place, rather than having to dig around our whole application making many changes.

Return Values

The blocks of our functions can be used to return a value or object (more on objects later). It can provide things that are useful to us. Let's take a look.

Example 08: Return values.

```php
<?php

function getMeaningOfLife() {
    return 42;
}
```

Here we have another function definition. This time, we use the `return` keyword to return a value from the function upon execution. When we call the function, we'll receive the value `42` as the result of the statement.

Now I know that we could have simply stored the value `42` in a `$meaningOfLife` variable, but when we continue in to the later sections of this chapter, you'll see why this `return` keyword might be useful. It's not easy keeping these examples simple, you know!?

Let's try calling the function, and assigning the result to a variable. I'll add some comments for clarity. I'm a nice guy like that!

Example 09: Assigning return values.

```
<?php

// Create a function.
function getMeaningOfLife() {

    // Return a value.
    return 42;
}

// Execute function and assign result.
$result = getMeaningOfLife();

// Dump the result.
var_dump($result);
```

The first section with the function definition is exactly the same, I've simply added some comments. On the next line we're executing our function `getMeaningOfLife()` and assigning the result to a variable named.. well.. `$result`. Finally, we'll dump the result. Let's run our code.

Example 10: Output.

```
<?php

int(42)
```

There we go! We see that the `42` that we `return`ed from our function was assigned to the variable when we executed it. We'd receive the same output with the following snippet.

Example 11: Dump the return value.

```
<?php

// Create a function.
function getMeaningOfLife() {

    // Return a value.
    return 42;
}
```

```
9
10 // Dump the result of the function.
11 var_dump(getMeaningOfLife());
```

This time we're dumping the result of the function directly, instead of assigning it.

There's one final thing that we need to keep in mind when using the `return` keyword. As soon as PHP comes across our `return` line, it will exit the function, returning the value. For example, consider the following code sample.

Example 12: Return terminates a function.

```
1  <?php
2
3  // Create a function.
4  function getMeaningOfLife() {
5      return 42;
6      echo 'Here I am, rock you like a hurricane!';
7  }
8
9  // Dump the result of the function.
10 var_dump(getMeaningOfLife());
```

In the above example, when the function `getMeaningOfLife()` is executed, it will return the value `42`, but the `echo` statement will not be executed.

It's worth noting that you can use `return` without a value to end a function before it's natural exit point. Doing this is the same as returning `null`. Here's an example.

Example 13: Empty return statement.

```
1  <?php
2
3  function getMeaningOfLife() {
4      return;
5      echo 'This line will never be executed!';
6  }
```

Parameters

Some machines take input. For example, a 3D printer takes plastic and a blueprint. An espresso machine will take one of those tiny coffee pods and some water.

Functions can also take input. We call these arguments or parameters; they make up part of the method signature, or the description of how the function works. Let's take a look at a simple function parameter.

Example 14: Function parameters.

```php
<?php

// Welcome someone!
function welcome($name) {
    echo "Welcome to PHP Pandas, {$name}!";
}
```

Between the rounded brackets within our function signature, just after the function name, we have our single parameter. We're calling it `$name`. These parameters are placeholders for values that we will be giving to the function when we use it. This is what makes the function dynamic.

The parameters that are given to the function are available for use within the function block. In the above example, we use the `$name` parameter handed to the function, and echo a welcome statement using the provided name.

Of course, this snippet of code won't do anything unless we tell PHP to execute, or 'call', the function. Let's add the call line to our snippet.

Example 15: Passing parameters to functions.

```php
<?php

// Welcome someone!
function welcome($name) {
    echo "Welcome to PHP Pandas, {$name}!";
}

// Call the welcome function.
welcome('reader');
```

We call the method using its name and rounded brackets. However, this time we provide a string value `reader` within the brackets as our parameter. This means that the `$name` parameter of our function will be set to `reader`. It will only be set to this value for this one execution, and any later calls to the function will be replaced by whichever parameter value we provide to it. Our function doesn't have any state.

Let's take a look at the result that we see in our terminal, shall we?

Example 16: Output.

```
Welcome to PHP Pandas, reader!
```

Great work! We receive a warm welcome. Why don't we try using a number of different values? This will serve to prove how the function is now dynamic.

Example 17: Different parameters.

```
<?php

// Welcome someone!
function welcome($name) {
    echo "Welcome to PHP Pandas, {$name}!\n";
}

// Call the welcome function.
welcome('reader');              // This is you!
welcome('Captain Jon Morgan');  // This is an awesome guy!
welcome('Arno Dorian');         // Nothing is true...
welcome(64.54);                 // That's not a name!
```

Here, we are invoking the function four times. Each time, we are supplying a different value. The fourth value is a float, which demonstrates that it doesn't matter what type of data we provide to our function. I've also added a new line character (`\n`) to the echo statement to give it some clarity. Let's execute this snippet.

Example 18: Output.

```
Welcome to PHP Pandas, reader!
Welcome to PHP Pandas, Captain Jon Morgan!
Welcome to PHP Pandas, Arno Dorian!
Welcome to PHP Pandas, 64.54!
```

See how the result of our function changes each time, depending upon the input that we provide? This gives us more control over our functions, and the ability to react differently to different values. As you can see from the fourth value, it doesn't matter what data types we pass into the function, our code is happy.

We aren't limited to only one parameter. We can use as many parameters in our functions as we like. Let's abuse this right, shall we? Let's make one with... a whole five parameters! What rebels we are!

Example 19: Multiple parameters.

```php
<?php

// Sum some values.
function addify($first, $second, $third, $fourth, $fifth) {
    return $first + $second + $third + $fourth + $fifth;
}

// Call the addify function.
echo addify(1, 1, 2, 3, 5);
```

We've written a function that will take 5 values as parameters, and return the sum of them. With our `addify()` function, the order doesn't matter, but it's worth noting that the order that you provide your parameters matches up to the placeholders in the signature. In the above example, the value of `$first` will be `1`, and the value of `$fifth` will be `5`.

Five parameters is great, but it feels like there shouldn't be a limit on a function like `addify()`. We should be able to sum as many figures as we like. I've got an idea! Let's use `func_get_args()`. This is a special PHP function that will return an array of all the values that have been passed as parameters to our function. What's even better, is that we don't need to define any parameters at all for it to work. Here's an example.

Example 20: Infinite parameters.

```php
<?php

function welcome() {

    // Get all function parameters.
    $names = func_get_args();

    // Iterate the names and welcome them.
    foreach ($names as $name) {
        echo "Welcome, {$name}!\n";
    }
}
```

Here we've used another function called `func_get_args()` within our own function. That's right, functions can use other functions too! When we use `func_get_args()` we receive any parameters that are given to the function in the form of an indexed array. We don't even need to create placeholder variables for them. This means that the function above can take as many parameters as you like. Here's an example:

Example 21: Lots of parameters.

```
<?php

welcome('Dayle', 'James', 'Andrea', 'Ben', 'Mateusz');
```

We call the function using five names, and receive the following response.

Example 22: Output.

```
Welcome, Dayle!
Welcome, James!
Welcome, Andrea!
Welcome, Ben!
Welcome, Mateusz!
```

All five people are welcomed. What happens if we change the number of parameters? Let's try it with two names instead.

Example 23: Two parameters.

```
<?php

welcome('Anthony', 'Alex');
```

Our function doesn't care. The array returned from `func_get_args()` will now contain only two values, and the loop will continue as normal.

Example 24: Output.

```
Welcome, Anthony!
Welcome, Alex!
```

Type Hinting

Earlier this chapter, we mentioned those nifty little espresso machines. The ones that take little plastic pods? I'm sure you've seen them. Well what would happen if you put an avocado into the slot where the pods go? Do you think it would work?

> It would make a huge mess!

You're quite right! We'd end up with green gooey avocado all over our lovely espresso machine. It's only made to take coffee pods. Sometimes our functions are only made to take specific input too. If we want to ensure that our parameters are of a certain type, we can use a technique known as type-hinting.

Let's see this in action.

Example 25: Type-hinting an array.

```
<?php

// Say hello to an array of people.
function sayHello(array $names) {
    foreach ($names as $name) {
        echo "Hello, {$name}!\n";
    }
}

sayHello(['Katie', 'Corissa', 'Lucy']);
```

We've got another welcoming function called `sayHello()`. The only new thing that we'll encounter in this example, is that the input array `$names` has been type hinted as an array. We've done this by specifying the type `array` before the parameter placeholder.

If we execute the code, we get the output that we expect.

Example 26: Output.

```
Hello, Katie!
Hello, Corissa!
Hello, Lucy!
```

Let's change the function call to pass a number. Aren't we naughty!

Example 27: Naughty, naughty.

```
<?php

sayHello(4);
```

What happens this time? Let's execute the code.

Example 28: Output.

```
PHP Catchable fatal error:  Argument 1 passed to sayHello() must be of the type \
array, integer given.
```

Interesting! PHP is now angry at us. With the type-hinting in place, PHP will only allow arrays to be passed to the `sayHello()` function. If we pass anything else, we get a fatal error. The `Catchable` part means that we could deal with this error sensibly using a `try` block, but we'll look at that later.

Type-hinting is super useful, isn't it? Well there's a catch. I'm sorry. There's always a catch, isn't there? You can't type-hint scalar values with PHP (yet.. dun dun dun!). Don't ask me why, I didn't write the language! What this means, is that you can't type hint `string`, `integer`, `boolean` and `float` values. It's a real pain, but you'll have to learn to deal with it. It's a hard truth, isn't it?

It seems a little strange just hinting `array`s, doesn't it? Well, later you'll discover that you can also type-hint parameters as classes, but we'll explain that in greater detail after we've learned about classes themselves. One step at a time young padawan. One step at a time...

14. Closures

Shall we get a little closure to closures? Snuggle up a bit? Okay, that was a bad one. It's a weird word though, isn't it? Closures are special functions. These types of functions don't have a name. Much like a Samurai!

Who needs a name anyway?

Example 01: A Closure.

```php
<?php

function () {
    return 'bar';
}
```

Great! Wait, not great. How do we call a function without a name? Well Closures are special, we can assign them to variables just like we would with a simple value. Let's take a look at the syntax for this.

Example 02: Assigning a Closure to a variable.

```php
<?php

$cat = function () {
    echo 'Oh long Johnson!';
};
```

Here, we've assigned an anonymous function, or 'Closure', to a variable called `$cat`. The function simply returns a string associated with a popular feline meme.

Note that we have a semi ; colon appended to our function. This is because it's part of an assignment statement, and functions like any other single line of PHP.

Now that we've trapped this Closure within our variable, we can call it with rounded brackets, just like any other function! For example:

Example 03: Executing an assigned Closure.

```
<?php

$cat();
```

We simply append the brackets to our variable, and the code is executed.

Example 04: Output.

```
Oh long Johnson!
```

Passing functions to functions...

Closures arrived in PHP 5.3, which means that they're fairly recent! They were a game-changer for the PHP world, because they allow you to pass small sections of logic into functions. That's right, you can pass a Closure to a function. Let's see this in action.

Oh, by the way, did I mention that you can type-hint a Closure? Simply place the word `Closure` before a parameter to ensure that only a closure is passed as a parameter to the method. If you were to call `get_type()` on a Closure, you'd receive the value `Closure`. It's a complex object in PHP. We'll see other objects of this type later.

Where were we? Oh yes, an example!

Example 05: Type-hinting a Closure.

```
<?php

// Create a math function.
function math(Closure $type, $first, $second) {

    // Execute the closure with parameters
    return $type($first, $second);
}
```

Okay, what's going on here?

First of all, we've got a function called `math()`, but it's not doing a lot of `math()` at the moment. It receives a type-hinted Closure, and two other variables as parameters.

Well, we're not going to be able to call it without a Closure, are we? Let's make a few Closures so that we can test it. We'll create Closures that perform mathematical operations. I know, I know. There's a lot of math, isn't there? I promise to keep it basic!

Example 06: Two Closures.

```php
<?php

// Create a math function.
function math(Closure $type, $first, $second) {

    // Execute the closure with parameters
    return $type($first, $second);
}

// Create an addition closure.
$addition = function ($first, $second) {

    // Add the values.
    return $first + $second;
};

// Create an subtraction closure.
$subtraction = function ($first, $second) {

    // Subtract the values.
    return $first - $second;
};
```

We've defined two Closures bound to the variables `$addition` and `$subtraction`. They both receive two parameters, and perform subtraction or addition on these values. Let's try to use them with our `math()` function, shall we?

