Gilles Crettenand

Functional PHP

Uncover the secrets of functional programming with PHP to ensure your applications are as great as they can be

Packt

Functional PHP

Table of Contents

Functional PHP

Functional PHP

First published: February 2017

Production reference: 1100217

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78588-032-2

www.packtpub.com

Credits

Author Gilles Crettenand	**Copy Editor** Safis Editing
Reviewer Andrew Caya	**Project Coordinator** Vaidehi Sawant
Commissioning Editor Kunal Parikh	**Proofreader** Safis Editing
Acquisition Editor Sonali Vernekar	**Indexer** Francy Puthiry
Content Development Editor Rohit Kumar Singh	**Graphics** Jason Monteiro
Technical Editor Pavan Ramchandani	**Production Coordinator** Arvindkumar Gupta

About the Author

Gilles Crettenand is a passionate and enthusiastic software developer. He thrives when solving challenges and is always on the lookout for a better way to implement his solutions. He has a bachelor's degree in computer science from the School of Engineering and Management Vaud in Switzerland and has more than 6 years of experience as a web developer.

Most of his professional career has been spent using PHP, developing a variety of applications ranging from accountability software to e-commerce solutions and CMS. At nights, however, he likes to try other languages, dabbling with Haskell, Scala, and more recently, PureScript, Elm, and Clojure.

Being a certified scrum master, he understands the need to use the right tool for the job and maintaining a readable and maintainable codebase using the best techniques and practices available.

I'd like to thank my wife, Charlotte, for being understanding throughout the process of writing this book. My friend Loris also helped me a lot in putting some thoughts to paper and when I was struggling with some theoretical aspects.

Kudos to my colleagues; without them, I wouldn't be half the developer I am today. I will always be grateful for the hours of peer reviews and discussions about technical solutions and application architecture.

I also want to thank the people from my coworking space who provided me with the much needed distraction when faced with a blank page. Thanks for the laughs and banter!

Finally, I am appreciative to Packt for giving me the chance to write a book about my passions although I had no prior experience.

About the Reviewer

Andrew Caya discovered his passion for computers at the age of 11 and started programming in GW-BASIC and QBASIC in the early 90s. He also did some software development in C, C++, and Perl, and some Linux system administration before becoming a PHP developer more than 7 years ago. He is now a professional contract programmer in Montreal, Canada, and a loving husband and father.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://goo.gl/wYWIUc.

If you'd like to join our team of regular reviewers, you can email us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Preface

Functional programming is a paradigm that is present every year at conferences. The JavaScript community is probably one of the first that approached the subject, but the topic is now also discussed among the developers using various other languages such as Ruby, Python, and Java.

PHP has most of the features that are needed to start using a functional approach for development. You have no reason to be left on the side, which is why this book proposes to teach you the fundamentals of functional programming.

If you are completely new to functional programming or you want to refresh your basics and learn a bit about its history and benefits, I recommend that you start with the appendix. It is not the first chapter of the book as the content is not directly related to PHP, but it will help you put various topics in context and have a better idea of the topics covered in this book.

What this book covers

Chapter 1, *Functions as First Class Citizen in PHP*, discusses how functional programming, as the name suggests, revolves around functions. In this chapter, you will learn the about the various way they can be declared and used in PHP.

Chapter 2, *Pure Functions, Referential Transparency, and Immutability*, covers the three concepts that are the cornerstone of any functional code base. You will learn what they are about and how to apply them to our benefit.

Chapter 3, *Functional Basis in PHP*, discusses how functional programming, like any paradigm, rests upon a few core concepts. This chapter will present them in a simple fashion before going further.

Chapter 4, *Composing Functions*, describes how functions are often used as a building block using function composition. In this chapter, you will learn how to do it in PHP what it is important to keep in mind when doing so.

Chapter 5, *Functors, Applicatives, and Monads*, starts with easier concepts, such as the functor and the applicative, and we will build up our knowledge to finally present the monad in a light that should dispel some of the fear floating around this term.

Chapter 6, *Real-life Monads*, helps you learn about some real-life usage of the monad abstraction and how it can be used to write better code.

Chapter 7, *Functional Techniques and Topics*, brushes upon topics such as type systems, pattern matching, point-free style, and others from the vast field of functional programming.

Chapter 8, *Testing*, teaches you that functional programming not only helps with writing code that is easier to understand and maintain, but it is also great to facilitate testing.

Chapter 9, *Performance Efficiency*, lets you know that using functional techniques in PHP has a cost. We will first discuss it and then see how it can help in other performance-related topics.

Chapter 10, *PHP Frameworks and FP*, introduces a technique that can be applied to improve your code in any project, as there is currently now dedicated framework for functional programming in PHP.

Chapter 11, *Designing a Functional Application*, will present you with some advice if you want to develop a whole application using the most functional code possible. You will also learn about Functional Reactive Programing and the RxPHP library.

`Appendix`, *What are We Talking about When We Talk about Functional Programming?*, is a presentation and history of functional programming alongwith its benefits and a glossary. It's really the first part of the book

you should read, but as we don’t approach the subject from the PHP angle, it is presented as an appendix.

What you need for this book

You will need to have access to a computer with PHP installed. It will be easier if you know how to use the command line, but all examples should also work in a browser with maybe some small adaptations.

While learning functional programming, I also recommend the usage of a Read-Eval-Print-Loop (REPL). I personally used **Boris** when writing this book. You can find it at https://github.com/borisrepl/boris. Another great option is **PsySH** (http://psysh.org).

Although not at all mandatory, a REPL will allow you to quickly test your ideas and play around with the various concepts that will be presented in this book without having to juggle between your editor and command line.

I also assume you have Composer available and that you know how to use it to install new packages; if not, you can find it at https://getcomposer.org. Multiple libraries will be presented throughout the book and the preferred way to install them is using composer.

All the code written in the book was tested on PHP 7.0, which is the de facto recommended version. It should, however, also run on any newer version. Running most of the examples should also be fine on PHP 5.6 after making some minor adaptations. We will use the new scalar type hinting feature introduced in PHP 7.0 throughout the book, but if you remove those, the code should be readily compatible with lower versions.

Who this book is for

This book requires no knowledge of functional programming; prior programming experience is, however, required. Also, basic concepts from object-oriented programming will not be covered in depth.

Deep knowledge of the PHP language is not mandatory, as uncommon syntax will be explained. The book should be understandable to someone who hasn't written a single line of PHP code, with some effort.

This book can be considered as a beginner book about functional programming in PHP, meaning that we will build knowledge incrementally. However, the topic being pretty vast and the limited page count, we will move quickly at times. This is why I encourage you to play with the variously presented concepts as we learn them and take some time at the end of each chapter to make sure you understood it correctly.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The next lines of code read the link and assign it to the `BeautifulSoup` function."

A block of code is set as follows:

```
<?php
function getPrices(array $products) {
  $prices = [];
  foreach($products as $p) {
    if($p->stock > 0) {
      $prices[] = $p->price;
    }
  }
  return $prices;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<?php
function getPrices(array $products) {
  $prices = [];
  foreach($products as $p) {
    if($p->stock > 0) {
      $prices[] = $p->price;
    }
  }
  return $prices;
}
```

Any command-line input or output is written as follows:

```
composer require rx/stream
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Functional-PHP. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Chapter 1. Functions as First Class Citizens in PHP

Functional programming, as its name suggests, revolves around functions. In order to apply functional techniques effectively, a language has to support functions as the first class citizen, or also **first functions**.

This means that functions are considered like any other value. They can be created and passed around as parameters to other functions and they can be used as return value. Luckily, PHP is such a language. This chapter will demonstrate the various way functions can be created and used.

In this chapter, we will cover the following topics:

- Declaring function and methods
- Scalar type hints
- Anonymous functions
- Closures
- Using objects as functions
- Higher Order Functions
- The callable type hint

Before we begin

As the first release of PHP 7 happened in December 2015, it will be the version that will be used for the examples in this book.

However, since it's a fairly new version, each time we use a new feature, it will be clearly outlined and explained. Also, since not everyone is able to migrate right away, changes needed to run the code on PHP 5 will be proposed whenever possible.

The latest version available at the time of writing is 7.0.9. All code and examples are validated using this version.

Coding standards

Examples in this book will respect **PSR-2** (**PHP Standard Recommendation2**) and its parent recommendation, PSR-1, for their coding style, as should most of the libraries presented. For people not familiar with them, here are the most important parts:

- Classes are in a namespace and use CamelCase with the first letter capitalized
- Methods use CamelCase without the first letter capitalized
- Constants are written with all letters in capital
- Braces for classes and methods are on a new line, other braces are on the same line

Also, although not being defined in PSR-2, the following choices were made:

- Function names are in snake_case
- Parameters, variables, and property names are in snake_case
- Properties are private whenever possible

Autoloading and Composer

The examples will also assume the presence of a PSR-4 compatible autoloader.

As we will use Composer dependency manager to install the presented libraries, we recommend using it as the autoloader.

Functions and methods

Although this book is not designed for PHP beginners, we will quickly cover some basis in order to be sure we share a common vocabulary.

In PHP, you usually declare a function using the `function` keyword:

```
<?php

function my_function($parameter, $second_parameter)
{
    // [...]
}
```

A function declared inside a class is called a **method**. It differs from a traditional function as it can access the object properties, have visibility modifiers, and can be declared static. As we will try to write code as pure as possible, our properties will be of `private` type:

```
<?php

class SomeClass
{
   private $some_property;

   // a public function
   public function some_function()
   {
       // [...]
   }

   // a protected static function
   static protected function other_function()
   {
       // [...]
   }
}
```

PHP 7 scalar type hints

You were already able to declare type hints for classes, callables, and arrays in PHP 5. PHP 7 introduces the idea of scalar type hints. This means you can now say that you want a `string`, an `int`, a `float`, or a `bool` data type, both for parameters and return types. The syntax is roughly similar to what can be found in other languages.

Contrary to class type hints, you can also choose between two modes: the **strict** mode and the **non-strict** mode, the latter being the default. This means PHP will try to cast the values to the desired type. The casts will happen silently if there is no loss of information, otherwise, a warning will be raised. This can lead to the same strange results you can have with string to numbers conversion or true and false values.

Here are some examples of such casts:

```
<?php

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

echo add(3.5, 1);
// 4.5
echo add(3, 1);
// 4
echo add("3.5", 1);
// 4.5
echo add(3.5, 1.2); // 1.2 gets casted to 1
// 4.5
echo add("1 week", 1); // "1 week" gets casted to 1.0
// PHP Notice:  A non well formed numeric value encountered
// 2
echo add("some string", 1);
// Uncaught TypeError Argument 1 passed to add() must be of the
type float, string given

function test_bool(bool $a): string {
    return $a ? 'true' : 'false';
}
```

```
echo test_bool(true);
// true
echo test_bool(false);
// false
echo test_bool("");
// false
echo test_bool("some string");
// true
echo test_bool(0);
// false
echo test_bool(1);
// true
echo test_bool([]);
// Uncaught TypeError: Argument 1 passed to test_bool() must be
of the type Boolean
```

If you want to avoid issues with casting, you can opt-in for the strict mode. This way PHP will raise an error each time the values do not exactly conform to the desired type. In order to do so, the `declare(strict_types=1)` directive must be added to the very first line of your file. Nothing must precede it.

The only cast that PHP allows itself is from `int` to `float` by adding `.0` as there is absolutely no risk of data loss.

Here are the same examples as before, but with strict mode activated:

```
<?php

declare(strict_types=1);

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

echo add(3.5, 1);
// 4.5
echo add(3, 1);
// 4
echo add("3.5", 1);
// Uncaught TypeError: Argument 1 passed to add() must be of
the type float, string given
echo add(3.5, 1.2); // 1.2 gets casted to 1
```

```
// Uncaught TypeError: Argument 2 passed to add() must be of
the type integer, float given
echo add("1 week", 1); // "1 week" gets casted to 1.0
// Uncaught TypeError: Argument 1 passed to add() must be of
the type float, string given
echo add("some string", 1);
// Uncaught TypeError: Argument 1 passed to add() must be of
the type float, string given

function test_bool(bool $a): string {
    return $a ? 'true' : 'false';
}

echo test_bool(true);
// true
echo test_bool(false);
// false
echo test_bool("");
// Uncaught TypeError: Argument 1 passed to test_bool() must be
of the type boolean, string given
echo test_bool(0);
// Uncaught TypeError: Argument 1 passed to test_bool() must be
of the type boolean, integer given
echo test_bool([]);
// Uncaught TypeError: Argument 1 passed to test_bool() must be
of the type boolean, array given
```

Although not demonstrated here, the same casting rules apply for return types. Depending on the mode, PHP will happily perform the same casting and display the same warning and errors as for parameters hints.

Also, another subtlety is that the mode that is applied is the one being declared at the top of the file where the function call is made. This means that when you call a function that was declared in another file, the mode this file was in is not taken into account. Only the directive at the top of the current file matters.

Concerning errors raised about types, we will see in Chapter 3, *Functional basis in PHP* how exception and error handling was changed in PHP 7 and how you can use it to catch those.

From now on, every time it makes sense, our examples will use scalar

type hints to make the code more robust and readable.

Imposing types can be seen as cumbersome and will probably lead to a few irritations when you start using them, but in the long run, I can assure you that it will save you from some nasty bugs. All checks that can be done by the interpreter are something you don't need to test yourself.

It also makes your function easier to understand and reason with. The person looking at your code won't have to ask themselves what could a value be, they know with certitude what kind of data they have to pass as parameters and what they will get back. The result is that the cognitive burden is lessened and you can use your time thinking of solving issues instead of keeping in mind menial details about your code.

Anonymous functions

You were probably well aware of the syntax we saw to declare functions. What you may not know is that a function does not necessarily need to have a name.

Anonymous functions can be assigned to variables, used as callbacks and have parameters.

In the PHP documentation, the term anonymous function is used interchangeably with the term *closure*. As we will see in the following code snippet, an anonymous function is even an instance of the `Closure` class, which we will discuss. According to the academic literature both concepts, although similar, are a bit different. The first usage of the term closure was in 1964 by Peter Landin in *The mechanical evaluation of expressions*. In the paper, a closure is described as having an environment part and a control part. The functions we will declare in this section won't have any environment, so they won't be, strictly speaking, closures.

In order to avoid confusion when reading other work, this book will use the term *anonymous function* to describe a function without a name, as presented in this section:

```
<?php

$add = function(float $a, float $b): float {
    return $a + $b;
};
// since this is an assignment, you have to finish the
statement with a semicolon
```

The previous code snippet declared an anonymous function and assigned it to a variable so that we can reuse it later either as a parameter to another function or call it directly:

```
$add(5, 10);
$sum = array_reduce([1, 2, 3, 4, 5], $add, 0);
```

You can also declare an anonymous function directly as a parameter if you don't plan to reuse it:

```
<?php
$uppercase = array_map(function(string $s): string {
  return strtoupper($s);
}, ['hello', 'world']);
```

Or you can return a function as you would return any kind of value:

```
<?php

function return_new_function()
{
  return function($a, $b, $c) { /* [...] */};
}
```

Closures

As we saw earlier, the academicals description of a closure is a function that has access to some outside environment. Throughout this book, we will keep to this semantics, despite PHP calling both anonymous functions and closure using the later term.

You may be familiar with JavaScript closures, where you can simply use any variable from the outside scope without doing anything particular. In PHP, you need to use the `use` keyword to import an existing variable into the scope of an anonymous function:

```
<?php

$some_variable = 'value';

$my_closure = function() use($some_variable)
{
  // [...]
};
```

PHP closures use an early-binding approach. This means that the variable inside the closure will have the value that the variable had at the closure creation. If you change the variable afterward, the change will not be seen from inside the closure:

```
<?php

$s = 'orange';

$my_closure = function() use($s) { echo $s; };
$my_closure(); // display 'orange'

$a = 'banana';
$my_closure(); // still display 'orange'
```

You could pass the variable by reference so that changes to the variable are propagated inside the closure, but since this is a book on functional programming where we try to use immutable data structures and avoid having state, figuring how to do it is left as an exercise to the reader.

Be aware that when you pass objects to a closure, any modification done to properties in the object will be accessible inside the closure. PHP does not make a copy of objects when passed to the closure.

Closures inside of classes

If you declare any anonymous function inside a class, it will automatically get access to the instance reference via the usual `$this` variable. To stay coherent about the vocabulary, the function will automatically become a closure:

```
<?php

class ClosureInsideClass
{
    public function testing()
    {
        return function() {
            var_dump($this);
        };
    }
}

$object = new ClosureInsideClass();
$test = $object->testing();

$test();
```

If you want to avoid this automatic binding, you can declare a static anonymous function:

```
<?php

class ClosureInsideClass
{
    public function testing()
    {
        return (static function() {
            // no access to $this here, the following line
            // will result in an error.
            var_dump($this);
        });
    }
```

```
};

$object = new ClosureInsideClass();
$test = $object->testing();

$test();
```

Using objects as functions

Sometimes, you might want to split your function into smaller parts, but without those parts being accessible to everyone. When this is the case, you can leverage the `__invoke` magic method on any object that let you use an instance as a function and hide that helper function as private methods inside your object:

```
<?php

class ObjectAsFunction
{
    private function helper(int $a, int $b): int
    {
        return $a + $b;
    }

    public function __invoke(int $a, int $b): int
    {
      return $this->helper($a, $b);
    }
}

$instance = new ObjectAsFunction();
echo $instance(5, 10);
```

The `__invoke` method will be called with any parameters you pass to the instance. If you want, you can also add a constructor to your object and use any methods and properties that it contains. Just try to keep it as pure as possible, because as soon as you use mutable properties, your function will be harder to understand.

The Closure class

All anonymous functions are in fact an instance of the `Closure` class. However, as stated in the documentation (http://php.net/manual/en/class.closure.php), this class does not use the aforementioned `__invoke` method; it's a special case in the PHP interpreter:

> *Besides the methods listed here, this class also has an __invoke method. This is for consistency with other classes that implement calling magic, as this method is not used for calling the function.*

This method on the class allows you to change to which object the `$this` variable will be bound inside the closure. You can even bind an object to a closure created outside of the class.

If you start using the features of the `Closure` class, keep in mind that the `call` method was just recently added in PHP 7.

Higher-order functions

PHP functions can take functions as parameters and return functions as return values. A function that does either of those is called a higher-order function. It is as simple as that.

In fact, if you read the following code samples, you will quickly see that we have already created multiple higher-order functions. You will also discover, without much surprise, that most of the functional techniques you will learn revolve around higher-order functions.

What is a callable?

A `callable` is a type hint that can be used to enforce that the parameter of a function is something that can be called, like a function. Beginning with PHP 7, it can also be used as a type hint for the return value:

```
<?php

function test_callable(callable $callback) : callable {
    $callback();
    return function() {
        // [...]
    };
}
```

However, what you cannot enforce with the type hint is the number and type of arguments your callable should have. But it is already great to guarantee to have something you can call.

A callable can take multiple forms:

- A string for named functions
- An array for class methods or static functions
- A variable for anonymous functions or closures
- An object with a `__invoke` method

Let's see how we can use all these possibilities. Let's start with calling a simple function by name:

```
$callback = 'strtoupper';
$callback('Hello World !');
```

We can also do the same for functions inside of classes. Let's declare an `A` class with some functions and use an array to call it.

```
class A {
    static function hello($name) { return "Hello $name !\n"; }
    function __invoke($name) { return self::hello($name); }
}

// array with class name and static method name
```

```
$callback = ['A', 'hello'];
$callback('World');
```

Using a string will only work for the static method, as other methods will need an object to use as their context. In the case of a static method, you can also use a simple string directly, this will, however, only work starting with PHP 7; the previous version didn't support this syntax:

```
$callback = 'A::hello';
$callback('World');
```

You can call a method on a class instance as easily:

```
$a = new A();

$callback = [$a, 'hello'];
$callback('World');
```

Since our `A` class has an `__invoke` method, we can use it as a `callable` directly:

```
$callback = $a;
$callback('World');
```

You can also use any variable to which an anonymous function is assigned as a `callable`:

```
$callback = function(string s) {
    return "Hello $s !\n";
}
$callback('World');
```

PHP also provides you with two helpers to call functions in the form of `call_user_func_array` and `call_user_func`. They take a callable as a parameter and you can also pass parameters. For the first helper, you pass an array with all the parameters; for the second one, you pass them separately:

```
call_user_func_array($callback, ['World']);
```

A final word of caution, if you are using the `callable` type hint: any

string that contains a function name that has been declared is considered valid; this can lead to some unexpected behavior sometimes.

A somewhat contrived example would be a test suite where you check that some functions only accept valid callables by passing it some strings and catching the resulting exception. At some point, you introduce a library and this test is now failing, although both should be unrelated. What is happening is that the library in question declares a function with the exact name that your string contained. Now, the function exists and no exception is raised anymore.

Summary

In this chapter, we discovered how you can create new anonymous functions and closures. You are also now familiar with the various ways you can pass those around. We also learned about the new PHP 7 scalar type hints that help us to make our program more robust, and the `callable` type hint so we can enforce having a valid function as a parameter or return value.

For anyone who has been using PHP for some time already, there was probably really nothing new in this chapter. But we now share a common ground that will help us dive into the functional world.

With the basics about functions in PHP covered, we will learn more about the fundamental concepts pertaining to functional programming in the next chapter. We will see that your functions have to respect certain rules in order to be truly useful in a functional code base.

Chapter 2. Pure Functions, Referential Transparency, and Immutability

Those who have read the appendix about functional programming will have seen that it revolves around pure functions, or in other words, functions that only use their input to produce a result.

It might seem easy to determine whether a function is pure or not. It's just about checking that you don't call any global state, right? Sadly, it's not that simple. There are also multiple ways a function can produce side effects. Some of them are pretty easy to spot; others are more difficult.

This chapter will not cover the benefits of using functional programming. If you are interested in the benefits, I suggest you read the appendix which tackles the subject in depth. However, we will discuss the advantages offered by immutability and referential transparency, as they are quite specific and are glossed over in the appendix.

In this chapter, we will cover the following topics:

- Hidden input and output
- Function purity
- Immutability
- Referential transparency

Two sets of input and output

Let's start with a simple function:

```
<?php

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

```
}
```

The input and output of this function are pretty obvious to spot. We have two parameters and one return value. We can say without doubt that this function is pure.

Parameters and return values are the first set of input and output a function can have. But there's a second set, which is usually more difficult to spot. Have a look at the following two functions:

```
<?php

function nextMessage(): string
{
    return array_pop($_SESSION['message']);
}

// A simple score updating method for a game
function updateScore(Player $player, int $points)
{
    $score = $player->getScore();
    $player->setScore($score + $points);
}
```

The first function has no obvious input. However, it's pretty clear that we get some data from the `$_SESSION` variable to create the output value, so we have one hidden input. We also have a hidden side-effect on the session because the `array_pop` method removes the message we just got from the list of messages.

The second method has no obvious output. But, updating the score of the player is clearly a side effect. Besides, the `$score` that we get from the player might be considered as a second input to the function.

In such simple and code examples, the hidden input and output is pretty easy to spot. However, it quickly becomes more difficult, especially in an object-oriented codebase. And make no mistake, anything hidden like that, even in the most obvious way, can have consequences such as:

- Increasing the cognitive burden. Now you have to think about what happens in the `Session` or `Player` classes.