Example 07: Passing Closures to functions.

```php
<?php

// Create a math function.
function math(Closure $type, $first, $second) {

    // Execute the closure with parameters
    return $type($first, $second);
}

// Create an addition closure.
$addition = function ($first, $second) {

```

```
    // Add the values.
    return $first + $second;
};

// Create an subtraction closure.
$subtraction = function ($first, $second) {

    // Subtract the values.
    return $first - $second;
};

// Execute math function.
echo math($addition, 2, 2);
echo PHP_EOL; // New line!
echo math($subtraction, 5, 3);
```

The PHP_EOL is a constant that will be replaced with a newline character that's compatible with the current operating system. It's super handy!

We call the `math()` function twice with a different assigned Closure each time, providing integers, and separating the calls with a newline character using the `PHP_EOL` constant. Let's take a look at the output.

Example 08: Output.

```
4
2
```

Great! By passing a different Closure to out `math()` function, we've changed the way that it works. For homework, why don't you try changing the above code to include multiplication and division functionality? Here's a hint, you're going to want more Closures!

15. Includes

You've been learning PHP for a while now, it's time to talk about your future. You're in a committed relationship with PHP. You have matching his and hers dressing gowns. You might be thinking about marriage already. You're probably going to want to write future applications with PHP, aren't you?

Include

It would be a shame if you had to keep writing the same code over and over again, wouldn't it? I mean, it was fun to write, but it would slowly wear away at you. Let's imagine that you have the following function.

Example 01: A familiar function.

```
<?php

function welcome($name) {
    return "Welcome, {$name}!";
}
```

Ah! The old `welcome()` function. It never gets old! We're gonna need this in a lot of our applications. After all, if we don't welcome our users, they might get angry at us! We can use the `include` statement to make this code somewhat re-usable.

First let's put the function in it's own file. We'll call it `welcome.php`. We'll put the same code inside.

Example 02: Here it is again.

```
<?php

function welcome($name) {
    return "Welcome, {$name}!";
}
```

Now then, let's create a new file, called `program.php`. We'll place this file next to the `welcome.php` file. We want to use the `welcome()` function within this file, so let's `include` the `welcome.php` file. Confused? Let's see the `program.php` file.

Example 03: Including a file.

```
<?php

include 'welcome.php';

echo welcome('Tamas');
```

We use `include()` to include the `welcome.php` file. It's like PHP is copying and pasting the code in `welcome.php` to where we've put the `include` statement. We can now use the `welcome()` function like before.

The `include` statement is followed by the path to the file that we want to include. This path is relative to the file that you execute with the `php` command. If our `welcome.php` file was located in a subfolder, then we could use slashes `/` like we would in a unix filesystem to denote its location.

Example 04: Including within subdirectories.

```
<?php

include 'foo/bar/baz/welcome.php';
```

You can use as many includes as you like! Go ahead and modularise your code into re-usable files. You can also `include` from your included files. Kind of like that dream movie.

You can use the __DIR__ constant to refer to the directory that the current file is in. This can be useful when `include()`ing other files!

Require

If you try to `include` a file that doesn't exist, then PHP will show a warning message, and continue executing. If this is a problem in our applications, we can use the `require` statement instead of the `include` to have PHP throw a fatal error when the file cannot be loaded.

The syntax is exactly the same as the `include` statement. Let's load an imaginary file.

Example 05: Requiring a file.

```
<?php

require 'foo.php';
```

Let's see what PHP has to say about the imaginary `foo.php` file, shall we?

Example 06: Output.

```
Fatal error: require(): Failed opening required 'foo.php'
```

Great! That's going to put an end to that program. Come back when the file exists, you crazy loon!

Require Once

What would happen if we were to include the `welcome.php` file twice? You know, the one with the `welcome()` function inside? Let's find out, shall we?

It's going to look a little like this:

Example 07: Requiring a file twice.

```
<?php

require 'welcome.php';
require 'welcome.php';

echo welcome('Alastair');
```

Let's go ahead and execute the code.

Example 08: Output.

```
Fatal error: Cannot redeclare welcome() (previously declared...
```

Uh oh! What's going on here? The problem is that you can't define two functions with the same name. They collide, and you get this error. You can't re-declare the `welcome()` function.

So why do we have two functions anyway? Well we're requiring `welcome.php` twice. So we're loading the `welcome()` function twice. How do we get around this problem? What if we forget that we've already required the file, and then require it again much later in the application? This can happen easily if you're using files that include other files that include other files.

Hey, how about that subchapter title? That might help us, right? I mean, it normally does. Take a look at this snippet.

Example 09: Requiring a file only once.

```
<?php

require_once 'welcome.php';
require_once 'welcome.php';

echo welcome('Alastair');
```

All we've done here, is replaced the word `require` with `require_once`. What's the effect? Well let's execute our program and find out.

Example 10: Output.

```
Welcome, Alastair!
```

Hurray! It works. You see, the `require_once` statement will only include a file once. The second time the statement is executed, the file is ignored, and the `welcome()` function collision is avoided.

When including PHP code into your application, be sure to consider the effects of including the file twice. Decide whether `require` or `require_once` is more appropriate.

Well that's another short chapter read! Don't worry, the next one's going to be a fair bit longer, but extremely exciting!

16. Classes

Well, now we're really cooking with fire. We get to learn about classes!

Classes give our applications structure. They allow for object-oriented programming. You've probably heard of that term before? It means treating our code as objects, bringing familiarity to how we use objects in the real world. Let's stick with the baby steps, though, shall we?

First Class

So how do we create a class? Let's take a look at some code. First we're going to need a purpose for our class. Let's get started with object-oriented programming, and create a class that will represent a book. We'll call the class `Book`.

Example 01: Creating a class.

```
1  <?php
2
3  class Book
4  {
5
6  }
```

The syntax for a class, is fairly similar to that of a function. You may have already spotted the block surrounded by `{` curly `}` braces. On the first line we have the keyword `class` and then the name of the class that we want to create.

It's good practice to name your classes with an uppercase letter at the beginning. It's also good practice to have a separate file for your class, with a similar file name. For example, the class above would be contained in a file called `Book.php`.

For the purpose of simplifying the examples of the book. I'll be keeping classes in the same file for now.

We can't really do much with the class as it stands, let's give it some properties. Properties are used to store data within the class. For our book, they'll be used to further describe the book.

Let's think about the properties of a book for a moment. I can think of several.

- Title.
- Author.
- Publisher.
- Year of publishing.

Let's reflect these properties on our `Book` class. Here goes nothing!

Example 02: Class properties.

```php
<?php

class Book
{
    var $title;

    var $author;

    var $publisher;

    var $yearOfPublication;
}
```

Variables that belong to a class, are called 'class properties'. For now, we'll use the `var` keyword, along with the typical variable notation to define our class properties. Don't worry about the `var` keyword too much, we'll take a look at some alternatives later.

We're now ready to start working with our class.

Instances

The class that we've defined, is actually a blueprint. We can use it to make a new instance of a `Book` object. The key word there is **new**. An instance, is a class that we can use. We can have multiple instances of the same class. It's something that's really hard to explain in text, so let's look at a code sample.

Example 03: Creating an instance of a class.

```php
<?php

class Book
{
    var $title;

    var $author;

    var $publisher;

    var $yearOfPublication;
}

$book = new Book;
```

We create a **new** instance of the `Book` class, and assign it to the `$book` variable. We do this using the `new` keyword, and then the class name that we wish to instantiate. Now that we have an instance of class, we can play with its properties. Let's start by setting the properties for our book.

Example 04: Setting class properties.

```php
<?php

// Define the Book class.
class Book
{
    // Declare properties.
    var $title;
    var $author;
    var $publisher;
    var $yearOfPublication;
}

// Create a new book instance.
$book = new Book;

// Set properties.
$book->title               = 'Game of Thrones';
$book->author              = 'George R R Martin';
```

```
$book->publisher            = 'Voyager Books';
$book->yearOfPublication    = 1996;
```

You can set the properties of a class instance by using the object `->` operator, which consists of a dash `-` followed by a greater-than `>` symbol. Next, you specify the name of the property that you wish to access.

In the above example, we use the assignment `=` operator to set the class properties of our instance. We can also retrieve them using the object operator. Let's `echo` each of our class properties, appended with a newline character.

Example 05: Retrieving class properties.

```
<?php

// Define the Book class.
class Book
{
    // Declare properties.
    var $title;
    var $author;
    var $publisher;
    var $yearOfPublication;
}

// Create a new book instance.
$book = new Book;

// Set properties.
$book->title                = 'Game of Thrones';
$book->author               = 'George R R Martin';
$book->publisher            = 'Voyager Books';
$book->yearOfPublication    = 1996;

// Echo properties.
echo $book->title               . PHP_EOL;
echo $book->author              . PHP_EOL;
echo $book->publisher           . PHP_EOL;
echo $book->yearOfPublication   . PHP_EOL;
```

We once again use the object `->` operator to echo the properties of the book instance. Executing our code yields the following output.

Example 06: Output.

```
Game of Thrones
George R R Martin
Voyager Books
1996
```

Earlier, I mentioned that we can have multiple instances of classes, so lets define two books, each with different properties. I'm going to omit the class declaration for now, just to simplify the examples.

Example 07: Multiple class instances.

```php
<?php

// Define the Book class.
class Book
{
    // Declare properties.
    var $title;
    var $author;
    var $publisher;
    var $yearOfPublication;
}
// Book class definition would be here.

// Create a new book instance.
$first = new Book;

// Set properties.
$first->title               = 'Game of Thrones';
$first->author              = 'George R R Martin';
$first->publisher           = 'Voyager Books';
$first->yearOfPublication   = 1996;

// Create another book instance.
$second = new Book;

// Set properties.
$second->title               = 'The Colour Of Magic';
$second->author              = 'Terry Pratchett';
$second->publisher           = 'Colin Smythe';
$second->yearOfPublication   = 1983;
```

In the above example, we've created two book instances. They both have their own unique set of properties. They are individuals. What we've done, is created a complex storage mechanism for information. Right now, our classes don't have a huge advantage over using an array to store this information. Why don't we unlock some other secrets that classes have to offer?

Default Values

Our classes are blueprints for new instances of objects. Sometimes, the values that we expose may have defaults. For example, the majority of books will be in a paperback format. However, rarely, you may need to mark a book as hardback format.

Let's add a `format` property to our book class. We can save time for ourselves by giving the class property a default value of 'Paperback'. Let's examine the following code snippet.

Example 08: Class property with default value.

```
<?php

// Define the Book class.
class Book
{
    // Declare properties.
    var $title;
    var $author;
    var $publisher;
    var $yearOfPublication;
    var $format = 'Paperback';
}
```

We've assigned a default value to the `$format` property, by providing an assignment operator, followed by the default value. This line ends with a semicolon. Let's check to make sure that our `$format` property is set by default. We can do this by creating a new instance of a `Book`, and echoing the value stored in `$format`.

Example 09: Inspecting default property value.

```php
<?php

// Define the Book class.
class Book
{
    // Declare properties.
    var $title;
    var $author;
    var $publisher;
    var $yearOfPublication;
    var $format = 'Paperback';
}

// Create a new book instance.
$book = new Book;

// Echo the default value.
echo $book->format;
```

We've simply instantiated the book, and echoed the value. What response to we receive?

Example 10: Output.

```
Paperback
```

Just as expected! Of course, we're free to change this value if we please. It's simply a default to help simplify the process of creating books.

Example 11: Altering a property with a default value.

```php
<?php

// Define the Book class.
class Book
{
    // Declare properties.
    var $title;
    var $author;
    var $publisher;
```

```
    var $yearOfPublication;
    var $format = 'Paperback';
}

// Create a new book instance.
$book = new Book;

// Change the value of format.
$book->format = 'Hardback';

// Echo the value of format.
echo $book->format;
```

If we execute our code once more, we'll receive the following value.

Example 12: Output.

```
Hardback
```

Great! You can set your default values to all scalar types including `string`s, `int`s, `float`s and even `array`s with multiple values, both associative and indexed. There's one catch though, remember, there's always a little catch. When defining your default values, you can't set them to the results of functions. This is illegal. The PHP police will come and take you away if you try to do so.

It means that the following would not be possible.

Example 13: Illegal setting of default values.

```
<?php

// Create a book class.
class Book
{
    var $bar = strtoupper('foo');
}
```

Note that the `strtoupper()` function is an in-built PHP function to transform strings to their uppercase variants. So `foo` would become `FOO`. However, in this context, setting the defaults for a class property, it will result in a syntax error.

There's methods to the madness.

It's time to start using methods. Methods are what we call functions that belong to classes. Let's examine these wording differences one more, shall we?

General	Class
Variable	Property / Attribute
Function	Method

When you're looking for help, knowing the distinction between a function and a method will make it much more simple to communicate with other developers. It's worth taking the time now to learn these names.

How do we define a method on our class? Well it's simple. In fact, you already know the format! Let's create a simple function to say hello.

Example 14: Creating class methods.

```php
<?php

// Define the Book class.
class Book
{
    // Declare properties.
    var $title;
    var $author;
    var $publisher;

    // Declare a method.
    function sayHello()
    {
        return 'Hello!';
    }
}

// Create a new book instance.
$book = new Book;

// Change the value of format.
$book->format = 'Hardback';

// Execute the hello method.
echo $book->sayHello();
```

Once again, we use the `function` keyword to define a new function. The format is exactly the same to that which we learned in the previous chapter, except that the function now sits within the curly `{` braces `}` of our class.

We can execute our class function using the object `->` operator that we used to retrieve and set our class properties. We need to add the rounded `(` brackets `)` to the end of our method name, just as we did with functions.

For example:

Example 15: Calling class methods.

```
1  <?php
2
3  echo $book->sayHello();
```

We receive a friendly response from our class.

Example 16: Output.

```
1  Hello!
```

Great! We can include as many methods as we like within our class structures, so go ahead and make a few more methods to learn the syntax.

Can't touch $this

Okay, you actually can touch `$this`. In fact, you're going to need to.

When writing functions for your classes, you're likely going to want to access your class properties, or even call other class methods. You can do this by using the `$this` property.

The `$this` property is a shortcut to reference the *current* class instance. You can use it to access class properties and other methods from within a class method. Let's take a look at this in action.

Example 17: Indicating current instance.

```php
<?php

// Define the Book class.
class Book
{
    // Declare properties.
    var $title;
    var $author;
    var $publisher;

    // Declare a method.
    function summary()
    {
        echo 'Title: '      . $this->title      . PHP_EOL;
        echo 'Author: '     . $this->author     . PHP_EOL;
        echo 'Publisher: '  . $this->publisher  . PHP_EOL;
    }
}

// Create a new book instance.
$book = new Book;

// Set class properties.
$book->title        = 'Reaper Man';
$book->author       = 'Terry Pratchett';
$book->publisher    = 'Victor Gollancz';

// Output a book summary.
$book->summary();
```

That's a big chunk of code, isn't it? Let's break it up a little. First we'll take a look at the `summary()` method.

Example 18: Inspecting properties for the current instance.

```
<?php

// Declare a method.
function summary()
{
    echo 'Title: '      . $this->title      . PHP_EOL;
    echo 'Author: '     . $this->author     . PHP_EOL;
    echo 'Publisher: '  . $this->publisher  . PHP_EOL;
}
```

Within the summary method, we output a string label for each field. Next, we concatenate the field value, by using `$this` to refer to the current instance, and then the object `->` operator, and finally the property. Finally, we append a PHP newline character for clarity.

Example 19: Excuting the summary method.

```
<?php

// Create a new book instance.
$book = new Book;

// Set class properties.
$book->title        = 'Reaper Man';
$book->author       = 'Terry Pratchett';
$book->publisher    = 'Victor Gollancz';

// Output a book summary.
$book->summary();
```

Next, we create a new `Book` instance, and populate the fields present. Finally, the `summary()` method is called on the instance. Let's take a look at the output that we receive from executing this code.

Example 20: Output.

```
Title: Reaper Man
Author: Terry Pratchett
Publisher: Victor Gollancz
```

We receive a clean representation of our book. As you can see, the properties were accessed correctly using the `$this` property.

We can also use `$this` to call instance methods. Let's take a look at an example, shall we?

Example 21: Calling class methods from class methods.

```php
<?php

// Define the Example class.
class Example
{
    // First function.
    function first()
    {
        return $this->second();
    }

    // Second function.
    function second()
    {
        return $this->third();
    }

    // Third function.
    function third()
    {
        return "Well that was rather pointless, wasn't it?";
    }
}
```

Here, we can see a chain of methods.

- The `first()` method uses `$this` to return the value of the `second()` method.
- The `second()` method uses `$this` to return the value of the `third()` method.

- Finally, the `third()` method returns a string value.

It's not a very practical example in the real world, but it does help to illustrate that function and method calls can be nested in a **nearly** endless fashion.

Nearly endless, within limitations of memory, and the depth of the PHP call stack. It's quite a large number by default though, you normally won't run into any problems unless you make a mistake with a loop.

Constructors

Constructors are special methods. They are called automatically when you create a new instance of the class. This gives you a great opportunity to set your class properties to values that you might otherwise be unable to set. For example, the results of a function or method.

To define a constructor, we need only create a method with the name `__construct()`. It's worth noting that there are two _ underscores at the beginning of the function name. Unfortunately, while PHP is an extremely flexible language, it often falls short when it comes to descriptive or convenient function names, and `__construct()` is no exception. Don't worry! It will soon become second nature.

Example 22: Creating a class constructor.

```
<?php

// Define the Panda class.
class Panda
{
    // LOUD NAME PROPERTY
    var $loudName;

    // Contructor method.
    function __construct()
    {
        $this->loudName = $this->makeNameLoud('Lushui');
    }

    // NAME LOUDENING METHOD
    function makeNameLoud($name)
    {
```

```
        return strtoupper($name);
    }
}
```

Here, we have a class with a method called `makeNameLoud()`, which performs a case transformation on a string. We've already discovered that we can't set default values for class properties using functions or methods directly, so instead, we use the `__construct()` method to set the initial value.

How can we check if the constructor has been executed on instantiation? Oh that's right, we just need to check the `$loudName` property! Let's try it out!

Example 23: Inspect class property.

```
<?php

// Create a new class instance.
$panda = new Panda;

// Output loud name.
echo $panda->loudName;
```

We instantiate the `Panda` class, and echo the value of the `loudName` property. Let's check the value that we receive.

Example 24: Output.

```
LUSHUI
```

Ouch, that's a loud panda for sure. Since the name is in uppercase, it's clear that our class constructor has been executed.

Let's try to make that constructor a little more dynamic. Constructors are methods, and so they take parameters. Let's give the constructor a `$name` parameter to be able to set the loud name of a panda upon instantiation. Here's how our class looks.

Example 25: Constructors with parameters.

```php
<?php

// Define the Panda class.
class Panda
{
    // LOUD NAME PROPERTY
    var $loudName;

    // Contructor method.
    function __construct($name)
    {
        $this->loudName = $this->makeNameLoud($name);
    }

    // NAME LOUDENING METHOD
    function makeNameLoud($name)
    {
        return strtoupper($name);
    }
}
```

All that we've changed in the above example, is the addition of a `$name` parameter in the constructor, which is passed to the `makeNameLoud()` method. Let's take a look at the new usage of this class, shall we?

Example 26: Using constructors with parameters.

```php
<?php

// Create a new class instance.
$panda = new Panda('Lushui');

// Output loud name.
echo $panda->loudName . PHP_EOL;

// Create a new class instance.
$secondPanda = new Panda('Pali');

// Output loud name.
echo $secondPanda->loudName;
```

In the above example, we instantiate two different instances of the `Panda` class. When we instantiate the class, we pass a string for the panda name into a pair of rounded `(` brackets `)` just as we would using a normal function.

We echo the values for `loudName` for each of the panda instances and receive...

Example 27: Output.

```
1 LUSHUI
2 PALI
```

...in response.

You can even keep the brackets if you aren't providing a parameter. For example, for a class with no parameters to its constructor, then the following is perfectly legal.

Example 28: Instantiating class with no parameters, but brackets.

```
1 <?php
2
3 $panda = new Panda();
```

However, I personally like to leave them out to increase clarity. It's up to you!

Note that since a constructor is called during the instantiation of a new object, it's a bad idea to use a `return` statement within. In fact, if you try it, PHP will shout at you. Actually, if you try it, I'll shout at you, and I'm generally a nice guy.

17. Inheritance

When a mommy panda and a daddy panda love each other very much, they might decide to have baby pandas together. Of course, I'm not talking about those lazy black and white pandas. They aren't very parentally motivated. I mean the beautiful red ones.

The baby pandas will likely share a number of properties with their parents. Not only their bright fluffy coat, but also their eating habits, sleeping patterns, and general personality. They *inherit* these properties from their parents.

> If I wanted a biology lesson, I'd have bought a biology book.

Well look at you! Aren't you a little antsy today?

Well, as it happens, pandas aren't the only things that inherit properties. You see, classes can also have parents. I'm not entirely sure how the breeding takes place, but I can certainly teach you about the result.

Let's take a look at a simple class.

Example 01: A simple class!.

```php
<?php

class Panda
{
    // Properties
    var $coat = 'fluffy';
    var $colour;

    // Method
    function getCoat()
    {
        return $this->coat;
    }

    // Method
    function getColour()
    {
        return $this->colour;
```

```
19      }
20  }
```

Here's our simple class. It's got two properties, `$coat` and `$color`. We've also got class methods called `getCoat()` and `getColour()`, which simply return our property values. It doesn't get more simple, does it? You've got this buddy!

Let's push our comfort zones. Let's do something new.

Example 02: Extending a class.

```
1   <?php
2
3   class Panda
4   {
5       // Properties
6       var $coat = 'fluffy';
7       var $colour;
8
9       // Method
10      function getCoat()
11      {
12          return $this->coat;
13      }
14
15      // Method
16      function getColour()
17      {
18          return $this->colour;
19      }
20  }
21
22  class GiantPanda extends Panda
23  {
24      // DUN DUN DUUUUUUN!
25  }
```

Hurray! There's some new syntax. That means that we're about to add a new tool to our arsenal!

What's this `extends` keyword doing? We're actually telling PHP that the `GiantPanda` class should inherit from the `Panda` class. You could say that the `GiantPanda` class is going to borrow genetic material from the `Panda`. This means that `GiantPanda` gains access to all of the properties and methods from `Panda`.

> Prove it!

Oh yeah? Don't believe me? I'm going to prove it so darn hard! In the next section, just imagine that the class definitions are still there. I'm gonna omit them for clarity. I don't want to be one of those cheesy authors who duplicate a bunch of code just to get more pages in! We're friends now! I'd never do that to you. Now don't you feel guilty for not trusting me?

Example 03: Using a child class.

```
 <?php

// Create a new giant panda instance.
$giantPanda = new GiantPanda;

// Get coat type.
echo $giantPanda->getCoat();
```

We create a new instance of the `GiantPanda` class and call the `getCoat()` method on it. We echo out the value, because we're going to want to see the result. Let's take a look at the output.

Example 04: Output.

```
fluffy
```

Awww. It's so fluffy I'm gonna die!

As you can see, our giant panda has inherited both the `getCoat()` method, and the `$coat` property (with default value) from its panda parent.

While red pandas and giant pandas are similar, the coats of the giant pandas are less fluffy, so we need to represent this in our `GiantPanda` class. Let's try it!

Example 05: Overriding properties.

```
 <?php

class Panda
{
    // Properties
    var $coat = 'fluffy';
    var $colour;

```

```
    // Method
    function getCoat()
    {
        return $this->coat;
    }

    // Method
    function getColour()
    {
        return $this->colour;
    }
}

class GiantPanda extends Panda
{
    var $coat = 'less fluffy';
}
```

In the above example, we have overridden the `$coat` property from the `Panda` class. Within the `GiantPanda` class, we redeclare `$coat` as 'less fluffy'. We haven't changed the `getCoat()` method at all. Let's try calling this method now.

Example 06: Accessing overriden properties.

```
 <?php

// Create a new giant panda instance.
$giantPanda = new GiantPanda;

// Get coat type.
echo $giantPanda->getCoat();
```

This time we receive new output...

Example 07: Output.

```
less fluffy
```

As you can see, the `getCoat()` method in the parent `Panda` class respects our change to the `$coat` variable in the child `GiantPanda` class. It returns the value `less fluffy`. We've overridden the value of this field.

We can also override class methods from parent classes. All you need to do is re-declare the property or method that you want to override in the child class.

If it's not clear yet, the '*parent*' class is the class that you `extend`. In the above example, the parent class is `Panda`. The '*child*' class is the one that implements the `extend` keyword. This was `GiantPanda` in the previous example.

Classes can have a limitless chain of inheritance. Here's an example of four classes which are part of an inheritance chain.

Example 08: Inheritance chain.

```
 <?php

class First
{
    var $legendary = 'Barney Stinson';
}

class Second extends First
{

}

class Third extends Second
{

}

class Fourth extends Third
{

}
```

In the above example:

- **Second** inherits from *First*
- **Third** inherits from *Second*
- **Fourth** inherits from *Third*

I like to call this an inheritance chain. It looks like this:

`First <– Second <– Third <– Fourth`

Of course, all four classes have access to a `$legendary` property, with the value of 'Barney Stinson'.