- Test results can vary for identical input parameters, as some other state of the software has changed, leading to difficult to understand behaviors.
- The function signature, or API, is not clear about what you can expect from the functions, making it necessary to read the code or documentation for them.

The problem with those two simple looking functions is that they need to read and update the existing state of your program. It is not yet the topic of this chapter to show you ways to write them better, we will look at that in Chapter 6, *Real-life monads*.

For readers accustomed to Dependency Injection, the first function is using a static call and it can be avoided by injecting an instance of the Session variable. Doing so will solve the hidden input issue, but the fact that we modify the state of the `$_SESSION` variable remains as a side-effect.

The remainder of this chapter will try to teach you how to spot impure functions and why they are important, both for functional programming and code quality in general

For the rest of this book, we will use the terms **side cause** for hidden inputs, and **side effects** for hidden output. This dichotomy is not always used, but I think it helps with being able to describe with more accuracy when we are speaking about a hidden dependency or a hidden output of the code we will discuss.

Although a broader concept, available functional literature might use the term **free variable** to refer to side causes. A Wikipedia article about the topic states the following:

> *In computer programming, the term free variable refers to variables used in a function that are not local variables nor parameters of that function. The term non-local variable is often a synonym in this context.*

Given this definition, variables passed using the use keyword to a PHP closure could be called a free variable; this is why I prefer using the term side cause to clearly separate the two.

Pure functions

Let's say you have the function signature function `getCurrentTvProgram (Channel $channel)`. Without any indication of the purity of the function, you have no idea of the complexity that may be hidden behind such a function.

You will probably get the program that is actually playing on the given channel. What you don't know is whether the function checked if you are logged into the system. Maybe there's some kind of database update for analytic purposes. Maybe the function will return an exception because the log file is in a read-only state. You cannot know for sure, and all of those are side causes or side effects.

Regarding all the complexity associated with those hidden dependencies, you are faced with three options:

- Dive deep down into the documentation or code to understand all that is happening
- Make the dependencies apparent
- Do nothing and pray for the best

The last option is clearly better in the really short term, but you might get bitten real hard. The first option might seem better, but what about your colleague who will also need to use this function somewhere else in the application, will they need to follow the same path as you?

The second option is probably the most difficult one, and it will require tremendous effort in the beginning because we are not at all accustomed to doing it this way. But the benefits will start to pile up as soon as you have finished. And it gets a lot easier with experience.

What about encapsulation?

Encapsulation is about hiding implementation detail. Purity is about making hidden dependencies known. Both are useful, good practices and they aren't in any kind of conflict. You can achieve both if you are

careful enough and this is usually what functional programmers strive for. They like clean, simple solutions.

To explain this in simple terms:

- Encapsulation is about hiding internal implementation
- Avoiding side causes is about making external inputs known
- Avoiding side effects is about making external changes known

Spotting side causes

Let's get back to our `getCurrentTvProgram` function. The implementation that follows isn't pure, can you spot why?

To help you a bit, I will tell you that what we've learned so far about pure functions implies that they always return the same result when called with the same arguments:

```
<?php

function getCurrentTvProgram(Channel $channel ): string

{
    // let's assume that getProgramAt is a pure method.
    return $channel->getProgramAt(time());
}
```

Got it? Our suspect is the call to the `time()` method. Because of it, if you call the function at a different time, you will get different results. Let's fix this:

```
<?php

functiongetTvProgram(Channel $channel, int $when): string
{
    return $channel->getProgramAt($when);
}
```

Not only is our function now pure, which is clearly an achievement in itself, we have just gained two benefits:

- We can now get the program for any time of the day as implied by

the name change
- The function can now be tested without having to use some kind of magic trick to change the current time

Let's quickly look at some other examples of side causes. Try to spot the issue yourself as an exercise before reading:

```
<?php

$counter = 0;

function increment()
{
    global $counter;
    return ++$counter;
}

function increment2()
{
    static $counter = 0;
    return ++$counter;
}

function get_administrators(EntityManager $em)
{
    // Let's assume $em is a Doctrine EntityManager allowing
    // to perform DB queries
    return $em->createQueryBuilder()
              ->select('u')
              ->from('User', 'u')
              ->where('u.admin = 1')
              ->getQuery()->getArrayResult();
}

function get_roles(User $u)
{
    return array_merge($u->getRoles(), $u->getGroup()-
>getRoles());
}
```

The use of the `global` keyword makes it pretty obvious that the first function uses some variable from the global scope, thus making the function impure. The key takeaway from this example is that PHP scoping rules work to our advantage. Any function where you can spot

this keyword is most probably impure.

The static keyword in the second example is a good indicator that we might try to store a state between function calls. In this example, it is a counter that is incremented at each run. The function is clearly impure. However, contrary to the `global` variable, the use of the `static` keyword might only be a way to cache data between calls, so you will have to check why it is used before drawing a conclusion.

The third function is without a doubt impure because some database access is made. You might ask yourself how to get data from a database or the user if you are only allowed pure functions. The sixth chapter will dig deeper into this subject if you want to write purely functional code. If you can't or won't be functional all the way, I suggest you regroup your impure calls as much as possible and then try to call only pure functions from there to limit the place where you have side cause and side effects.

Concerning the fourth function, you cannot tell if it is pure just by looking at it. You will have to look at the code of the methods that are called. This is what you will encounter in most cases, a function calling other functions and methods, which you will also have to read in order to determine purity.

Spotting side effects

Usually a spotting side effects is a bit easier than spotting side causes. Anytime you change a value that will have visible effects on the outside, or call another function in doing so, you are creating a side effect.

If we go back to our two `increment` functions previously defined, what would you say about them? Consider the following code:

```
<?php

$counter = 0;

function increment()
{
```

```
    global $counter;
    return ++$counter;
}

function increment2()
{
    static $counter = 0;
    return ++$counter;
}
```

The first one clearly has a side effect on the global variable. But what about the second version? The variable itself is not accessible from the outside, so could we consider that the function is free of side-effects? The answer is no. Since the change implies that a following call to the function will return another value, this also qualifies as a side effect.

Let's look at some functions to see if you can spot the side effects:

```
<?php

function set_administrator(EntityManager $em, User $u)
{
    $em->createQueryBuilder()
       ->update('models\User', 'u')
       ->set('u.admin', 1)
       ->where('u.id = ?1')
       ->setParameter(1, $u->id)
       ->getQuery()->execute();
}

function log_message($message)
{
    echo $message."\n";
}

function updatePlayers(Player $winner, Player $loser, int
$score)
{
    $winner->updateScore($score);
    $loser->updateScore(-$score);
}
```

The first function obviously has a side effect because we update a value in the database.

The second method prints something to the screen. Usually this is considered a side effect because the function has an effect on something outside its scope, in our case, the screen.

Finally, the last function probably has side effects. This is a good, educated guess based on the name of the methods. But we can't say for sure until we've seen the code of the methods to verify it. As when spotting side causes, you will often have to dig a little deeper than just the one function in order to ascertain if it causes side effects or not.

What about object methods?

In a purely functional language, as soon as you need to change a value inside an object, an array or any kind of collection, you will in fact return a copy with the new value. This means any method, such as the `updateScore` method, for example, will not modify an inner property of the object, but will return a new instance with the new score.

This may not seem practical at all, and given the possibilities offered by PHP out of the box, I agree with you. However, we will see that there are some functional techniques that really help to manage this.

Another option would be to decide that the instance is not the same value after a change. In a way, this is already the case with PHP. Consider the following example:

```
<?php
class Test
{
    private $value;
    public function __construct($v)
    {
        $this->set($v);
    }

    public function set($v) {
        $this->value = $v;
    }
}
```

```
function compare($a, $b)
{
    echo ($a == $b ? 'identical' : 'different')."\n";
}

$a = new Test(2);
$b = new Test(2);

compare($a, $b);
// identical

$b->set(10);
compare($a, $b);
// different

$c = clone $a;
$c->set(5);
compare($a, $c);
```

When doing a simple equality comparison between two objects, PHP considers the inner value and not the instances themselves to make the comparison. It's important to note that as soon as you use a strict comparison (such as, using the `===` operator), PHP verifies that both variables hold the same instance, returning the `'different'` string in all three cases.

However, this is incompatible with the idea of referential transparency, which we will discuss later in this chapter.

Closing words

As we tried to show in the preceding examples, trying to determine if a function is pure or not can be tricky in the beginning. But as you start to get a feel for it, you will be a lot quicker and comfortable.

The best course of action to check whether a function is pure is to verify the following :

- The use of the global keyword is a dead give away
- Check if you use any value that is not a parameter of the function itself

- Verify that all functions called by yours are also pure
- Any access to outside storage is impure (database and files)
- Pay special attention to functions whose return value depends on an outside state (`time`, `random`)

Now that you know how to spot those impure functions, you might be wondering how to make them pure. There is no easy answer to this request sadly. The following chapters will try to give recipes and patterns to avoid impurity.

Immutability

We say a variable is immutable if, once it has been assigned a value, you cannot change its content. After function purity, this is the second most important thing about functional programming.

In some academic languages such as **Haskell**, you cannot declare variables at all. Everything has to be a function. Since all those functions are also pure, this means you have immutability for free. Some of these languages offers some kind of syntactic sugar that resembles variable declaration to avoid the potential tediousness of always declaring functions.

Most functional languages let you only declare immutable variables or constructs that serves the same purpose. This means you have a way of storing values but it is impossible to change the value after the initial assignment. There are also languages that let you choose what you want for each variable. In Scala, for example, you have the `var` keyword to declare traditional mutable variables and the `val` keyword to declare immutable variables.

Most languages however, as is the case for PHP, have no notion of immutability for variables.

Why does immutability matter?

First of all, it helps to reduce cognitive burden. It's already quite hard to keep in mind all the variables involved in an algorithm. Without immutability you also need to remember all value changes. It's a lot easier for the human mind to associate a value to a particular label (that is the variable name). If you can be sure that the value won't change, it will be a lot easier to reason about what is happening.

Also, if you have some global state you can't get rid of, as long as it is immutable, you can just note the values on a piece of paper near you and keep it for reference. Whatever happens during execution, what is

written will always be the current state of the program, meaning you don't have to fire up a debugger or echo the variable to ensure that the value has not changed.

Imagine that you pass an object to a function. You don't know whether the function is pure or not, meaning the object properties could be changed. This introduces worry in your mind and distracts you from your line of thought. The fact that you have to ask yourself if an internal state has changed, reduces your ability to reason about your code. If your object is immutable, you can be 100% assured that it is exactly as it was before, speeding up your understanding of what is happening.

You also have advantages linked to thread safety and parallelization. If all your state is immutable, is is much easier to ensure your program will be able to run on multiple cores or computers at the same time. Most concurrency issues happens because some thread modified a value without correctly synchronizing with other threads. This leads to inconsistency between them, and often, error in computations. If your variables are immutable, as long as all threads were sent the same data, this scenario is a lot less likely to happen. This is however not really useful as PHP is primarily used in non-threaded scenarios.

Data sharing

Another benefit of immutability is that when it is enforced by the language itself, the compiler can perform an optimization called **data sharing**. Since PHP does not support this yet, I will only present it in a few words.

Data sharing is the process of sharing a common memory location for multiple variables containing the same data. This allows for smaller memory footprints, and "copying" data from one variable to another with almost no cost at all.

For example, imagine the following piece of code:

```
<?php
```

```
//let's assume we have some big array of data
$array= ['one', 'two', 'three', '...'];

$filtered = array_filter($array, function($i) { /* [...] */ });
$beginning = array_slice($array, 0, 10);
$final = array_map(function($i) { /* [...] */ }, $array);
```

In PHP, each new variable will be a new copy of the data. Meaning we have a memory and time cost that could become problematic the bigger our array is.