Type-hinting classes

In a previous chapter, we discovered how to type hint function parameters so that we can ensure that they are of the correct type. We demonstrated this technique using arrays, but they aren't the only types of variables that can be type-hinted.

In PHP functions and methods, we can type-hint classes. With a parameter type-hinted to a class, only instances of the specified class can be passed to the method or function.

Let's demonstrate this functionality by first creating two classes.

Example 09: Type-hinting a class.

```
1  <?php
2
3  class Sephiroth
4  {
5      function equipWeapon(Weapon $weapon)
6      {
7          echo 'Die cloud die!';
8      }
9  }
10
11 class Weapon
12 {
13
14 }
```

Our first class, `Sephiroth` has a single method called `equipWeapon()`, which has a parameter type-hinted to the second class named `Weapon`. (Don't worry about the functionality of the `equipWeapon()` method for now, we're merely thinking about structure at this point.) This means that you will only be able to pass instances of the `Weapon` class as parameters for this method.

Let's attempt this.

Example 10: Passing a class instance as parameter.

```php
<?php

// Instantiate a Sephiroth class.
$sephiroth = new Sephiroth;

// Instantiate a Weapon class.
$weapon = new Weapon;

// Call the equipWeapon() method.
$sephiroth->equipWeapon($weapon);
```

If we execute the code, we see the 'Die cloud die!' message that we expect. Go ahead and try changing the type of the parameter passed to the `equipWeapon()` method. I can assure you, PHP will not be pleased.

Earlier, I told you that only instances of the type-hinted class would be allowed, but actually, I lied a little. You see, you can also pass classes which **inherit** from the type-hinted class. Let's see this in action.

Example 11: Passing child classes to parent type-hinted method.

```php
 <?php

class Sephiroth
{
    function equipWeapon(Weapon $weapon)
    {
        echo 'Die cloud die!';
    }
}

class Weapon
{

}

class Masamune extends Weapon
{

}

```

```
21 class Murasame extends Weapon
22 {
23
24 }
25
26 // Instantiate a Sephiroth class.
27 $sephiroth = new Sephiroth;
28
29 // Instantiate a Masamune class.
30 $masamune = new Masamune;
31
32 // Instantiate a Murasame class.
33 $murasame = new Murasame;
34
35 // Call the equipWeapon() method with masamune.
36 $sephiroth->equipWeapon($masamune);
37
38 // Call the equipWeapon() method with murasame.
39 $sephiroth->equipWeapon($murasame);
```

We've added added two new classes, `Masamune` and `Murasame`, which extend the `Weapon` parent class. Since they both have `Weapon` as a parent class, we can pass their instances to methods which type-hint `Weapon`.

In a later chapter, we're going to take a look at a special structure called an `interface` and you'll see how type-hinting combined with interfaces can lead to some extremely useful tricks!

18. Scope

We've all got secrets, haven't we? Mine is that I've stolen countless red pandas from Cardiff Zoo. I've got them all stashed away in my basement, where I'm training them for a third world war. Actually, just pretend you didn't hear all that.

If you've got something that you don't want people messing with, you're going to want to keep it **private**. Some things are meant to be shared, and will be **public**ly available. Sometimes, you're willing to share your secrets, but you'll want to **protect** them so that they can't damage you.

Your code is no different. Others may want to use your class, and that's just fine, but you don't want them fiddling with things that might break its functionality. We can help others to use our code in a sensible manner by defining scope on our methods and properties.

Public

Up until now, we've been using `var` to declare class properties, and only `function` to create our class methods. Declaring variables in this way makes them public.

A public property can be accessed from outside a class instance by using the object `->` operator. For example:

Example 01: Access a class attribute.

```
1  <?php
2
3  $panda = new Panda;
4  echo $panda->name;
```

Public methods can be called on an instance using the object `->` operator once more.

Example 02: Access a class method.

```
<?php

$panda = new Panda;
echo $panda->squeak();
```

Since our properties and methods so far have been public, why don't we start indicating their scope with a more descriptive keyword?

The truth is, most programmers won't use `var` to declare their class properties. Instead, it's better to use the `public` keyword. Let's take a look at an example, shall we?

Example 03: A public property.

```
<?php

class Panda
{
    public $coat = 'fluffy';
}
```

This is how you're going to declare your public class properties from now on, isn't it? Go on, promise me! It's just a better way of doing it!

How about those public methods? Well we can apply the `public` keyword to those too, and you definitely should. Let's take a look at a quick example.

Example 04: A public method.

```
<?php

class Panda
{
    public $coat = 'fluffy';

    public function getCoat()
    {
        return $this->coat;
    }
}
```

We've put the keyword `public` just before the keyword `function`. It won't make any difference, but it's now clearer that the function is a public one.

So we haven't really learned any new tricks in this section, have we? At least we found a way to make our code more descriptive. Don't worry! In the next section we'll be learning something new. I promise!

Private

In the previous section, we used the `public` keyword to mark our class properties and methods as public. Let's try marking a class property as `private` instead, shall we?

We'll start by modifying our previous example.

Example 05: A private property.

```
<?php

class Panda
{
    // Declare a private property...
    private $coat = 'fluffy';

    // ... and a public method.
    public function getCoat()
    {
        return $this->coat;
    }
}
```

In the above example, you'll notice that the `$coat` property has now been marked as private by using the `private` keyword. For now, we'll leave the `getCoat()` method as public.

Let's try to play with that private `$coat` property. We'll just try to echo it out.

Example 06: Access a private property.

```
<?php

// Create a new panda.
$panda = new Panda;

// Try to echo the coat property.
echo $panda->coat;
```

What do you think will happen when we execute the above piece of code? Well let's find out already!

Example 07: Output.

```
Fatal error: Cannot access private property Panda::$coat
```

Because our property is now marked as `private`, we can't access it from outside the class. We'll receive a fatal error if we decide to do so. We've protected that `$coat` property from our grubby little hands!

You can't set the property either! Let's give it a go.

Example 08: Setting a private property.

```
<?php

// Create a new panda.
$panda = new Panda;

// Try to set the coat property.
$panda->coat = 'slimey';
```

Let's execute the code once again.

Example 09: Output.

```
Fatal error: Cannot access private property Panda::$coat
```

We receive the same error message as when we tried to retrieve the value of the property.

Let's try accessing that public method. I'll attach the class to the example once again to refresh your memory.

Example 10: Access public method.

```php
<?php

class Panda
{
    // Declare a private property...
    private $coat = 'fluffy';

    // ... and a public method.
    public function getCoat()
    {
        return $this->coat;
    }
}

// Create a new panda.
$panda = new Panda;

// Try to echo the coat property.
echo $panda->getCoat();
```

The method is public, so when we try to call it, we receive the following result.

Example 11: Output.

```
fluffy
```

If the `$coat` property is private, then how is the `getCoat()` method able to access it? Well you see, the scope of a method or property, only applies to retrieving, setting, or calling it from outside of the class. Class methods have access to all of the other properties and other methods, regardless of the scope.

Let's change the `getCoat()` method to `private`.

Example 12: Access a private method.

```
<?php

class Panda
{
    // Declare a private property...
    private $coat = 'fluffy';

    // ... and a private method.
    private function getCoat()
    {
        return $this->coat;
    }
}

// Create a new panda.
$panda = new Panda;

// Try to echo the coat property.
echo $panda->getCoat();
```

Let's execute the code once more.

Example 13: Output.

```
Fatal error: Call to private method Panda::getCoat()
```

Since our method has now been given `private` scope, we are unable to call it on our method instance.

Let's add a public method alongside our private one.

Example 14: Access a public proxy method.

```
<?php

class Panda
{
    // Declare a private property...
    private $coat = 'fluffy';

    // ... and a private method...
```

```
    private function getCoat()
    {
        return $this->coat;
    }

    // ... and a public one.
    public function noReallyGetCoat()
    {
        return $this->getCoat();
    }
}

// Create a new panda.
$panda = new Panda;

// Try to echo the coat property.
echo $panda->noReallyGetCoat();
```

We've added a new method called `noReallyGetCoat()`, which makes an internal call to the `getCoat()` method. Let's see what happens when that method is executed.

Example 15: Output.

```
fluffy
```

As we discovered earlier, scope does not affect internal calls to methods, or accessing properties from within the class. For this reason, the public `noReallyGetCoat()` method is able to make a call to the private `getCoat()` method.

You can make a class constructor private, but then you won't be able to instantiate it. This might be useful later when you learn more about the `static` keyword.

Protected

Protected properties are a little different. From the outset they look to function exactly like properties within the `private` scope. If you were to try and access them directly, you would be unable to.

So why do we have two types of scope that are the same? Well, this is one of those situations where being an author is really difficult. I'd like to teach you about the

protected scope now, however, it's hugely related to a future chapter on abstract classes. Difficult, right? Let's instead share a light overview of protected, and we'll examine it in detail later.

In a later chapter, you'll learn about `abstract` classes. Ones which can be extended by other classes so that they can share common functionality. If you have a `private` property or method on the abstract class or "base class" (the base template), then any classes that extend the class are unable to modify the methods or properties directly. However, if you were to instead define these methods and properties as `protected`, then you'd be able to overwrite their purpose within your extending classes.

Does this sound complicated? Don't worry! It will become much clearer in the Abstract chapter which will arrive later.

Static

There's another scope for variables and methods called `static`, but this one is a little more complicated, and is best saved for an entire chapter of its own.

19. Constants

In a previous chapter, we discovered variables, and how they could be used to store a variety of different types of information. You could re-assign a variable to a new value, and PHP wouldn't give two hoots about it!

Sometimes, things aren't meant to change. Old people will always prefer to wear beige. The government will continually attempt to throw your money away. Red pandas will consistently become excitable in periods of cold weather.

Defined Constants

There are values in your code that might not want to change too. For example, let's use that wibbly-wobbly time fading technique they use on TV shows to go back in time and take another look at the assignment of the `$pi` variable.

Example 01: A PI variable.

```
1  $pi = 3.14159265359;
```

There she is. Beautiful.

Do you think that the value for PI is ever going to change? I mean sure, it can always be calculated to additional decimal places by clever white coat wearing boffins, but is it likely to need to change within our program?

I doubt it. In fact, if someone was to do this...

Example 02: Set PI to 5.

```
1  $pi = 5;
```

... then our circumference calculations would be in big trouble. We'd have our middle school Math professor frowning at us, and we don't want that.

It would be great if we could *lock* our `$pi` variable to a value, so that it couldn't be changed at all. Well, as it turns out, a variable isn't what we need at all. We need something called a constant.

Constants used outside of classes (I like to call this the `Global` scope.) are defined using a special function. Let's take a look.

Example 03: Define a PI constant.

```
1  define('PI', 3.14159265359);
```

We use the `define()` function to set our PI constant to the correct value. As you can see, the constant name is the first parameter to the function, and its value is the second.

You'll notice that the constant name is written in uppercase. This isn't only true for the PI constant. A convention for programmers is to write constants in uppercase, and use underscores for spaces. THIS_IS_A_CONSTANT.

Let's try to use our constant. It's very similar to a variable, only that it doesn't begin with a `$` dollar sign. Let's attempt to echo our constant.

Example 04: Use the PI constant.

```
1  // Define our constant.
2  define('PI', 3.14159265359);
3
4  // Echo the value of PI.
5  echo PI;
```

Executing our snippet of code yields the result:

Example 05: Output.

```
1  3.14159265359
```

Great! This is exactly what we'd get using a variable. Let's now try to re-define the constant. For example:

Example 06: Attempt to re-define a constant.

```
1  // Define our constant.
2  define('PI', 3.14159265359);
3
4  // Re-define our constant.
5  define('PI', 42);
```

What do you think will happen when we try to execute our code? Let's give it a go.

Example 07: Output.

```
Notice: Constant PI already defined in <file> on line <number>
```

A notice! It's telling us that we've already defined PI, and that we can't change its value. The notice won't break our code, but it will protect us from accidentally changing the value of the constant. Our circumference calculations are safe.

The naming limitations of constants are identical to those of variables, and you can define as many constants as you need. Constants are best described as 'immutable', and will not be changed.

Class Constants

The constants defined in the previous section are considered 'global', and are accessible by all parts of our application. Sometimes, we only need constants that are to be used by our classes.

For example, let's make a simple circle class.

Example 08: The circle class.

```php
<?php

class Circle
{
    /**
     * The value of PI.
     *
     * @var float
     */
    private $pi = 3.14159265359;

    /**
     * Calculate the circumference of a circle from diameter.
     *
     * @param  mixed $diameter
     * @return mixed
     */
    public function circumference($diameter)
    {
        return $diameter * $this->pi;
    }
}
```

Our circle class has a `private` property called `$pi`, which contains the appropriate float value. It also has a `public` method called `circumference()`, which takes a diameter of the circle as a parameter to calculate its circumference.

Let's test this class.

Example 09: Using the Circle class.

```
$circle = new Circle;

echo $circle->circumference(32);
```

Executing the above code snippet yields us the value `100.53096491488`, which is not only my jeans size, but also the correct circumference for our circle. However, while we know that the `private` variable cannot be changed outside the class, it can still be changed from within.

Example 10: Alter the PI property.

```
<?php

class Circle
{
    /**
     * The value of PI.
     *
     * @var float
     */
    private $pi = 3.14159265359;

    /**
     * Calculate the circumference of a circle from diameter.
     *
     * @param  mixed $diameter
     * @return mixed
     */
    public function circumference($diameter)
    {
        $this->pi = 48;
        return $diameter * $this->pi;
    }
}
```

This time our value would be incorrect, due to the re-assignment of the `$pi` class property. It may seem an unlikely change, but within a large class with many functions, the risk of change increases dramatically.

We like PI. Especially cherry pie. We don't want it to change. Let's make it a class constant instead.

Within classes, constants can be defined using the `const` keyword. Let's take a look at an example.

Example 11: PI as a class constant.

```
<?php

class Circle
{
    /**
     * The value of PI.
     */
    const PI = 3.14159265359;

    /**
     * Calculate the circumference of a circle from diameter.
     *
     * @param  mixed $diameter
     * @return mixed
     */
    public function circumference($diameter)
    {
        return $diameter * self::PI;
    }
}
```

We've used the `const` keyword to define our PI class constant, and have given it a value.

Example 12: Defining a class constant.

```
const PI = 3.14159265359;
```

The value of PI is now immutable, and PHP will throw an error if you attempt to change it. Now, you can't use `$this` to access a class constant, and I'll explain why shortly. For now, you'll need to trust me, and instead use the `self` keyword, followed by the double `::` colon operator, and finally the constant name. Like this:

Example 13: Accessing a class constant.

```
return $diameter * self::PI;
```

Simple! Now our value for PI can be used, but will never change. But what's this `self::` stuff all about?

Well, you may have noticed that the constant we created did not include any scope (public/private/protected/static). Our constant cannot be changed, and because of this, its value will be the same across all instances of our `Circle` class.

For this reason, constants are not accessed in the same manner as class properties. In fact, you don't even need a class instance to access the constant. Confused? Take a look at the following piece of code, which makes use of our `Circle` class.

Example 14: Accessing a constant on the class definition.

```
echo Circle::PI;
```

If we execute the snippet above, we'll receive the correct value for PI. We don't need a class instance. Using the double `::` colon operator, we can access the PI constant directly on the class definition. This means that class constants can be considered somewhat public, since their values can be accessed (but never modified!) from outside of a class instance.

The double `::` colon operator is actually known as the Scope Resolution Operator, or even the Paamayim Nekudotayim. Both of these sound terrifying, so let's stick with double colon for now.

It's worth noting that we can access the constant in a similar fashion from a class instance. For example, the following code is entirely valid, and yields an identical result to the previous example.

Example 15: Access the constant on an instance.

```
$circle = new Circle;

echo $circle::PI;
```

Within the class, `self` is used as a reference to the current class definition. On the following line...

Example 16: The self reference.

```
1 return $diameter * self::PI;
```

... we could have also accessed the PI constant using the class definition itself...

Example 17: Accessing through class definition.

```
1 return $diameter * Circle::PI;
```

... and the code would execute as expected.

However, if we later decide to rename the `Circle` class, we'd also have to replace all constant accessors within the class to match. For this reason, using `self` to address the current class definition is far more convenient.

When defining variables, always take a moment to think about their purpose, and whether their value is needed/likely to change. Use constants where appropriate to protect the functionality of your code.

20. Abstracts

In a previous chapter, we learned how we could use inheritance to share common properties and methods between our classes, effectively creating a template. We're going to build on our previous knowledge within this chapter. Excited? Then let's jump right in.

Abstract Classes

For some classes, their sole purpose is to provide common properties and methods. They live for it. When they go to sleep at night, they dream of it.

They'll never be instantiated, because they make no sense on their own. Consider the following classes, that I've left empty for clarity.

Example 01: Class inheritance.

```php
<?php

class Animal {}

class Panda extends Animal {}

class Owl extends Animal {}

class Giraffe extends Animal {}
```

While the Animal class might provide lots of shared features to the Panda, Owl and Giraffe classes, it doesn't make sense to instantiate it directly. Our Animal class is a prime example of an `abstract` class. Let's go ahead and make it one.

Example 02: Abstract class Animal.

```php
<?php

abstract class Animal
{
    /**
     * Is the animal awesome (all are).
     *
     * @var boolean
     */
    private $awesome = true;

    /**
     * Access the awesome attribute.
     *
     * @return boolean
     */
    public function isAwesome()
    {
        return $this->awesome;
    }
}
```

In the above example, we've declared our class as abstract, simply by starting the class definition with the keyword `abstract`. Everything else should feel familiar. If we were to extend this class, all of our animals would inherit the `isAwesome()` method.

Let's try to instantiate the `Animal` class directly.

Example 03: Create a new Animal instance.

```php
$animal = new Animal;
```

What happens when we execute the code?

Example 04: Output.

```
Fatal error: Cannot instantiate abstract class Animal in <file> on line <line>.
```

Now that our class is marked as abstract, it can no longer be instantiated. This is desirable, since the class has only a single purpose, to provide functionality to other classes.

Abstract Methods

Abstract methods are used within abstract classes. Once again, the keyword `abstract` is used to define an abstract method. Let's lead with an example.

Example 05: An abstract method.

```php
<?php

abstract class Animal
{
    abstract public function makeNoise();
}
```

Here, we have defined a new method called `makeNoise()` using the `abstract` keyword before the scope. You'll notice that the `makeNoise()` method has no body, instead it's simply terminated with a semi ; colon.

> What a pointless method!

You're quite right. On it's own, it is quite pointless. In fact, an abstract method can't even be called. So why bother with it? Well you see, the abstract method forms a type of 'contract' with all classes that extend it. It's telling PHP "Any classes which extend this one, ***MUST*** define a `makeNoise()` method.".

Let's demonstrate this by adding another class.

Example 06: Extend the Animal class.

```php
<?php

abstract class Animal
{
    abstract public function makeNoise();
}

class Dog extends Animal
{

}
```

We've extended the base `Animal` class within our `Dog` class. Let's try to instantiate a new `Dog`, shall we?

Example 07: Create a new dog instance.

```
$dog = new Dog;
```

Now we'll simply execute the code and... oh no..

Example 08: Output.

```
PHP Fatal error:  Class Dog contains 1 abstract method and must therefore be dec\
lared abstract or implement the remaining methods (Animal::makeNoise) in <file> \
on <line>.
```

Fortunately, I've always found PHP errors to be quite easy to understand. PHP is telling us that our `Dog` class contains an abstract method from the Animal base class called `Animal::makeNoise`.

PHP has given us two options. Firstly, we could make the `Dog` class an `abstract` class too, which would mean that we couldn't instantiate it, but it could be extended. Our second, and more appropriate option is to implement the `makeNoise()` method on the `Dog` class.

Using the second option, would fulfil the 'contract' that our abstract class defines. Let's do that.

Example 09: Implement the makeNoise() method.

```
<?php

class Dog extends Animal
{
    public function makeNoise()
    {
        echo 'Woof!';
    }
}
```

As you can see, we don't put the `abstract` keyword in this time. This is because we're implementing the method, and we want it to be callable. When implementing an abstract method, it's important to note that the method signature (scope, name and parameters) must be identical to that of the abstract method.

Now that we've implemented our `makeNoise()` method, our `Dog` class can finally be instantiated.

Example 10: Create another dog.

```
1 $dog = new Dog;
```

> What's the point of defining the abstract method, if we're just going to write the full method in all of our classes?

I thought you might ask this question. Let's consider a scenario. We change our `Animal` base class and remove the abstract method.

Example 11: Remove abstract method.

```
1 <?php
2
3 abstract class Animal
4 {
5
6 }
```

We can still define `makeNoise()` on all of our animal classes. However, what if we don't implement it on one of them. When we attempt to call the `makeNoise()` method, our code is going to break.

Using the abstract method to build a contract with the classes that extend our abstract class, means that we have built some trust. We can use the `makeNoise()` method without checking that it first exists. We have confidence that all animals will have this method.

In the next chapter, we'll take a look at a technique called `polymorphism`, and while it sounds scary, I promise that you'll find it quite useful.

Remember that abstract classes can contain both normal and abstract methods. This is what makes them different from the interfaces that you'll discover in the next chapter.

21. Interfaces

In the previous chapter, we learned about abstract classes, and how they form a partial contract. This is because abstract classes can contain both abstract methods **and** real methods. For example...

Example 01: An abstract class.

```php
<?php

abstract class Animal
{
    public function huggle()
    {
        echo 'I am huggled!';
    }

    abstract public function makeNoise();
}
```

Interfaces are very similar to abstract classes, except that they cannot contain logic. They are purely to shape the classes that implement them. This means that the `huggle()` method above, would not be possible within an interface.

Let's define a new interface.

Example 02: An interface.

```php
<?php

interface PandaInterface
{
    /**
     * Eat some food!
     */
    public function eat($food);

    /**
     * Poop! (Normally after eating.)
     */
```

```
13      public function poop();
14
15      /**
16       * Sleepy time!
17       */
18      public function sleep($time);
19  }
```

We define interfaces by using the `interface` keyword. As you can see, we don't use `class` at all this time. Like abstract classes, interfaces cannot be instantiated.

We've called our interface `PandaInterface`, and while not required, it's considered good practice to add the 'Interface' suffix to the name.

Our `PandaInterface` defines a number of methods that are common to pandas. All pandas eat, sleep and poop. They may do so differently, but they will definitely take part in these activities.

This means that any class which implements this interface **must** provide these methods, and the logic for them. Let's take a look at how we can implement the panda interface in our new class, the 'RedPanda'.

Example 03: Implement an interface.

```
1  <?php
2
3  class RedPanda implements PandaInterface
4  {
5
6  }
```

Instead of the `extends` keyword that we might use to indicate inheritance, this time we use the `implements` keyword to indicate that our class must implement the methods within the interface.

Earlier, we discovered that a PHP class can only extend one class. This same is not true for interfaces. A PHP class may implement as many interfaces as necessary. For example, the following is perfectly legal.

Example 04: Implement multiple interfaces.

```
<?php

class RedPanda implements PandaInterface, FurryInterface, CuteInterface
{

}
```

The class above, implements three different interfaces, and ***must*** implement the methods held within each of the interfaces.

A class can even extend another class while implementing interfaces, for example:

Example 05: Extend and implement.

```
class RedPanda extends Animal implements PandaInterface, FurryInterface, CuteInt\
erface
{

}
```

In the above example, either `RedPanda` or its parent class `Animal` must implement the methods held within the interfaces. As long as all are present, with each method on either of the classes, then the `RedPanda` class can be instantiated.

It took me a while to understand interfaces, and their ideal use case, but once I'd worked it out, I couldn't stop using them! Simply put, interfaces create contracts so that you can trust that a class has the methods you want inside.

Polymorphism

Polymorphism is a horrible word, but a wonderful technique. All of the explanations I found online were overcomplicated and scary. Instead, let's lead with an example. We'll make use of our `PandaInterface`. It enforces the implementation of three methods.

Example 06: The PandaInterface.

```
<?php

interface PandaInterface
{
    /**
     * Eat some food!
     */
    public function eat();

    /**
     * Poop! (Normally after eating.)
     */
    public function poop();

    /**
     * Sleepy time!
     */
    public function sleep();
}
```

I've stripped out some of the parameters to simplify the example. Let's create two classes that implement this interface. Here is the first.

Example 07: The RedPanda class.

```
<?php

class RedPanda implements PandaInterface
{
    /**
     * Eat some food!
     */
    public function eat()
    {
        echo "The red panda eats some fruit.\n";
    }

    /**
     * Poop! (Normally after eating.)
     */
    public function poop()
```

```
17      {
18          echo "The red panda takes a poop.\n";
19      }
20
21      /**
22       * Sleepy time!
23       */
24      public function sleep()
25      {
26          echo "The red panda sleeps up a tree.\n";
27      }
28  }
```

Next, we have a second class to represent a Giant Panda.