A functional language might, using clever techniques, only store the data once in memory and then describe using another mean which part of the data each variable contains. This will still require some computation, but with big structures you will gain a lot of memory and time.

Such optimizations are also implementable in non-immutable languages. But it's often not done because you have to keep track of each write access to each variable to ensure data coherence. The implied complexity for the compiler is thought to outweigh the benefits of such an approach.

However, the time and memory penalty is not big enough in PHP to warrant avoiding using immutability. PHP has a pretty good garbage collector, meaning the memory is cleaned up pretty efficiently when an object is not used anymore. Also we often work with relatively small data structures, meaning the creation of nearly identical data is quite fast.

Using constants

You could use constants and class constants to have some kind of immutability, but they work only for scalar values. You currently have no way to use them for objects or more complex data structures. Since it's the only available option out-of-the-box, let's have a look anyway.

You can declare globally available constants containing any scalar value. Beginning with PHP 5.6, you can also store an array of scalar values

inside constants when using the `const` keyword and, since PHP 7, it also works with the define syntax.

Constant names must start with a letter or an underscore, not a number. Usually, constants are in full caps so they can be easily spotted. It is also discouraged to begin with an underscore as it may collide with any constant defined by the PHP core:

```
<?php

define('FOO', 'something');
const BAR=42;

//this only works since PHP 5.6
const BAZ = ['one', 'two', 'three'];

// the 'define' syntax for array work since PHP 7
define('BAZ7', ['one', 'two', 'three']);

// names starting and ending with underscores are discouraged
define('__FOO__', 'possible clash');
```

You can use the result of a function to populate the constant. This is possible only when using the defined syntax, however. If you use the `const` keyword you must use a scalar value directly:

```
<?php

define('UPPERCASE', strtoupper('Hello World !'));
```

If you try to access a constant that does not exist, PHP will assume that you are in fact trying to use the value as a string:

```
<?php

echo UPPERCASE;
//display 'HELLO WORLD !'

echo I_DONT_EXISTS;
//PHPNotice:  Use of undefined constant

I_DONT_EXISTS
//- assumed'I_DONT_EXISTS'
```

```
//display 'I_DONT_EXISTS'anyway
```

This can be pretty misleading, as the assumed string will evaluate to `true`, potentially breaking your code if you expected your constant to hold a `false` value.

If you want to avoid this pitfall, you can use the defined or constant function. Sadly, this will add a lot of verbosity:

```
<?php

echo constant('UPPERCASE');
// display 'HELLO WORLD !'

echo defined('UPPERCASE') ? 'true' : 'false';
// display 'true'

echo constant('I_DONT_EXISTS');
// PHP Warning:  constant(): Couldn't find constant
I_DONT_EXISTS
// display nothings as 'constant' returns 'null' in this case

echo defined('I_DONT_EXISTS') ? 'true' : 'false';
// display 'false'
```

PHP also allows you to declare constants inside of classes:

```
<?php

class A
{
    const FOO='some value';

    public static function bar()
    {
        echo self::FOO;
    }
}

echo A::FOO;
// display 'some value'

echo constant('A::FOO');
// display 'some value'
```

```
echo defined('A::FOO') ? 'true' : 'false';
// display 'true'

A::bar();
// display 'some value'
```

Sadly, you can only use scalar values directly when doing so; there is no way to use the return value of a function, as is the case with the `define` keyword:

```
<?php

class A
{
    const FOO=uppercase('Hello World !');
}

// This will generate an error when parsing the file :
// PHP Fatal error:  Constant expression contains invalid
operations
```

However, beginning with PHP 5.6, you can use any scalar expression or previously declared constants with the `const` keyword:

```
<?php

const FOO=6;

class B
{
    const BAR=FOO*7;
    const BAZ="The answer is ": self::BAR;
}
```

There is also one other fundamental difference between constants and variables besides their immutability. The usual scoping rule does not apply. You can use a constant anywhere in your code as soon as it is declared:

```
<?php

const FOO='foo';
$bar='bar';
```

```
function test()
{
    // here FOO is accessible
    echo FOO;

    // however, if you want to access $bar, you have to use
    // the 'global' keyword.
    global $bar;
    echo $bar;
}
```

At the time of writing, PHP 7.1 is still in the beta phase. The release is planned at the end of fall 2016. This new version will introduce class constants visibility modifiers:

```
<?php

class A
{
    public const FOO='public const';
    protected const BAR='protected const';
    private const BAZ='private const';
}

// public constants are accessible as always
echo A::FOO;

// this will however generate an error
echo A::BAZ;
// PHP Fatal error: Uncaught Error: Cannot access private const
A::BAR
```

A final word of warning. Although they are immutable, constants are global, and this makes them a state of your application. Any function using a constant is de facto impure, so you should use them with caution.

An RFC is on its way

As we just saw, constants, are at best, a wooden leg when it comes to immutability. They're quite alright to store simple information like the number of items we want displayed per page. But as soon as you want to have some complex data structures you will be stuck.

Fortunately, members of the PHP core team are well aware that immutability is important and there is currently some work being done on an RFC to include it at the language level (https://wiki.php.net/rfc/immutability).

For those not privy to the process involved for new PHP features, a **Request for Comment** (**RFC**), is a proposition from on the core team members to add something new to PHP. The proposition first gets through a draft phase, where it is written and some example implementation is done. Afterwards, there is a discussion phase where other people can give advice and recommendation. Finally, a vote occurs to decide whether the feature will be included in the next PHP version.

At the time of writing, the *Immutable classes and properties* RFC is still in draft phase. There was no real argument either for or against it. Only time will tell if it is accepted or not.

Value objects

From https://en.wikipedia.org/wiki/Value_object:

> *In computer science, a value object is a small object that represents a simple entity whose equality is not based on identity: i.e. two value objects are equal when they have the same value, not necessarily being the same object.*
>
> *[...]*
>
> *Value objects should be immutable: this is required for the implicit contract that two value objects created equal, should remain equal. It is also useful for value objects to be immutable, as client code cannot put the value object in an invalid state or introduce buggy behavior after instantiation.*

Since there is no mean to obtain real immutability in PHP, it is often achieved by having private properties and no setter on the class. Thus

forcing the developer to create a new object when they want to modify a value. The class can also provide utility methods to ease the creation of new objects. Let's look at a short example:

```
<?php

class Message
{
    private $message;
    private $status;

    public function __construct(string $message, string
$status)
    {
        $this->status = $status;
        $this->message = $message;
    }

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

    public function getStatus()
    {
        return $this->status;
    }

    public function equals($m)
    {
        return $m->status === $this->status &&
               $m->message === $this->message;
    }

    public function withStatus($status): Message
    {
        $new = clone $this;
        $new->status = $status;
        return $new;
    }
}
```

This kind of pattern can be used to create data entities that are immutable from the point of view of the data consumer. However, you will have to take special care to guarantee that all the methods on the

class do not break the immutability; otherwise all your efforts will be moot.

Besides immutability, using value objects has other benefits as well. You can add some business or domain logic inside the object, thus keeping everything related in the same place. Also, if you use them instead of arrays, you can:

- Use them as type hint instead of simply array
- Avoid any possible error due to a misspelled array key
- Enforce the presence or format of some items
- Provide methods that format your values for different context

A common use of value objects is to store and manipulate money related data. You can have a look at http://money.rtfd.org which is a great example of how to efficiently use them.

Another use of value objects for a really important piece of code is the **PSR-7: "HTTP message interfaces"**. This standard introduced and formalized a way for frameworks and applications to manage HTTP requests and responses in an inter-operable way. All major frameworks either have core support or plugins available. I invite you to read their full rationale as to why you should use immutability for such an important part of the PHP ecosystem: http://www.php-fig.org/psr/psr-7/meta/#why-value-objects.

In essence, modeling HTTP messages as value objects ensures the integrity of the message state, and prevents the need for bi-directional dependencies, which can often go out of sync or lead to debugging or performance issues.

All in all, value objects are a good way to obtain some kind of immutability in PHP. You don't get all the benefits, especially those related to performance, but most of the cognitive burden is removed. Going further on this topic is out of the scope of this book; if you want to learn more, there is a dedicated website: http://www.phpvalueobjects.info/.

Libraries for immutable collections

If you want to go further down the path of immutability, there are at least two libraries that offer immutable collections: **Laravel Collections** and **immutable.php**.

Both these libraries harmonize the discrepancies regarding the parameters order for array-related PHP functions such as `array_map` and `array_filter`. They also provide the possibilities to work with any kind of `Iterable` or `Traversable`; easily contrary to most PHP functions which require a real array.

This chapter will only present the libraries quickly. Example of usage will be given in Chapter 3, *Functional Basis in PHP* so that they can be shown alongside other libraries that allow the performance of the same task. Also, we haven't yet covered in detail techniques such as mapping or folding, so examples might not be as clear as possible right now.

Laravel Collection

The Laravel framework contains a class called `Collection` to supersede PHP arrays. This class uses a simple array internally, but it can be created from any collection like variable using the collect helper function. It then proposes a lot of really useful methods to work with the data, mostly in a functional way. This is also a central part of Laravel, as **Eloquent**, the ORM, returns database entities as `Collection` instances.

If you are not using Laravel, but still want to benefit from this great library, you can use https://github.com/tightenco/collect, which is only the Collection part separated from the rest of the Laravel support package in order to remain small. You can also refer to the official documentation of the Laravel collection (https://laravel.com/docs/5.3/collections).

Immutable.php

This library defines the `ImmArray` class, which implements an immutable

array like collection.

The `ImmArray` class is a wrapper around the `SplFixedArray` class to fix some of the shortcomings of its API by providing methods for performance operation that you usually want to perform on collections. The advantage of using the `SplFixedArray` class behind the scenes is that the implementation is written in C and is really performant and memory efficient. You can refer to the GitHub repository for more insight on Immutable.php at https://github.com/jkoudys/immutable.php.

Referential transparency

An expression is said to be referentially transparent if you can substitute it by its output at any time without changing the behavior of your program. In order to do that for all expressions of your code base, all your functions have to be pure and all your variables have to be immutable.

What do we gain from referential transparency? Once again, it helps a lot with reducing cognitive burden. Let's imagine we have the following functions and data:

```
<?php

// The Player implementation is voluntarily simple for brevity.
// Obviously you would use immutable.php in a real project.
class Player
{
    public $hp;
    public $x;
    public $y;

    public function __construct(int $x, int $y, int $hp) {
        $this->x = $x;
        $this->y = $y;
        $this->hp = $hp;
    }
}

function isCloseEnough(Player $one, Player $two): boolean
{
    return abs($one->x - $two->x) < 2 &&
           abs($one->y - $two->y) < 2;
}

function loseHitpoint(Player $p): Player
{
    return new Player($p->x, $p->y, $p->hp - 1);
}

function hit(Player $p, Player $target): Player
{
```

```
    return isCloseEnough($p, $target) ?
        loseHitpoint($target) :
        $target;
}
```

Now let's simulate a really simple brawl between two people:

```
<?php

$john=newPlayer(8, 8, 10);
$ted =newPlayer(7, 9, 10);

$ted=hit($john, $ted);
```

All functions defined above are pure, and since we don't have mutable data structures, they are also referentially transparent by extension. Now, in order to better understand our piece of code, we can use a technique called **equational reasoning**. The idea is pretty simple, you simply substitute *equals for equals* to reason about code. In a way, it is like evaluating the code manually.

Let's start by inlining our `isCloseEnough` function. Doing so, our hit function can be transformed as such:

```
<?php

return abs($p->x - $target->x) < 2 && abs($p->y - $target->y) <
2 ?
    loseHitpoint($target) :
    $target;
```

Our data being immutable, we can now simply use the values as the following:

```
<?php

return abs(8 - 7) < 2 && abs(8 - 8) < 2 ?
    loseHitpoint($target) :
    $target;
```

Let's do some math:

```
<?php
```

```
return 1<2 && 0<2 ?
    loseHitpoint($target) :
    $target;
```

The condition clearly evaluates to true so we can keep only the right branch:

```
<?php

return loseHitpoint($target);
```

Let's keep at it with the remaining function call:

```
<?php

return newPlayer($target->x, $target->y, $target->hp-1);
```

Once again, we replace the values:

```
<?php

return newPlayer(8, 7, 10-1);
```

Finally, our initial function call becomes:

```
<?php

$ted = newPlayer(8, 7, 9);
```

By using the fact that you can replace a referentially transparent expression with its resulting value, we were able to reduce a relatively lengthy piece of code with multiple function calls to a simple object creation.

This ability applied to refactoring or understanding code is very useful. If you have trouble understanding some code and you know some part of it is pure, you can simply replace it with the result while you are trying to understand it. This will probably help you get to the heart of the matter.

Non-strictness or lazy evaluation

One of the great benefits of referential transparency is the possibility for a compiler or parser to evaluate values lazily. For example, Haskell allows you to have an infinite list defined by a mathematical function. The lazy nature of the language ensures that values of the list will be computed only when you need the value.

In the glossary, we defined non-strict languages as languages where evaluation happens lazily. In fact, there's a slight difference between laziness and non-strictness. If you are interested in the details, you can head to https://wiki.haskell.org/Lazy_vs._non-strict and read about it. For the purpose of this book, we will use those terms interchangeably.

You might ask yourself how this can be useful. Let's gloss over use cases.

Performance

By using lazy evaluation, you ensure that only the values that are needed are effectively computed. Let's have a look at a short and naive example to illustrate this benefit:

```
<?php

function wait(int $value): int
{
    // let's imagine this is a function taking a while
    // to compute a value
    sleep(10);
    return $value;
}

function do_something(bool $a, int $b, int $c): int
{
    if($a) {
        return $b;
    } else {
        return $c;
    }
}

do_something(true, sleep(10), sleep(8));
```

Since PHP does not perform lazy evaluation on function parameters, when calling `do_something` you will first have to wait two times 10 seconds before even starting to execute the function. If PHP were a non-strict language, only the value we need would have been computed, thus dividing by two the time needed. It gets even better, since the return value isn't even saved in a new variable, it might be possible to not execute the function at all.

There is one case where PHP performs a kind of lazy evaluation: Boolean operator short-circuits. When you have a series of Boolean operations, as soon as PHP can be certain of the outcome, it will stop the execution:

```
<?php

// 'wait' will never get called as those operators are short-
circuited

$a= (false && sleep(10));
$b = (true  || sleep(10));
$c = (false and sleep(10));
$d = (true  or  sleep(10));
```

We could rewrite our previous example to take advantage of that. But as you can see in the following example, it is at the expense of readability. Also, our example was really simple, not something you would encounter in real-life application code. Imagine doing the same for some complex function with multiple possible branches? This is shown in the following snippet:

```
<?php

($a && sleep(10)) || sleep(8);
```

There are also two bigger issues with the previous code:

- If, for any reason, the first call to sleep returns a false value, the second call will also be executed
- The return value of your methods will automatically be cast to Boolean

Code readability

When your variable and function evaluation are lazy, you can spend less time considering which is the best order of declaration, or even whether the data you are computing will be used at all. Instead, you can concentrate on writing readable code. Imagine a blogging application with lots of posts, tags, categories, and archived by year. Would you rather have to write custom queries for each page, or use lazy evaluation, demonstrated as follows:

```
<?php

// let's imagine $blogs is a lazily evaluated collection
// containing all the blog posts of your application order by
date
$posts = [ /* ... */ ];

// last 10 posts for the homepage
return $posts->reverse()->take(10);

// posts with tag 'functional php'
return $posts->filter(function($b) {
    return $b->tags->contains('functional-php');
})->all();

// title of the first post from 2014 in the category 'life'
return $posts->filter(function($b) {
    return $b->year == 2014;
})->filter(function($b) {
    return $b->category == 'life';
})->pluck('title')->first();
```

To be clear, this code would probably work just fine if we loaded all posts into `$posts`, but the performance would be pretty bad. However, if we had lazy evaluation and an ORM powerful enough, the database queries could be delayed to the last moment. At that time, we would know exactly the data we need and the SQL will be tailored for this exact page automatically, leaving us with easy to read code and great performance.

As far as I can tell, this idea is purely hypothetical. I am not currently

aware of any ORM powerful enough, even in the most functional languages, to attain this level of laziness. But wouldn't it be great if it were?

If you are wondering about the syntax used in the example, it is inspired by the API of the Laravel's Collection we were discussing earlier.

Infinite lists or streams

Lazy evaluation allows you to create infinite lists. In Haskell, to get the list of all positive integers, you can simply do `[1..]`. Then, if you want the first ten numbers, you can take `10 [1..]`. I admit this example isn't very exciting, but more complicated ones are more difficult to understand.

PHP supports generators since version 5.5. You can achieve something akin to infinite lists by using them. For example, our list of all positive integers is as follows:

```
<?php

function integers()
{
    $i=0;
    while(true) yield $i++;
}
```

However, there is at least one notable difference between the lazy infinite list and our generator. You can perform any operation you would normally perform on collections with the Haskell version-computing its length and sorting it, for example. Whereas our generator is an `Iterator` and if you try to use say `iterator_to_array` on it there's a good chance that your PHP process will hang until you run out of memory.

How can you compute the length of an infinite list or sort it you ask me? It is in fact pretty simple; Haskell will only compute list values until it has enough to perform its computation. Say we have the the condition `count($list) < 10` in PHP, even if you have an infinite list, Haskell will stop counting items as soon as you reach 10 because it will have an

answer for the comparison at that time.

Code optimization

Have a look at the next piece of code and try deciding which is faster:

```
<?php

$array= [1, 2, 3, 4, 5, 6 /* ... */];

// version 1
for($i = 0; $i < count($array); ++$i) {
    // do something with the array values
}

// version 2
$length = count($array);
for($i = 0; $i < $length; ++$i) {
    // do something with the array values
}
```

Version 2 should be a lot faster. Because you only compute the length of the array once, whereas in version 1, PHP has to compute the length each time it verifies the condition for the for loop. This example is pretty simple, but there are some cases where such a pattern is harder to spot. If you have referential transparency, this does not matter. The compiler can perform this optimization on its own. Any referentially transparent computation can be moved around without changing the result of the program. This is possible because we have the guarantee that the execution of each function does not depend on a global state. Thus, moving computation around to achieve better performance is possible without changing the outcome.

Another possible improvement is performing common sub expression elimination or CSE. Not only can the compiler move part of the code more freely, it can also transform some operations that share a common computation to use an intermediary value instead. Imagine the following code:

```
<?php
```

```
$a= $foo * $bar + $u;
$b = $foo * $bar * $v;
```

If computing `$foo * $bar` has a big cost, the compiler could decide to transform it by using an intermediary value:

```
<?php

$tmp= $foo * $bar;
$a = $tmp + $u;
$b = $tmp * $v;
```

Again, this is quite a simple example. This kind of optimization could be performed on the whole span of the code base.

Memoization

Memoization is a technique where you cache the results of a function for a given set of parameters so that you don't have to perform it again on the next call. We will see this in detail in Chapter 8, *Performance Efficiency*. For now, let me just say that if your language only possesses referentially transparent expressions, it can perform memoization automatically when needed.

This means it can decide, based on the frequency of calls and various other parameters, whether it's worth memoizing a function automatically without any intervention or hint from the developer.

PHP in all that?

Why bother with pure functions, immutability and, ultimately, referential transparency if PHP developers can only benefit from a small number of its advantages?

First of all, as with the RFC for immutability, things are going in the right direction. This means that, eventually, the PHP engine will start to incorporate some of those advanced compiler techniques. When this happens, if your codebase already uses those functional techniques, you will have a huge performance boost.

Secondly, in my opinion, the major benefit of all that is the reduced cognitive burden. Sure, it takes some time to get used to this new style of programming. But once you have practiced a bit, you will quickly discover that your code is easier to read and reason about. The corollary being that your application will contain less bugs.

Finally, if you are willing to use some external libraries, or if you can cope with syntax that are not always well polished, you can already benefit from other improvements right now. Obviously, we won't be able to change the core of PHP to add the compiler optimization we were talking about earlier, but we will see in the following chapters how some of the benefits of referential transparency can be emulated.

Summary

This chapter contained a lot of theory. I hope you didn't mind too much. It was necessary to lay the foundation that will allow us to share a common vocabulary and also explain why the concepts are important. You are now well aware of what purity and immutability are and you learned some tricks to spot impure functions. We also discussed how those two properties lead to something called referential transparency what the benefits are.

We also learned that, sadly, PHP does not support most of the benefits natively. However, the key takeaway is that using a functional approach reduces the cognitive burden of understanding your code, thus making it easier to read. The net benefit being that now your code will be easier to maintain and refactor and you can find and fix bugs quickly. Usually, pure functions are also easier to test, which also results in fewer bugs.

Now that we have the theoretical basis well discussed, the next chapter will focus on techniques that will help us achieve purity and immutability in our software.

Chapter 3. Functional Basis in PHP

After covering functions in PHP in the first chapter, followed by theoretical aspects of functional programming in the second, we will finally start to write real code. We will start with the available functions in PHP that allow us to write functional code. Once the basic techniques are well understood, we will move on to various libraries that will help us throughout the book.

In this chapter, we will cover the following topics:

- Mapping, folding, reducing, and zipping
- Recursion
- Why exceptions break referential transparency
- A better way of handling errors using the Maybe and Either types
- Functional libraries available for PHP

General advice

In the previous chapters, we described the important properties a functional application must have. However, we never really discussed how it can be achieved. Besides the various techniques we will learn about later on, there are a few simple pieces of advice that could really help you right away.

Making all inputs explicit

We discussed purity and hidden inputs, or side causes, a lot in the previous chapter. By now, it should be pretty clear that all the dependencies of your functions should be passed on as parameters. This advice, however, goes a bit further.

Avoid passing objects or complex data structure to your functions. Try to limit your input to what is necessary only. Doing so will make the

scope of your function easier to understand and it will ease determining how the function operates. It also has the following benefits:

- It will be easier to call
- Testing it will require stubbing less data

Avoiding temporary variables

As you may have gotten to understand, state is evil-particularly global state. However, local variables are a kind of local state. As soon as you start peppering your code with them, you are slowly opening the can of worms. This is especially true in a language such as PHP, where all variables are mutable. What happens if the value changes along the way?

Each time you declare a variable, you have to keep its value in mind if you are to understand how the remainder of the code works. This greatly increases the cognitive burden. Also, as PHP is dynamically typed, a variable can be reused with totally different data.

When using a temporary variable, there is always the risk that it gets modified somehow or reused without it being evident, leading to bugs that are difficult to debug.

In nearly all cases, using a function is better than a temporary variable. Functions allow for the same benefits:

- Improving readability by naming intermediate results
- Avoiding repeating yourself
- Caching the result of a lengthy operation (this requires the use of memoization, which we will discuss in Chapter 8, *Performance Efficiency*)

The added cost of calling a function is usually minimal enough to not tip the balance. Also, using functions instead of temporary variables means that you can then reuse those functions in other places. They could also make future refactoring easier and they improve the separation of concerns.

As it can be expected with best practices, there are, however, some times when it's easier to use temporary variables. For example, if you need to store a return value that will be used just after in a short function so that you can keep the line length comfortable, don't hesitate to do so. The only thing that should be strictly forbidden is to use the same temporary variables to store various different bits of information.