Example 08: The GiantPanda class.

```
1   <?php
2
3   class GiantPanda implements PandaInterface
4   {
5       /**
6        * Eat some food!
7        */
8       public function eat()
9       {
10          echo "The giant panda eats some bamboo.\n";
11      }
12
13      /**
14       * Poop! (Normally after eating.)
15       */
16      public function poop()
17      {
18          echo "The giant panda takes a giant poop.\n";
19      }
20
21      /**
22       * Sleepy time!
23       */
24      public function sleep()
25      {
26          echo "The giant panda sleeps on the ground.\n";
```

```
27      }
28  }
```

It's time to create a new class. We're going to call it the `ZooKeeper`. This class doesn't extend any other classes, and doesn't implement any interfaces.

Example 09: The ZooKeeper class.

```
1   <?php
2
3   class ZooKeeper
4   {
5       /**
6        * Care for a panda.
7        *
8        * @param  PandaInterface $panda
9        * @return void
10       */
11      public function care(PandaInterface $panda)
12      {
13          // Perform panda stuff.
14          $panda->eat();
15          $panda->poop();
16          $panda->sleep();
17      }
18  }
```

The `ZooKeeper` class contains a single method to care for a panda. Instead of type-hinting a `RedPanda` or `GiantPanda` class directly, we type-hint the interface instead.

What this means, is that any class that implements the `PandaInterface` interface, can be passed as a parameter to the `care()` method. Not only that, but because we're type-hinting the interface directly, we can be sure that the instance passed to the method will have three methods, `eat()`, `poop()` and `sleep()`.

Let's try our `ZooKeeper` class.

Example 10: Be a ZooKeeper.

```php
<?php

// Create panda instances.
$redPanda   = new RedPanda;
$giantPanda = new GiantPanda;

// Create the zookeeper.
$keeper = new ZooKeeper;

// Care for both pandas.
$keeper->care($redPanda);
$keeper->care($giantPanda);
```

First, we instantiate both panda implementations. Next, we create a new `ZooKeeper` instance. Finally, we call the `care()` method twice, passing both panda implementations.

Let's check the result.

Example 11: Output.

```
The red panda eats some fruit.
The red panda takes a poop.
The red panda sleeps up a tree.
The giant panda eats some bamboo.
The giant panda takes a giant poop.
The giant panda sleeps on the ground.
```

You see, it doesn't matter what type of panda we have. As long as it's the right shape for a panda, we can be sure that it can eat, sleep and poop. Type-hinting the interface is our guarantee that these methods are present.

Polymorphism. A complicated word, for a simple concept! It's also one that's incredibly useful. Imagine that you have written a `TaxCalculator` class, and that it needs to write some tax information to a long-term data store. We could create a `DataStoreInterface` with `read()` and `write()` methods. This way we could create data stores for writing to the disk, writing to a database, or writing to a remote filesystem, and all of them would be interchangeable. We could let the user decide. That is how you create good applications. That is power.

22. Statics

Static is that pixely-dotty stuff that you get on your television before you tune all your channels in. Right, next chapter. Wait, you're a young handsome/beautiful/both developer aren't you? You're going to be too young to know about tuning terrestrial televisions. Let's take a look at static methods and properties instead then, shall we?

Static Properties

First you're going to need a class.

Example 01: A class!

```
1  <?php
2
3  class RedPanda
4  {
5
6  }
```

There she is, bushy tail, fine fluffy coat... what a fine specimen of a class! Normally, we'd have to create an instance of our class to use it, wouldn't we? Something like this...

Example 02: Create a class instance.

```
1  <?php
2
3  $panda = new RedPanda;
```

As we've learned previously, each instance can contain different values and states to other instances of the RedPanda class. For example, if the class had a name property, one instance might have the name 'Hamish' while another might be called 'Daniel'. The instances of the classes have state.

Well... unless you cheat a little. You see, the class itself can have state. Let's add a `static` property to our `RedPanda` class, shall we?

Example 03: A static property.

```php
<?php

class RedPanda
{
    /**
     * The panda's name!
     *
     * @var string
     */
    public static $name;
}
```

By adding the keyword `static` after our new property's scope, we've made it static. What does that mean exactly? Well, we can actually use this property without having to create an instance of the class. Here's a simple example...

Example 04: Setting a static property.

```php
<?php

RedPanda::$name = 'Hamish';

echo RedPanda::$name;
```

In the example above, you'll notice that we're not creating an instance of RedPanda. Instead, we're setting the `$name` static property using the 'scope resolution operator'... or.. `::`. Don't forget that you still need to use the `$`, when pointing out the property. This is a common gotcha, since `$this->name` wouldn't require the dollar sign.

Using this method, we can set and retrieve static properties on the class, as if they were public instance properties. That's not all! The instances that we create of this class will also retain the state of that variable across all instances. For example, let's take a look at the following snippet.

Example 05: Access a static property.

```php
<?php

RedPanda::$name = 'Hamish';

$panda = new RedPanda;

echo $panda::$name;
```

In the example above, we're setting the static parameter to 'Hamish' on the class directly. We then create a new instance of the class, and echo the static property directly from that instance. We receive the result 'Hamish'. No matter whether you are working with the class directly, or an instance of that class, the state of that variable will be common across all circumstances.

Do you think you're limited to only static properties? Oh no! Think again dear reader...

Static Methods

Classes may also have methods that are static. Let's create an example.

Example 06: A static method.

```php
<?php

class RedPanda
{
    /**
     * Make the panda be cute!
     *
     * @return void
     */
    public static function beCute()
    {
        echo 'The Red Panda rolls around in the snow.';
    }
}
```

Once again, we simply add the keyword `static` to the method declaration to make it a static one. Now let's try to use this method on the class itself, rather than an instance of a RedPanda class.

Example 07: Call a static method.

```
<?php

RedPanda::beCute();
```

Example 08: Output.

```
The Red Panda rolls around in the snow.
```

Aww, isn't that sweet! Because our method is defined as static, we can use it without having to create a class instance.

Now let's setup a trap... no not for the Panda! For ourselves. Consider the following example.

Example 08: This won't work!

```
<?php

class RedPanda
{
    public $name = 'Hamish';

    public static function beCute()
    {
        $pandaName = $this->name;
        echo "{$pandaName} rolls around in the snow.";
    }
}
```

You'll notice that the function is still static, but we've added a class property that **is not** static. The static method tries to retrieve this property using `$this->name`. What do you think will happen when we call `RedPanda::beCute()`?

Example 09: Output.

```
PHP Fatal error:  Using $this when not in object context in [file]
```

Oh dear! Non-static properties belong to instances of classes. They can hold different values in each instances. We're calling our static method on the class itself, so it

has no instance to play with. Since there is no instance, `$this` resolves to nothing. Any attempts to use it will result in a PHP fatal error.

This also means that we can't call other non-static methods from a static method on a class, because we'd have to use `$this->otherMethod()` to do so.

Does that mean that our static methods can't call to other methods? Well not entirely. Our static methods can call to other **static** methods. Here's an example.

Example 10: Chain a static method.

```
<?php

class RedPanda
{
    public static function eat()
    {
        return self::eatLeaves();
    }

    private static function eatLeaves()
    {
        return 'The Panda munches on some tasty leaves.';
    }
}
```

Right, let's run through this. We have a public static method called `eat()`, this calls to the `eatLeaves()` method and returns the result. Note that the `eatLeaves()` method is private, and thus cannot be called directly on the class. It can only be called from within another method of the class.

We're using the `self` keyword in the above example. It's used to refer to itself, or the current class. So `self::eatLeaves()` means, please make a **static** call to the `eatLeaves()` method of this class.

When we run `RedPanda::eat()` we get..

Example 11: Output.

```
The Panda munches on some tasty leaves.
```

Many developers use these public static methods to create instances of the classes that they are attached to. For example...

Example 12: A factory method.

```php
<?php

class RedPanda
{
    protected $name;

    public function __construct($name = null)
    {
        $this->name = $name;
    }

    public static function make()
    {
        return new self('Hamish');
    }

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

In the above example, our RedPanda class takes an optional `$name` parameter through it's constructor. Then, it sets the `$name` protected property to the value you have passed. The public `getName()` function can be used to access the name of the panda.

We've also got the `make()` public static method, which returns `new self()` to create a new instance of the current class, and return it. It even passes 'Hamish' as the name, so we can use `RedPanda::make()` to create a new Hamish panda instance quickly.

We mentioned that you can't access instance methods from a static method, but the reverse is perfectly acceptable! For example, you could use `self::staticMethodHere()` from a non-static method and PHP would be perfectly happy.

Late Static Binding

Late static binding is a feature that was introduced in version 5.3 of PHP. It has a really confusing name, so let's demonstrate with code instead. First we need two

classes, with a touch of inheritance.

Example 13: Statics with inheritance.

```php
<?php

class Panda
{
    public static function whoAmI()
    {
        return self::getType();
    }

    public static function getType()
    {
        return 'I am a Giant panda!';
    }
}

class RedPanda extends Panda
{
    public static function getType()
    {
        return 'I am a Red panda!';
    }
}
```

Here we have two classes. Panda is our base class, and represents a giant panda. RedPanda extends from Panda, and represents a red panda... duh. The base class has a static method called 'getType()', which tells you it's a Giant Panda. This method is overridden in the RedPanda class to let you know that it's a RedPanda.

We can use the static `whoAmI()` method on both classes to find out what type of panda we have. Let's call it now...

Example 14: Who am I?

```php
<?php

echo Panda::whoAmI() . PHP_EOL;
echo RedPanda::whoAmI();
```

Upon running this snippet, we receive the following output.

Example 15: Output.

```
I am a Giant panda!
I am a Giant panda!
```

Wait, why are they both Giant pandas!? We've overridden the `getType()` method in the RedPanda class, so shouldn't PHP respect the inheritance?

Actually, the `self` keyword will only reference the base class. If we wish to use late static binding, we need to replace the `self` keyword with `static`. Yes, I know... we sure are overusing this keyword. Let's take a look at the transformation.

Example 16: Late static binding.

```php
<?php

class Panda
{
    public static function whoAmI()
    {
        return static::getType();
    }

    public static function getType()
    {
        return 'I am a Giant panda!';
    }
}

class RedPanda extends Panda
{
    public static function getType()
    {
        return 'I am a Red panda!';
    }
}
```

You'll notice that all we've done is replaced `self` with `static`. Now let's try executing the code once more.

Example 17: Output.

```
I am a Giant panda!
I am a Red panda!
```

There we go! PHP has now respected the inheritance, and the overriding of the `getType()` method. The world is once again... dandy!

Understanding the concept of statics can be a tricky business. It was not until very late into my career as a PHP developer that I started to use them effectively. If you find this chapter a little confusing, I'd suggest moving ahead using class instances for now, and coming back to this chapter with a little more experience.

For now at least, you know how to identify static properties and methods by their keyword.

23. Exceptions

Exceptions let us know that our programs haven't worked as expected. If something goes wrong, and our code can't continue to work as expected, then we use an exception to let the user know.

Throwing

Let's create a function. We've been using lots of classes recently, let's make sure you still know how to use a function. It's best to make sure, isn't it?

Here we go...

Oh dear, I've forgotten how to create a function. Do you mind reminding me?

> Sure thing Dayle, not a problem!

Example 01: Panda feeding function.

```php
<?php

/**
 * Feed the panda some food.
 *
 * @param  string $food
 * @return void
 */
function feedPanda($food)
{
    echo "The panda eats the {$food}.";
}
```

Good job! Now I remember. Here we've got a function that lets us feed a panda, right? Let's feed it some apple. They like apple.

Example 02: Feed the panda an apple.

```
<?php

feedPanda('apple');
```

Let's run the code.

Example 03: Output.

```
The panda eats the apple.
```

Awesome! What about chocolate? Do you think the panda would like some chocolate? No? Well that's probably because chocolate is poisonous to pandas. We definitely don't want any chocolate being fed to them.

Hey, I've got an idea. Why don't we throw an exception if someone tries to feed the panda some chocolate? Let's try it out.

Example 04: Throw an exception.

```
<?php

/**
 * Feed the panda some food.
 *
 * @param  string $food
 * @return void
 */
function feedPanda($food)
{
    if ($food === 'chocolate') {
        throw new Exception('NO CHOCOLATE FOR THE PANDA!');
    }

    echo "The panda eats the {$food}.";
}
```

First, we check to see whether the food we're trying to feed the panda is equal to the string value 'chocolate'. If the string matches, we'll need to use an exception. Let's take a look at that piece of code in isolation.

Example 05: Throwing an exception.