Smaller functions

We already mentioned that functions are like building blocks. Usually, you want your building blocks to be versatile and sturdy. Both those properties are better enforced if you write small functions that only focus on doing one thing well.

If your function does too much, it is difficult to reuse. We will look at composing functions in the next chapter and how you can leverage all your small utility functions to create new ones with bigger reaches.

Also, it is easier to read smaller pieces of code and reason about them. The implications are simpler to understand and there are usually fewer edge cases, making the function easier to test.

Parameter order matters

Choosing the order of parameters of your functions does not seems like much, but in fact it matters a lot. Higher-order functions are a core feature of functional programming; this means that you will be passing a lot of functions around.

Those functions could be anonymous, in which case you might want to avoid having a function declaration as the middle parameter, for readability reasons. Optional parameters are also constrained to the end of the signature in PHP. As we will see, some functional constructs take functions that can have default values.

We will also dwell on this topic further in Chapter 4, *Compositing Functions*. When you chain multiple functions together, the first

parameter of each is the return value of the previous one. This means you will have to take special care when choosing which parameters go first.

The map function

The map, or `array_map` method in PHP, is a higher-order function that applies a given callback to all elements of a collection. The `return` value is a collection in the same order. A simple example is:

```
<?php

function square(int $x): int
{
    return $x * $x;
}
$squared = array_map('square', [1, 2, 3, 4]);
// $squared contains [1, 4, 9, 16]
```

We create a function that computes the square of the given integer and then use the `array_map` function to compute all the square values of a given array. The first parameter of the `array_map` function is any form of callable and the second parameter has to be a *real array*. You cannot pass an Iterator or an instance of Traversable.

You can also pass multiple arrays. Your callback will receive a value from each array:

```
<?php

$numbers = [1, 2, 3, 4];
$english = ['one', 'two', 'three', 'four'];
$french = ['un', 'deux', 'trois', 'quatre'];

function translate(int $n, string $e, string $f): string
{
    return "$n is $e, or $f in French.";
}
print_r(array_map('translate', $numbers, $english, $french));
```

This code will display:

```
Array
(
    [0] => 1 is one, or un in French.
    [1] => 2 is two, or deux in French.
```

```
    [2] => 3 is three, or trois in French.
    [3] => 4 is four, or quatre in French.
)
```

The longest array will determine the length of the result. Shorter arrays will be expanded with the null value so that they all have matching lengths.

If you pass null as a function, PHP will merge the arrays:

```
<?php

print_r(array_map(null, [1, 2], ['one', 'two'], ['un',
'deux']));
```

And the result:

```
Array
(
    [0] => Array
        (
            [0] => 1
            [1] => one
            [2] => un
        )
    [1] => Array
        (
            [0] => 2
            [1] => two
            [2] => deux
        )
)
```

If you pass only one array, the keys will be preserved; but if you pass multiple arrays, they will be lost:

```
<?php
  function add(int $a, int $b = 10): int
  {
      return $a + $b;
  }

  print_r(array_map('add', ['one' => 1, 'two' => 2]));
  print_r(array_map('add', [1, 2], [20, 30]));
```

And the result:

```
Array
(
    [one] => 11
    [two] => 12
)
Array
(
    [0] => 21
    [1] => 32
)
```

As a final note, it is sadly impossible to access the key of each item easily. Your callable can, however, be a closure so you are able to use any variable accessible from your context. Using this, you can map over the keys of your array and use a closure to retrieve the values like that:

```
$data = ['one' => 1, 'two' => 2];

array_map(function to_string($key) use($data) {
    return (str) $data[$key];
},
array_keys($data);
```

The filter function

The filter, or `array_filter` method in PHP, is a higher-order function that keeps only certain elements of a collection, based on a Boolean predicate. The `return` value is a collection that will only contain elements returning true for the predicate function. A simple example is:

```
<?php

function odd(int $a): bool
{
    return $a % 2 === 1;
}

$filtered = array_filter([1, 2, 3, 4, 5, 6], 'odd');
/* $filtered contains [1, 3, 5] */
```

We first create a function that takes a value and returns a Boolean. This function will be our predicate. In our case, we check whether an integer is an odd number. As with the `array_map` method, the predicate can be anything that is a `callable` and the collection must be an array. Be aware, however, that the parameter order is reversed; the collection comes first.

The callback is optional; if you don't give one, all elements which PHP will evaluate to false, like empty strings and arrays for example, will be filtered out:

```
<?php

$filtered = array_filter(["one", "two", "", "three", ""]);
/* $filtered contains ["one", "two", "three"] */

$filtered = array_filter([0, 1, null, 2, [], 3, 0.0]);
/* $filtered contains [1, 2, 3] */
```

You can also pass a third parameter that acts as a flag to determine whether you want to receive the key instead of the value, or both:

```
<?php
```

```
$data = [];
function key_only($key) {
    // [...]
}

$filtered = array_filter($data, 'key_only',
ARRAY_FILTER_USE_KEY);

function both($value, $key) {
    // [...]
}

$filtered = array_filter($data, 'both', ARRAY_FILTER_USE_BOTH);
```

The fold or reduce function

Folding refers to a process where you reduce a collection to a return value using a combining function. Depending on the language, this operation can have multiple names like fold, reduce, accumulate, aggregate, or compress. As with other functions related to arrays, the PHP version is the `array_reduce` function.

You may be familiar with the `array_sum` function, which calculates the sum of all the values in an array. This is, in fact, a fold and can be easily written using the `array_reduce` function:

```
<?php

function sum(int $carry, int $i): int
{
    return $carry + $i;
}

$summed = array_reduce([1, 2, 3, 4], 'sum', 0);
/* $summed contains 10 */
```

Like the `array_filter` method, the collection comes first; you then pass a callback and finally an optional initial value. In our case, we were forced to pass the initial value 0 because the default null is an invalid type for our function signature of int type.

The callback function has two parameters. The first one is the current reduced value based on all previous items, sometimes called **carry** or **accumulator**. The second one is the array element currently being processed. On the first iteration, the carry is equal to the initial value.

You don't necessarily need to use the elements themselves to produce a value. You could, for example, implement a naive replacement for `in_array` using fold:

```
<?php

function in_array2(string $needle, array $haystack): bool
```

```
{
    $search = function(bool $contains, string $item) use
($needle):bool
    {
        return $needle == $item ? true : $contains;
    };
    return array_reduce($haystack, $search, false);
}

var_dump(in_array2('two', ['one', 'two', 'three']));
// bool(true)
```

The reduce operation starts with the initial value false because we assume that the array does not contain our needle. This also allows us to nicely manage the case where we have an empty array.

Upon each item, if the item is the one we are searching for, we return true, which will be the new value passed around. If it does not match, we simply return the current value of the accumulator, which will be either `true` if we found the item earlier, or `false` if we did not.

Our implementation will probably be a tad slower than the official one because, no matter what, we have to iterate over the entire array before returning a result instead of being able to exit the function as soon as we encounter the searched item.

We could, however, implement an alternative to the max function where performances should be on par, because any implementation will have to iterate over all values:

```
<?php

function max2(array $data): int
{
    return array_reduce($data, function(int $max, int $i) : int
    {
        return $i > $max ? $i : $max;
    }, 0);
}

echo max2([5, 10, 23, 1, 0]);
// 23
```

The idea is the same as before, although using numbers instead of a Boolean value. We start with the initial `0`, our current maximum. If we encounter a bigger value, we return it so that it gets passed around. Otherwise, we keep returning our current accumulator, already containing the biggest value encountered so far.

As the max PHP functions works on both arrays and numbers, we could reuse it for our reducing. This would, however, bring nothing, as the original function can already operate directly on arrays:

```
<?php

function max3(array $data): int
{
    return array_reduce($data, 'max', 0);
}
```

Just to be clear, I don't recommend using those in production. The functions already in the language are better. Those are just for educational purposes to demonstrate the various possibilities of folding.

Also, I totally understand if those short examples do not seem better than a `foreach` loop, or any other more imperative approach, to implement those two functions. They have, however, a few advantages:

- If you are using PHP 7 scalar type hinting, the types are enforced for each item, making your software more robust. You can verify that by putting a string in the array used for the `max2` method.
- You can unit test the function that you are passing to the `array_reduce` method, or the `array_map` and `array_filter` functions for that matter, to ensure its correctness.
- You could distribute the reducing of a big array between multiple threads or network nodes if you have such an architecture. This would be a lot harder with a `foreach` loop.
- As shown with the `max3` function, this approach allows you to reuse existing methods instead of writing custom loops to manipulate data.

The map and filter functions using fold

For now, our `fold` only returned simple scalar values. But nothing prevents us from building more complex data structures. For example, we can implement the map and filter functions using `fold`:

```
<?php

function map(array $data, callable $cb): array
{
    return array_reduce($data, function(array $acc, $i) use
($cb) {
        $acc[] = $cb($i);
        return $acc;
    }, []);
}

function filter(array $data, callable $predicate): array
{
  return array_reduce($data, function(array $acc, $i)
use($predicate) {
      if($predicate($i)) {
          $acc[] = $i;
      }
      return $acc;
  }, []);
}
```

Again, those are mostly for the purposes of demonstrating that it is possible to return arrays with folding. The native functions are enough if you don't need to manipulate more complex collections.

As an exercise for the reader, try to implement the `map_filter` or the `filter_map` function if you prefer, and the `array_reverse` function. You can also try writing head and tail methods, which return the first, respectively last, element of an array and are often found in functional languages.

As you can see, folding is really powerful and the idea behind it is central to a lot of functional techniques. It is why I largely prefer to talk about fold rather than reduce, which I find a bit reductive, pun intended.

Before going further, make sure you understand how fold works, as it will make everything else much easier.

Folding left and right

Functional languages often implement two versions of fold, `foldl` and `foldr`. The difference is that the first folds from the left and the second from the right.

For example, if you have the array `[1, 2, 3, 4, 5]` and you want to compute its sum, you can have either `(((1 + 2) + 3) + 4) + 5` or `(((5 + 4) + 3) + 2) + 1`. If you have an initial value, it will always be the first value used in the computation.

If the operation you are applying to the values is commutative, both the left and right variants will produce the same results. The notion of commutative operation comes from mathematics and is explained in Chapter 7, *Functional Techniques and Topics*.

For languages allowing infinite lists, such as Haskell, depending on how the list is generated, one of the two folds could be able to compute a value and stop. Also, if the language implements tail call elimination, a topic that we will discuss in Chapter 7, *Functional Techniques and Topics*, choosing the right side to start the fold might avoid a stack overflow and allow the operation to finish.

As neither infinite list or tail call elimination is performed by PHP, there is, in my opinion, no reason to bother with the distinction. If you are interested, the `array_reduce` function folds from the left and implementing a function that does the same from the right should not be too complicated.

The MapReduce model

You may have already heard the name the **MapReduce** programming model. At first, it referred to a proprietary technology developed by Google but nowadays there are multiple implementations in a variety of languages.

Although the ideas behind MapReduce are inspired by the map and

reduce functions we just discussed, the concept is broader. It describes a whole model to process large datasets using parallel and distributed algorithms on a cluster.

Every technique you learn in this book could help you when implementing a MapReduce to analyze data. However, the topic is out of scope, so if you want to learn more, you can start with the Wikipedia page by visiting https://en.wikipedia.org/wiki/MapReduce.

Convolution or zip

Convolution, or more often zip is the process of combining each nth element of all given arrays. In fact, this is exactly what we did by passing null value to the `array_map` function before:

```
<?php

print_r(array_map(null, [1, 2], ['one', 'two'], ['un',
'deux']));
```

And the output:

```
Array
(
    [0] => Array
        (
            [0] => 1
            [1] => one
            [2] => un
        )
    [1] => Array
        (
            [0] => 2
            [1] => two
            [2] => deux
        )
)
```

It is important to note that if the arrays are of different lengths, PHP will use null as the padding value:

```
<?php

$numerals = [1, 2, 3, 4];
$english = ['one', 'two'];
$french = ['un', 'deux', 'trois'];

print_r(array_map(null, $numerals, $english, $french));
Array
(
    [0] => Array
        (
```

```
            [0] => 1
            [1] => one
            [2] => un
        )
    [1] => Array
        (
            [0] => 2
            [1] => two
            [2] => deux
        )
    [2] => Array
        (
            [0] => 3
            [1] =>
            [2] => trois
        )
    [3] => Array
        (
            [0] => 4
            [1] =>
            [2] =>
        )
)
```

Be aware that in most programming languages, including Haskell, Scala, and Python, the zip operation will, however, stop at the shortest array without padding any values. You can try to implement a similar function in PHP using, for example, the `array_slice` function to reduce all arrays to the same size before calling the `array_merge` function.

We can also perform the inverse operation by creating multiple arrays from an array of arrays. This process is sometimes called **unzip**. Here is a naive implementation which is missing a lot of checks to make it robust enough for production use:

```php
<?php

function unzip(array $data): array
{
    $return = [];

    $data = array_values($data);
    $size = count($data[0]);
```

```
    foreach($data as $child) {
        $child = array_values($child);
        for($i = 0; $i < $size; ++$i) {
            if(isset($child[$i]) && $child[$i] !== null) {
                $return[$i][] = $child[$i];
            }
        }
    }

    return $return;
}
```

You could use it like this:

```
$zipped = array_map(null, $numerals, $english, $french);

list($numerals2, $english2, $french2) = unzip($zipped);

var_dump($numerals == $numerals2);
// bool(true)
var_dump($english == $english2);
// bool(true)
var_dump($french == $french2);
// bool(true)
```

Recursion

In the academic sense, recursion is the idea of dividing a problem into smaller instances of the same problem. For example, if you need to scan a directory recursively, you first scan the starting directory and then scan its children and the children's children. Most programming languages support recursion by allowing a function to call itself. This idea is often what is described as recursion.

Let's see how we can scan a directory by using recursion:

```
<?php

function searchDirectory($dir, $accumulator = []) {
    foreach (scandir($dir) as $path) {
        // Ignore hidden files, current directory and parent
directory
        if(strpos($path, '.') === 0) {
            continue;
        }

        $fullPath = $dir.DIRECTORY_SEPARATOR.$path;

        if(is_dir($fullPath)) {
            $accumulator = searchDirectory($path,
$accumulator);
        } else {
            $accumulator[] = $fullPath;
        }
    }
    return $accumulator;
}
```

We start by using the `scandir` function to obtain all files and directories. Then, if we encounter a child directory, we call the function on it again. Otherwise, we simply add the file to the accumulator. This function is recursive because it calls itself.

You could write this using control structures, but as you don't know in advance what the depth of your folder hierarchy is, the code will probably be a lot messier and harder to understand.

Some books and tutorials use the Fibonacci sequence, or computing a factorial as recursion examples but, to be fair, those are quite poor, as they are better implemented using a traditional `for` loop for the second, and compute terms in advance for the first.

Instead, let's wrap our heads around a more interesting challenge, the *Hanoi Towers*. For those unaware of this game, the traditional version features three rods with discs of different sizes stacked in top of one another, the smallest on the top. At the beginning of the game, all discs are on the leftmost rod and the goal is to bring them to the rightmost one. The game obeys the following rules:

- Only one disc can move at a time
- Only the topmost disc of a rod can be moved
- A disc cannot be placed on top of a smaller disc

The setup for this game looks like the following:

If we want to solve the game, the larger disc must be placed first on the last rod. In order to do that, we need to move all other discs to the

middle rod first. Following this line of reasoning, we can draw three big steps that we must achieve:

1. Move all discs but the bigger one to the middle.
2. Move the large disc to the right.
3. Move all discs on top of the large one.

Steps 1 and *3* are smaller versions of the initial problem. Each of those steps can, in turn, be reduced to a smaller version until we have only one disc to move-the perfect situation for a recursive function. Let's try implementing that.

To avoid cluttering our function with variables related to the rods and discs, we will assume the computer will give orders to someone making the moves. In our code, we will also assume the largest disc is number 1, smaller discs having larger numbers:

```
<?php

function hanoi(int $disc, string $source, string $destination,
string $via)
{
    if ($disc === 1) {
        echo("Move a disc from the $source rod to the
$destination  rod\n");
    } else {
        // step 1 : move all discs but the first to the "via"
rod         hanoi($disc - 1, $source, $via, $destination);
        // step 2 : move the last disc to the destination
        hanoi(1, $source, $destination, $via);
        // step 3 : move the discs from the "via" rod to the
destination
        hanoi($disc - 1, $via, $destination, $source);
    }
}
```

On using the `hanoi(3, 'left', 'right', 'middle')` input for the three discs, we get the following output:

```
Move a disc from the left rod to the right rod
Move a disc from the left rod to the middle rod
Move a disc from the right rod to the middle rod
```

```
Move a disc from the left rod to the right rod
Move a disc from the middle rod to the left rod
Move a disc from the middle rod to the right rod
Move a disc from the left rod to the right rod
```

It takes a while to think in terms of recursion instead of using a more traditional loop, and obviously recursion is not a silver bullet that is better for all problems you are trying to solve.

Some functional languages have no loop structures at all, forcing you to use recursion. This is not the case with PHP, so let's use the right tool for the job. If you can think of the problem as a combination of smaller similar issues, usually it will be easy to use recursion. For example, trying to find an iterative solution to the *Towers of Hanoi* requires a lot of careful thinking. Or you could try to rewrite the directory scanning function using only loops to convince yourself.

Some other areas where recursion is useful are:

- Generating the data structure for a menu with multiple levels
- Traversing an XML document
- Rendering a series of CMS components that could contain child components

A good rule of thumb is to try recursion when your data has a tree-like structure with a root node and children.

Although often easier to read, once you have gotten to grip with it, recursion comes with a memory cost. In most applications, you should not encounter any difficulties, but we will discuss the topic further in Chapter 10, *PHP Frameworks and FP*, and present some methods to avoid those issues.

Recursion and loops

Some functional languages, such as Haskell, do not have any loop structure. This means the only way to iterate over a data structure is to use recursion. Although it is discouraged in the functional world to use a for loop due to all issues that arise when you can modify the loop index,

there are no real dangers to using a `foreach` loop, for example.

For the sake of completeness, here are some ways you can replace a loop with a recursive call if you want to try it or need to understand code written in another language without a loop construct.

Replace a `while` loop:

```
<?php

function while_iterative()
{
    $result = 1;
    while($result < 50) {
        $result = $result * 2;
    }
    return $result;
}

function while_recursive($result = 1, $continue = true)
{
    if($continue === false) {
        return $result;
    }
    return while_recursive($result * 2, $result < 50);
}
```

Or a `for` loop:

```
<?php

function for_iterative()
{
    $result = 5;

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

    return $result;
}

function for_recursive($result = 5, $i = 1)
{
    if($i >= 10) {
```

```
        return $result;
    }

    return for_recursive($result * $i, $i + 1);
}
```

As you can see, the trick is to use function parameters to pass the current state of the loop to the next recursion. In the case of a while loop, you pass the result of the condition and when you emulate a for loop, you pass the loop counter. Obviously, the current state of computation must also always be passed around.

Usually, the recursion itself is done in a helper function to avoid cluttering the signature with optional parameters used to perform the loop. In order to keep the global namespace clean, this helper is declared inside the original function. Here is an example:

```
<?php

function for_with_helper()
{
    $helper = function($result = 5, $i = 1) use(&$helper) {
        if($i >= 10) {
            return $result;
        }

        return $helper($result * $i, $i + 1);
    };

    return $helper();
}
```

Notice how you need to pass the variable containing the function by reference with the `use` keyword. This is due to a fact we already discussed. The variable passed to the closure is bound at the declaration time, but when the function is declared, the assignment has not happened yet and the variable is empty. However, if we pass the variable by reference, it will be updated once the assignment is complete and we will be able to use it as a callback inside the anonymous function.

Exceptions

Error management is one of the toughest problems you face when writing software. It is often difficult to decide which piece of code should treat the error. Do it in the low-level function and you might not have access to the facilities to display an error message or enough context to decide the best course of action. Do it higher up and this might cause havoc in your data or put the application into an unrecoverable state.

The usual way to manage errors in OOP codebases is to use exceptions. You throw an exception in your library or utility code and you catch it whenever you are ready to manage it as you want.

Whether exception throwing and catching can be considered side effects or side causes is a matter for debate even among academics. There's a variety of points of view. I don't want to bore you with rhetorical arguments, so let's stick to some points nearly everyone agrees upon:

- An exception thrown by any **external source** (database access, filesystem errors, unavailable external resource, invalid user input, and so on) is inherently impure because accessing those sources is already a side cause.
- An exception thrown due to a **logical error** (index out of bound, invalid types or data, and so on) is, usually, considered pure as it can be considered a valid `return` value for the function. The exception must, however, be clearly documented as a possible outcome.
- Catching an exception breaks referential transparency and thus makes any function with a catch block impure.

The first two statements should be fairly easy to understand, but what about the third one? Let us start our demonstration with a short piece of code:

```
<?php
function throw_exception()
{
```

```
    throw new Exception('Message');
}

function some_function($x)
{
    $y = throw_exception();
    try {
        $z = $x + $y;
    } catch(Exception $e) {
        $z = 42;
    }

    return $z;
}

echo some_function(42);
// PHP Warning: Uncaught Exception: Message
```

It's easy to see that our call to the `some_function` function will result in an uncaught exception because the call to the `throw_exception` function is outside the `try ... catch` block. Now, if we apply the principles of referential transparency, we should be able to replace the `$y` parameter in the addition by its value. Let's try that:

```
<?php

try {
    $z = $x + throw_exception();
} catch(Exception $e) {
    $z = 42;
}
```

What is the value of the `$z` parameter now and what will our function return? Contrary to before, we will now have a return value of `42`, clearly changing the outcome of calling our function. By simply trying to apply equation reasoning, we just proved that catching an exception can break referential transparency.

What good are exceptions if you cannot catch them? Not much; this is why we will refrain from using them throughout the book. You could, however, consider them as a side effect and then apply the techniques we will see in Chapter 6, *Real-Life Monads*, to manage them. Haskell,

for example, allows throwing exceptions as long as they are caught using the IO Monad.

Another issue is cognitive burden. As soon as you use them, you cannot know for sure when they will be caught; they might even be displayed directly to the end user. This breaks the ability to reason about a piece of code on its own as you now have to think of what will happen higher up.

This issue is usually why you hear advice such as *Use exceptions for errors only, not flow control*. This way, you can at least be sure that your exception will be used to display some kind of error instead of wondering in which state you put the application.

PHP 7 and exceptions

Even if we are discussing exceptions mostly in a negative light, let me take this opportunity to present the improvements that have been made in the new PHP version concerning the topic.

Before, some type of errors would generate fatal errors or errors which would stop the execution of the script and display an error message. You were able to use the `set_error_handler` exception to define a custom handler for non-fatal errors and eventually continue execution.

PHP 7.0 introduces a `Throwable` interface, which is a new parent for the exception. The `Throwable` class also a new child called the `Error` class, which you can use to catch most of the errors that you weren't able to manage before. There are still some errors, such as parsing errors, which you can obviously not catch, as it means your whole PHP file is somehow invalid.