```
<?php

throw new Exception('NO CHOCOLATE FOR THE PANDA!');
```

Exceptions are classes like any other; they need to be instantiated with the `new` keyword. The first parameter to constructor of the `Exception` class, is the message that will be associated with the exception.

Exceptions are thrown, like a tennis ball, or a party. The `throw` keyword is used to trigger an exception. Once this exception has been thrown, then the program will be halted, and the user will be alerted to the error. Let's give it a go, shall we?

Example 06: Feed the panda chocolate.

```
<?php

feedPanda('chocolate');
```

It's ok. I promise that the panda won't be harmed. Let's execute the code.

Example 07: Output.

```
PHP Fatal error:  Uncaught exception 'Exception' with message 'NO CHOCOLATE FOR \
THE PANDA!' in <file>:<line>
```

Excellent, our panda is safe.

Since exceptions are classes like any other, we are able to extend them. We can do this to make the nature of the problem that little bit clearer. Let's go ahead and extend the `Exception` class.

Example 08: A PandaException.

```
<?php

class PandaException extends Exception
{
    protected $message = 'NO CHOCOLATE FOR THE PANDA!';
}
```

You'll notice that in addition to the extension of the `Exception` class, we've also defined a protected property. Do you remember how protected works? If not, now is a great time to check the scope chapter for a refresh.

You see, the `Exception` class has a property called `$message`. This normally gets set to the parameter that we give it within the constructor. Since the property is `protected`, we are able to override its value within our own class. This means that we no longer have to pass the message when defining our extension.

Let's go ahead and throw our `PandaException` instead.

Example 09: Throw the PandaException.

```
<?php

/**
 * Feed the panda some food.
 *
 * @param  string $food
 * @return void
 */
function feedPanda($food)
{
    if ($food === 'chocolate') {
        throw new PandaException;
    }

    echo "The panda eats the {$food}.";
}
```

We're throwing our `PandaException` without any parameters. Let's see what happens when we pass the function some chocolate.

Example 10: Output.

```
PHP Fatal error:  Uncaught exception 'PandaException' with message 'NO CHOCOLATE\
 FOR THE PANDA!' in <file>:<line>
```

Great! Our exception is now more descriptive, and still has the message that we desired. Let's go one step further and expand our single exception into two.

Example 11: Two exceptions.

```
<?php

class PandaChocolateException extends Exception
{
    protected $message = 'NO CHOCOLATE FOR THE PANDA!';
}

class PandaChilliException extends Exception
{
    protected $message = 'NO CHILLI FOR THE PANDA. OUCH!';
}
```

Now we have two exceptions, `PandaChocolateException` and `PandaChilliException`, for the two kinds of food that the panda's delicate digestive tract would struggle with.

There's no limit to the number of exceptions that we can throw, so let's go ahead and modify our code.

Example 12: Throw two exceptions.

```
<?php

/**
 * Feed the panda some food.
 *
 * @param  string $food
 * @return void
 */
function feedPanda($food)
{
    // Switch on food.
    switch ($food) {

        // Handle chocolate.
        case 'chocolate':
            throw new PandaChocolateException;

        // Handle chillies.
        case 'chilli':
            throw new PandaChilliException;
```

```
21
22          // Handle other food.
23          default:
24              echo "The panda eats the {$food}.";
25      }
26  }
```

Here, we've used a switch to deal with the different types of food. Why a switch? Well it's a great excuse to make sure you've learned how they work.

In the switches section, we discovered that the `break` keyword should be used to halt the execution of the application. Hold on, we've not used the `break` keyword in the switch statement above. Why do you think that is?

Well you see, exceptions also halt the thread of execution, so we don't need the `break` statements. Let's try and trigger the `PandaChilliException`, shall we?

Example 13: Output.

```
1  Fatal error: Uncaught exception 'PandaChilliException' with message 'NO CHILLI F\
2  OR THE PANDA. OUCH!' in <file> on line <line>
```

Awesome! Well it looks like we're now masters of throwing exceptions. The trouble is, how is this better than just using a PHP function such as `exit()` or `die()` to halt execution instead? What do we have to gain? Well, we're not quite done yet!

The `exit()` or `die()` functions can be used to terminate the PHP application. If you pass a string as a parameter, it will display the string before exiting. Pretty useful!

Try & Catch

Now that we've learned to throw exceptions, it's time to learn how to catch them. Go ahead and put on your baseball mitt.

If our code throws exceptions, then we can use `try` and `catch` to handle them appropriately. This way, our code can recover, and reach it's true end.

Let's consider the final code snippet from the previous section.

Example 14: Throwing two exceptions again.

```
<?php

/**
 * Feed the panda some food.
 *
 * @param  string $food
 * @return void
 */
function feedPanda($food)
{
    // Switch on food.
    switch ($food) {

        // Handle chocolate.
        case 'chocolate':
            throw new PandaChocolateException;

        // Handle chillies.
        case 'chilli':
            throw new PandaChilliException;

        // Handle other food.
        default:
            echo "The panda eats the {$food}.";
    }
}
```

Let's start small. Let's handle the situation where the panda is fed some chocolate. Here's how we might achieve this.

Example 15: Catch an exception.

```
<?php

// Set the type of food.
$food = 'chocolate';

try {
    feedPanda($food);
} catch (PandaChocolateException $exception) {
```

```
9       echo 'That was a close one! No chocolate for the panda.';
10  }
```

There's some new syntax here, isn't there? Let's pull it out of the code for clarity.

Example 16: Catch any exception.

```
1   <?php
2
3   try {
4
5       // Code that might trigger the exception.
6
7   } catch (Exception $exception) {
8
9       // Code to handle the exception.
10
11  }
```

What we have here, is two connected code blocks. They can be broken down into two purposes.

The `try` block wraps around the code that might trigger an exception. Any exceptions that are thrown by the code executed within this block, are handed to the `catch` section.

The `catch` block traps the exceptions that are throwing in the `try`. Instead of halting the execution of the code, you can handle them here, allowing for a chance at recovery. For example, in the code snippet above, we're trapping any exceptions of type `Exception`. Note that this will also trap exceptions which inherit from the `Exception` class, so that's all of them!

Whatever exception class you type-hint in the catch block, will become trapped when it is thrown within the try. The instance that is thrown, will be passed as a parameter. This allows you to access properties such as the message, or error code. Or if you have decided that it couldn't be handled, you could `throw` it once more.

Now then, let's take a look at our original code snippet once more.

Example 17: Catch the chocolate exception.

```
<?php

// Set the type of food.
$food = 'chocolate';

try {
    feedPanda($food);
} catch (PandaChocolateException $exception) {
    echo 'That was a close one! No chocolate for the panda.';
}
```

We've wrapped our `feedPanda()` function within the try catch, because we know that's the code that might throw an exception.

In the `catch` section we've type hinted the `PandaChocolateException`, so when this gets thrown, it will be caught within the second code block. Instead of letting the exception trigger an error, we've decided to echo a useful message to our users.

Let's go ahead and execute the code.

Example 18: Output.

```
That was a close one! No chocolate for the panda.
```

Perfect! We've prevented the panda from being ill. That's one of our exceptions though, shouldn't we try to catch both of them? After all, what use is only catching one exception? Let's give this a go, shall we?

Example 19: Catch both exceptions.

```
<?php

// Set the type of food.
$food = 'chilli';

try {
    feedPanda($food);
} catch (PandaChocolateException $exception) {
    echo 'That was a close one! No chocolate for the panda.';
} catch (PandaChilliException $exception) {
    echo 'That was a close one! No chilli for the panda.';
}
```

To catch additional exceptions, we can simply chain additional `catch` blocks onto our try-catch block. We type-hint a different exception within each block, so that we can deal with our exceptions in different ways.

Let's execute the code once more.

Example 20: Output.

```
That was a close one! No chilli for the panda.
```

Great. There's one final... heh... section to the try-catch block. It's optional, but let's take a look anyway.

Finally

The `finally` block allows for some code to be executed after the `try` and `catch`. If the code is terminated within the `catch` then it will never reach the `finally`. Let's take a look at the structure.

Example 21: The finally block.

```php
<?php

try {

    // Code that might trigger the exception.

} catch (Exception $exception) {

    // Code to handle the exception.

} finally {

    // Code to run after the try and catch.

}
```

Now we can examine this construct as a three stage process.

1. `try` to execute a piece of code.
2. Attempt to `catch` any problems that might arise.
3. `finally` execute some additional code.

Let's look at a practical example for this construct.

Example 22: A database connection.

```php
<?php

try {

    $database->connect('username', 'password');

} catch (DatabaseException $databaseException) {

    echo 'Unable to connect to the database.';
    die();

} finally {

    $database->fetchRecords();

}
```

We wrap the code for connecting to the database within a `try` block. We can't be sure that the username and password is correct, and if this is not the case, then an exception is thrown.

To prevent this exception from halting the code in an ugly manner, we `catch` the exception, and exit the application with a useful error message.

If the connection to the database is successful, we can use it to retrieve the record that we want to use, and our application can continue as normal.

Use exceptions in your applications to handle errors in an elegant manner, and use `try` blocks to ensure that these exceptions are dealt with in an appropriate manner. I promise that this will make you a better programmer.

24. Traits

In the inheritance chapter, we learned how to extend classes to share common functionality. The problem with extending classes, is that you can only extend one. This is a little limiting.

Implementation

In PHP 5.4 a new construct, called a 'trait', was introduced to the language. Using traits, it was now possible for PHP classes to inherit methods and properties from multiple sources. A direct solution to the problem of single inheritance.

Let's take a look at a trait, shall we?

Example 01: My first trait.

```
1  <?php
2
3  trait SomethingTrait
4  {
5
6  }
```

As you can see, a trait looks very similar to a class. Instead, it is declared using the `trait` keyword. The convention of naming a trait with the `Trait` suffix is quite a popular one, so it might be something that you should consider adopting.

Unlike a class, what's important to remember, is that you cannot add `extends` or `implements` to a trait. You can't instantiate a trait, either. Their sole purpose is to support our classes, and not to replace them.

Traits can have methods, just like classes do. For a moment, let's think back to our `Animal` class. Do you remember it? Here it is once again to refresh our memory.

Example 02: An abstract class.

```
<?php

abstract class Animal
{

}
```

Our class called `Animal` is an abstract one. This means that it's meant to be extended by our classes that represent actual animals.

Consider the possibility that we'd like to add a `stroke()` method. For example, we'd like to `stroke()` our red pandas to make them purr. We could add something like the following.

Example 03: The stroke method.

```
<?php

abstract class Animal
{
    /**
     * Stroke the animal.
     *
     * @return void
     */
    public function stroke()
    {
        echo 'This animal purrs contently.';
    }
}
```

Sure! That would work, right? The trouble is, you wouldn't stroke a snake, would you? No, it doesn't make sense to stroke every animal... Hmm, what can we do? Oh right, traits!

You see, traits are perfect for adding a new.. well... trait, to an object. We want to add a trait to animals which are strokeable, so what we're looking for is a `StrokeableTrait`. Sure, that makes sense to me! Let's get going with this idea.

Example 04: The StrokableTrait.

```
<?php

trait StrokeableTrait
{
    /**
     * Stroke the animal.
     *
     * @return void
     */
    public function stroke()
    {
        echo 'This animal purrs contently.';
    }
}
```

We've gone ahead and moved the `stroke()` method to the trait. You'll find no other differences here. This means, that we can also remove the method from `Animal`, like this.

Example 05: Abstract Animal.

```
<?php

abstract class Animal
{

}
```

Perfect, we're all set. Let's create a red panda class, shall we? After all, that's what this book is all about. Well, that and PHP I suppose...

Example 06: The RedPanda class.

```
class RedPanda extends Animal
{
    /**
     * The panda's name.
     *
     * @var string
     */
    private $name = 'Jasmina';
}
```

Beautiful, isn't she? We've extended the `Animal` class, and while it might be empty at the moment, go ahead and assume that it could be full of useful methods such as `eat()`, `sleep()` and `poop()` for example.

Now, let's think about our Panda for a moment. She's definitely strokeable, isn't she? Let's go ahead and use the trait.

Example 07: Use the StrokeableTrait.

```
<?php

class RedPanda extends Animal
{
    use StrokeableTrait;

    /**
     * The panda's name.
     *
     * @var string
     */
    private $name = 'Jasmina';
}
```

I find this a little bit annoying, but unfortunately, the keyword used to apply a trait to a class is once again the `use` keyword. This is the same keyword that you'll find in the namespaces chapter. It's unfortunate naming, but it works all the same.

By placing the statement...

Example 08: Use the trait.

```
use StrokeableTrait;
```

... right inside the class, we've applied the trait, and our `RedPanda` has gained the functionality from the trait.

Let's try using the method inherited from our trait.

Example 09: Stroke the panda.

```
<?php

// Create a new red panda.
$panda = new RedPanda;

// Stroke the panda.
$panda->stroke();
```

We create a new instance of the `RedPanda` class, and call the `stroke()` method directly. Let's see what happens.

Example 10: Output.

```
This animal purrs contently.
```

That's purrrfect! Now, we've only used a single trait in this scenario, but it's entirely possible to use as many traits as you need. For example, the following is entirely legal.

Example 11: Multiple traits.

```
<?php

class RedPanda extends Animal
{
    use PlayableTrait;
    use CareableTrait;
    use StrokeableTrait;

    /**
     * The panda's name.
     *
```

```
12          * @var string
13          */
14         private $name = 'Jasmina';
15     }
```

Use traits to enhance the functionality of your classes. Of course, you can only use them on PHP version 5.4 or higher, so keep that in mind.

Priority

Now that you've learned about both inheritance and traits, you might be asking yourself a very fair question. What happens if we extend a class, and use a trait that both have a method of the same name.

The best way to demonstrate this, is by testing it out. Consider the following code sample.

Example 12: Create test classes and traits.

```
1   <?php
2
3   trait TestTrait
4   {
5       public function test()
6       {
7           echo 'This is the trait method.';
8       }
9   }
10
11  class TestClass
12  {
13      public function test()
14      {
15          echo 'This is the class method.';
16      }
17  }
```

Here we have a `TestTrait` and a `TestClass`. They both have a method called `test()`, which indicates the source of the method call.

It's time to setup our test-case.

Example 13: Set our gauntlet.

```php
<?php

// Define a class which extends the baseclass
// and implements the trait.
class Testing extends TestClass
{
    use TestTrait;
}

// Create our testcase.
$test = new Testing;

// Run the test method.
$test->test();
```

We've created a class which extends the `TestClass` and uses the `TestTrait`. By calling the shared method `test()`, we can find out which one takes priority from the output. Let's give it a go.

Example 14: Output.

```
This is the trait method.
```

Sometimes, it's better to let the code answer our questions, don't you find? As we can see, the method within the trait, has taken priority over the method within the inherited class.

This means that we can use traits to override the functionality inherited from abstract classes. This is useful knowledge to a budding young programmer such as yourself. Use it wisely!

25. Namespaces

In PHP version 5.3, a new feature known as namespacing was added to the language. Many modern languages have already had this feature for some time, but PHP was a little late to the scene. None the less, every new feature has a purpose. Let's find out why PHP namespaces can benefit our applications.

The truth is, I stole this chapter from one of my blog articles. It was actually one of the most popular blog articles I've ever written, and for that reason, I'd rather use this than have another attempt and not be so successful. I'm a little bit sorry for the lazy-reuse, but only a little bit. Now you're learning one of the most important parts about being a programmer. DONT REPEAT YOURSELF (DRY).

In PHP, you can't have two classes that share the same name. They have to be unique. The issue with this restriction, is that if you are making use of someone else's code, and they have a class named `User`, then you can't create your own class also called `User`. This is a real shame, because that's a pretty convenient class name, am I right?

PHP namespaces allow us to circumvent this issue. In fact, we can have as many `User` classes as we like. Not only that, but we can use namespaces to contain our similar code into neat little packages, or even to show ownership.

Let's take a look at a normal class. Yes... I know you have used them before. Just trust me on this one okay?

Global Namespace

Here's a really simple class.

Example 01: The Eddard class.

```
1  <?php
2
3  // app/models/Eddard.php
4
5  class Eddard
6  {
7
8  }
```

There's nothing special to it. If we want to use it then we can do this.

Example 02: Create a new Eddard.

```php
<?php

// app/routes.php

$eddard = new Eddard();
```

> Dayle, I know some PHP...

Okay, okay sorry. Basically, we can think of this class as being in the 'global' namespace. I don't know if that's the right term for it, but it sounds quite fitting to me. It essentially means that the class exists without a namespace. It's just a normal class.

Simple Namespacing

Let's create another class alongside the original, global Eddard.

Example 03: The Stark namespace.

```php
<?php

namespace Stark;

// app/models/another.php

class Eddard
{

}
```

Here we have another Eddard class, with one minor change. The addition of the `namespace` directive. The line `namespace Stark;` informs PHP that everything we do is relative to the `Stark` namespace. It also means that any classes created within this file will live inside the 'Stark' namespace.

Now, let's try to use the 'Stark' class once again.

Example 04: Create Eddard again.

```
<?php

// app/routes.php

$eddard = new Eddard();
```

Once again, we get an instance of the first class we created in the last section, and not the one within the 'Stark' namespace. Let's try to create an instance of the 'Eddard' within the 'Stark' namespace.

Example 05: Create a StarkEddard.

```
<?php

// app/routes.php

$eddard = new Stark\Eddard();
```

We can instantiate a class within a namespace, by prefixing it with the name of the namespace, and separating the two with a backward `\` slash. Now we have an instance of the 'Eddard' class within the 'Stark' namespace. Aren't we magical?!

You should know that namespaces can have as many levels of hierarchy as they need to. For example:

Example 06: A horrible, but legal namespace.

```
This\Namespace\And\Class\Combination\Is\Silly\But\Works
```

The Theory of Relativity

Remember how I told you that PHP always reacts **relative** to the current namespace? Let's take a look at this in action.

Example 07: Create Eddard within Stark namespace.

```
<?php

namespace Stark;

// app/routes.php

$eddard = new Eddard();
```

By adding the namespace directive to the instantiation example, we have moved the execution of the PHP script into the 'Stark' namespace. Since we are inside the same namespace as the one we put 'Eddard' into, this time we receive the namespaced 'Eddard' class. Do you see how it's all relative?

Now that we have changed namespace, we have created a little problem. Can you guess what it is? How do we instantiate the original 'Eddard' class? The one not in the namespace.

Fortunately, PHP has a trick for referring to classes that are located within the global namespace, we simply prefix them with a backward (\) slash.

Example 08: Create a global Eddard instance.

```
<?php

// app/routes.php

$eddard = new \Eddard();
```

With the leading backward (\) slash, PHP knows that we are referring to the 'Eddard' in the global namespace, and instantiates that one.

Use your imagine a little, like how Barney showed you. Imagine that we have another namespaced class called `Tully\Edmure`. We want to use this class from within the 'Stark' framework. How do we do that?

Example 09: Root namespace prefix.

```
<?php

namespace Stark;

// app/routes.php

$edmure = new \Tully\Edmure();
```

Again, we need the prefixing backward slash to bring us back to the global namespace, before instantiating a class from the 'Tully' namespace.

It could get tiring, referring to classes within other namespaces by their full hierarchy each time. Luckily, there's a nice little shortcut we can `use`. Let's see it in action.

Example 10: Create a Tully\Edmure instance.

```
<?php

namespace Stark;

use Tully\Edmure;

// app/routes.php

$edmure = new Edmure();
```

Using the `use` statement, we can bring one class from another namespace into the current namespace. This will allow us to instantiate it by name only.

Let me show you another neat trick. We can give our imported classes little nicknames, like we used to in the PHP playground. Let me show you.

Example 11: Namespace aliasing.

```
<?php

namespace Stark;

use Tully\Brynden as Blackfish;

// app/routes.php

$brynden = new Blackfish();
```

By using the ‘as‘ keyword, we have given our ‘Tully/Brynden’ class the ‘Blackfish’ nickname. This will allow us to use the new nickname to identify it within the current namespace. Neat trick right? It’s also really handy if you need to use two similarly named classes within the same namespace. For example:

Example 12: Similar named classes.

```
<?php

namespace Targaryen;

use Dothraki\Daenerys as Khaleesi;

// app/routes.php

class Daenerys
{

}

// Targaryen\Daenerys
$daenerys = new Daenerys();

// Dothraki\Daenerys
$khaleesi = new Khaleesi();
```

By giving the ‘Daenerys’ within the ‘Dothraki’ namespace a nickname of ‘Khaleesi’, we are able to use two ‘Daenerys’ classes by name only. Handy right? The game is all about avoiding conflicts, and grouping things by purpose or faction.

You can `use` as many classes as you need to.

Example 13: Use multiple classes.

```
<?php

namespace Targaryen;

use Dothraki\Daenerys;
use Stark\Eddard;
use Lannister\Tyrion;
use Snow\Jon as Bastard;
```

Structure

Namespaces aren't just about avoiding conflicts. We can also use them for organisation, and for ownership. Let me explain with another example.

Let's say I want to create an open source library. I'd love for others to use my code. It would be great! The trouble is, I don't want to cause any problematic class name conflicts for the person using my code. That would be terribly inconvenient. Here's how I can avoid causing hassle for the wonderful, open source embracing, individual.

Example 14: Namespace structure.

```
Dayle\Blog\Content\Post
Dayle\Blog\Content\Page
Dayle\Blog\Tag
```

Here we have used my name to show that I created the original code, and to separate my code from that of the person using my library. Inside the base namespace, I have created a number of sub-namespaces to organise my application by its internal structure.

Limitations

In truth, I feel a little guilty for calling this sub-heading 'Limitations'. What I'm about to talk about isn't really a bug.

You see, in other languages, namespaces are implemented in a similar way, and those other languages provide an additional feature when interacting with namespaces.

In Java for example, you are able to import a number of classes into the current namespace by using the import statement with a wildcard. In Java, 'import' is equivalent to 'use', and it uses dots to separate the nested namespaces (or packages). Here's an example.

Example 15: Java namespace importing.

```
import dayle.blog.*;
```

This would import all of the classes that are located within the 'dayle.blog' package.

In PHP you can't do that. You have to import each class individually. Sorry. Actually, why am I saying sorry? Go and complain to the PHP internals team instead, only, go gentle. They have given us a lot of cool stuff recently.

Here's a neat trick you can use, however. Imagine that we have this namespace and class structure, as in the previous example.

Example 16: Example structure.

```
Dayle\Blog\Content\Post
Dayle\Blog\Content\Page
Dayle\Blog\Tag
```

We can give a sub-namespace a nickname, to use its child classes. Here's an example:

Example 17: Namespace aliasing for relative access.

```
<?php

namespace Baratheon;

use Dayle\Blog as Cms;

// app/routes.php

$post = new Cms\Content\Post;
$page = new Cms\Content\Page;
$tag  = new Cms\Tag;
```

This should prove useful if you need to use many classes within the same namespace.

26. What now?

Oh no! You've gone and ran out of book, haven't you? I suppose that means that we've come to the end of our journey, for now.

You see, I'm constantly updating my books to include new content, better explanations, and a bunch of other neat stuff. Don't worry, all the updates are free! If you haven't downloaded a fresh copy in a while, you might want to do so.

Hopefully, you've found inspiration to start building some applications with PHP, and I very much encourage you to do so! Practical experience is something that every developer will need to further their skillset.

Did you find something missing from the book, or is there something that you didn't quite understand? Send me an email to [me@daylerees.com] to let me know, and I'll update the book to help both you and others.

Didn't find anything missing? Send me an email anyway to say hi! I love hearing from my readers.

If you're eager to learn more about PHP, then let me share with you a little secret. PHP Pandas, is part of a bigger series of books that I like to call 'PHP for Everyone'. The next instalment in the series, will be a book about using PHP on the web. So be sure to follow @daylerees on twitter, and keep your eyes open for future updates!

If you've enjoyed reading PHP Pandas, then please share your enjoyment with the world. Recommend the book to your friends, tweet about it, or post it whenever someone is looking to learn the language. I'd love more than anything for this book to become the #1 resource for learning the PHP language.

What, you're still here? Well, our journey may have come to an end, but if you're itching for more PHP, then let me refer you to a friend of mine. My buddy Alex Garrett runs a website called 'PHPAcademy'. He's currently rebranding it to the more awesome 'CodeCourse' name, and you'll find a bunch of videos on PHP and other topics over at https://www.youtube.com/user/phpacademy/videos[1].

http://phpacademy.org/ <– Will redirect to CodeCourse soon!

Well, that's all from me for now. You've been a great, handsome AND / OR beautiful reader. Now go and be a great programmer.

No pandas were harmed during the writing of this book. My arthritic fingers on the other hand...

[1] https://www.youtube.com/user/phpacademy/videos

Dayle.

xx