Let's demonstrate this with a piece of code that tries to call a non-existing method on an object:

```
<?php
class A {}

$a = new A();
```

```
$a->invalid_method();

// PHP Warning: Uncaught Error: Call to undefined method
A::invalid_method()
```

If you are using PHP 5.6 or lower, the message will say something along the lines of:

```
Fatal error: Call to undefined method A::invalid_method()
```

Using PHP 7.0, however, the message will be (emphasis is mine):

```
Fatal error: Uncaught Error: Call to undefined method
A::invalid_method()
```

The difference being that PHP informs you that this is an uncaught error. This means you can now catch it using the usual `try ... catch` syntax. You can catch the `Error` class directly, or if you want to be broader and catch any possible exception, you can use the `Throwable` interface. However, I discourage this as you will lose the information about which error you have exactly:

```
<?php class B {}

$a = new B();

try {
    $a->invalid_method();
} catch(Error $e) {
    echo "An error occured : ".$e->getMessage();
}
// An error occured : Call to undefined method
B::invalid_method()
```

Also interesting for us, the `TypeError` parameter is a child of the `Error` class which is raised when a function is called with parameters of the wrong type or the return type is wrong:

```
<?php
function add(int $a, int $b): int
{
    return $a + $b;
}
```

```
try {
    add(10, 'foo');
} catch(TypeError $e) {
    echo "An error occured : ".$e->getMessage();
}
// An error occured : Argument 2 passed to add() must be of the
type integer, string given
```

For those wondering why a new interface was created alongside the new `Error` class, it is mostly for two reasons:

- To clearly separate the `Exception` interface from what were internal engine errors before
- To avoid breaking existing code catching the `Exception` interface, letting the developer choose whether they want to also start catching errors or not

Alternatives to exceptions

As we just saw, we cannot use exceptions if we want to keep our code pure. What are our options for making sure we can signify an error to the caller of our function? We want our solution to have the following features:

- Enforce error management so that no errors can bubble up to the end user
- Avoid boilerplate or complex code structure
- Advertised in the signature of our function
- Avoid any risk of mistaking the error for a correct result

Before presenting a solution possessing all those benefits in the next section of the chapter, let's have a look at various ways error management is done in imperative languages.

In order to test the various ways, we will try to implement the `max` function we already used a bit earlier:

```
<?php
function max2(array $data): int
{
    return array_reduce($data, function(int $max, int $i) : int
{
        return $i > $max ? $i : $max;
    }, 0);
}
```

Because we chose the initial value 0, if we call the function with an empty array, we will get the result 0. Is 0 really the maximal value of an empty array? What happens if we call the version bundled with PHP, the `max([])` method?

```
Warning: max(): Array must contain at least one element
```

Also, the value false is returned. Our version uses the value 0 as a default value, and we could consider false to be an error code. The PHP version also greets you with a warning.

Now that we have a function we can improve, let us try the various options we have at our disposal. We will go from the worst to the best one.

Logging/displaying error message

As we just saw, PHP can display a warning message. We could also go with a message of level notice or error. This is probably the worst you could do because there is no way for the caller of your function to know something went wrong. Messages will only be displayed in the logs or on the screen once your application is run.

Also, in some cases, an error is something you can recuperate from. Since you have no idea something happened, you cannot do that in this case.

To make matters worse, PHP allows you to configure which error level gets displayed. In most cases, notices are just hidden, so no one will ever see that an error happened somewhere in the application.

To be fair to PHP, there is a way to catch those warnings and notices at runtime using a custom error handler declared with the `set_error_handler` parameter. However, in order to manage errors correctly, you will have to find a way to determine inside the handler which is the function that generated the error and act accordingly.

If you have multiple functions using these kinds of messages to signal errors, you will soon have either a really big error handler, or a multitude of smaller ones, making the whole process error prone and really cumbersome.

Error codes

Error codes are a heritage from the C language, which does not have any concept of exception. The idea is that a function always returns a code to signify the status of the computation and some other way is found to pass the return value around. Usually, the code 0 means that all

went well, and anything else is an error.

When it comes to numerical error codes, PHP has no function using them as return value as far as I can tell. The language has, however, a lot of functions returning the `false` value when an error occurred instead of the expected value. Only having one potential value to denote failure can lead to difficulties in transmitting information about what happened. For example, the documentation of the `move_uploaded_file` states that:

> *Returns TRUE on success.*
>
> *If filename is not a valid upload file, then no action will occur, and move_uploaded_file() will return False.*
>
> *If filename is a valid upload file, but cannot be moved for some reason, no action will occur, and move_uploaded_file() will return False. Additionally, a warning will be issued.*

This means you will be informed when you have an error, but you are unable to know which category of error it is without resorting to reading the error message. And even then, you will lack important information, such as why the uploaded file is invalid, for example.

If we wanted to better mimic the `max` function of PHP, we could do it like this:

```
<?php
function max3(array $data)
{
    if(empty($data)) {
        trigger_error('max3(): Array must contain at least one
element', E_USER_WARNING);
        return false;
    }

    return array_reduce($data, function(int $max, int $i) : int
{
        return $i > $max ? $i : $max;
    }, 0);
}
```

Since now our function needs to return the false value in case of error, we've had to remove the type hint for the return value, thus making our signature a bit less self-documenting.

Other functions, usually those that wrap an external library, also return the `false` value, in case of error but have companion functions in the form `X_errno` and `X_error` that return more information about the error of the last function that was executed. A few examples would be the `curl_exec`, `curl_errno`, and `curl_error` functions.

Such helpers allow for more fine-grained error handling but come with the cognitive cost that you must think about them. Error management is not enforced. To further my point, let us note that even the example for the `curl_exec` function in the official documentation does not set the best practice of checking the return value:

```
<?php

/* create a new cURL resource */
$ch = curl_init();

/* set URL and other appropriate options */
curl_setopt($ch, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch, CURLOPT_HEADER, 0);

/* grab URL and pass it to the browser */
curl_exec($ch);

/* close cURL resource, and free up system resources */
curl_close($ch);
```

Using the `false` value as a marker for failure also has another consequence in a language performing loose type casting like PHP. As stated in the aforementioned documentation, if you don't perform strict equality comparison, you risk considering a valid return value that evaluates as false as an error:

> *Warning: This function may return Boolean FALSE, but may also return a non-Boolean value which evaluates to False. Please read the section on Boolean values for more information. Use the ===*

operator for testing the return value of this function.

PHP uses the false error code only in the case of errors but does not return `true` or `0` as is usually the case in C. You don't have to find a way to transmit the return value to the user.

But if you want to implement your own function using a numerical error code to have the possibility of categorizing the error, you have to find a way to return both the code and the value. Usually, you can use one of two options:

- Using a parameter passed by reference which will hold the result; the `preg_match` parameter does that, for example, even if it is for different reasons. This is not strictly against function purity as long as the parameter is clearly identified as a return value.
- Returning an array or some other data structure that can hold two or more values. This idea is the beginning of what we will present as our functional solution in the next section.

Default value/null

A tad better than error codes when it comes to cognitive burden is the default value. If your function has only a reduced set of input that could result in an error, or if the error reason is not important, you could imagine returning a default value instead of specifying the error reason via an error code.

This will, however, open a new can of worms. It is not always easy to determine what a good default value is and, in some cases, your default value will also be a valid value, making it impossible to determine whether there was an error or not. For example, if you get 0 as a result when calling our `max2` function, you cannot know whether the array is empty or contains only the value 0 and negative numbers.

The default value could also depend on the context, in which case you will have to add a parameter to your function so that you can also specify the default value when calling it. Besides making the function signature bigger, this also defeats some performance optimization we

will learn later on and, although being totally pure and referentially transparent, increases the cognitive burden.

Let's add a default value parameter to our `max` function:

```
<?php

function max4(array $data, int $default = 0): int
{
    return empty($data) ? $default :
      array_reduce($data, function(int $max, int $i) : int
      {
          return $i > $max ? $i : $max;
      }, 0);
}
```

As we enforce the type of the default, we are able to restore the type hint for the return value. If you would like to pass anything as a default value, you will also have to remove the type hint.

To avoid some of the discussed issues, the value null is sometimes used as a default return value. Although not really a value, null is not categorized in the *error code* category as it is a perfectly valid value in some cases. Say you are searching for an item in a collection, what do you return if you find nothing?

Using the null value as a possible return value has, however, two issues:

- You cannot use a return type hint as null will not be considered of the correct type. Also, if you plan to use the value as a parameter, it also cannot be type hinted or it must be optional with the value null as a default value. This forces you to either remove your type hints or make your parameters optional.
- If your function usually returns objects, you will have to check the value for null, otherwise you risk what Tony Hoare called *The Billion Dollar Mistake*, a null pointer reference. Or, as it is reported in PHP, *Call to a member function XXX() on null*.

For the anecdote, Tony Hoare is the one that introduced null value to the world back in 1965, because it was so easy to implement. Later on, he

strongly regretted this decision and decided it was his billion dollar mistake. If you want to learn more about the reasons, I invite you to watch this talk he gave at https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.

Error handler

The last method is used a lot in the JavaScript world, where callbacks are everywhere. The idea is to pass an error callback each time the function is called. It can be even more powerful if you allow the caller to pass multiple callbacks, one for each kind of error that can arise.

Although it alleviates some of the issues that the default value has, such as the possibility to mix up a valid value with the default one, you still need to pass different callbacks depending on the context, making this solution only marginally better.

How will this approach look for our function? Consider the following implementation:

```
<?php

function max5(array $data, callable $onError): int
{
    return empty($data) ? $onError() :
      array_reduce($data, function(int $max, int $i) : int {
          return $i > $max ? $i : $max;
      }, 0);
}

max5([], function(): int {
    // You are free to do anything you want here.
    // Not really useful in such a simple case but
    // when creating complex objects it can prove invaluable.
    return 42;
});
```

Again, we kept the return type hint because the contract we have with our caller is to return an integer value. As stated in the comment, in this particular case, the default value as parameter will probably suffice, but in more complex situations, this approach provides more power.

We could also imagine passing the initial parameters to the callback along with information about the failure so that the error handler can act accordingly. In a way, this approach is a bit like the combination of everything we've seen earlier, as it allows you to:

- Specify a default return value of your choice
- Display or log any kind of error message you want
- Return a more complex data structure with an error code if you so wish

The Option/Maybe and Either types

As hinted before, our solution is to use a return type that contains the wanted value or something else in case of error. Those kinds of data structures are called **union types**. A union can contain values of different types, but only one at a time.

Let's start with the easiest of both union types we will see in this chapter. As always, naming is a difficult thing in computer science and people came up with different names to designate mostly the same structure:

- Haskell calls it the Maybe type, as does **Idris**
- Scala calls it the Option type, as does **OCaml**, **Rust**, and **ML**
- Since version 8, Java has an Optional type, as does Swift and the next specification of C++

Personally, I prefer the denomination Maybe as I consider an option to be something else. The remainder of the book will thus use this, except when a specific library has a type called **Option**.

The Maybe type is special in the sense that it can either hold a value of a particular type or the equivalent of *nothing*, or if you prefer, the null value. In Haskell, those two possible values are called `Just` and `Nothing`. In Scala, it is `Some` and `None` because `Nothing` is already used to designate the type equivalent of the value null.

Libraries implementing only a Maybe or Option type exist for PHP, and some of the libraries presented later in this chapter also ship with such types. But for the sake of correctly understanding how they work and their power, we will implement our own.

Let us reiterate our goals first:

- Enforce error management so that no errors can bubble up to the

end user
- Avoid boilerplate or complex code structure
- Advertised in the signature of our function
- Avoid any risk of mistaking the error for a correct result

If you type hint your function return value using the type that we will create in a few moments, you are taking care of our third goal. The presence of two distinct possibilities, the `Just` and `Nothing` values, ensure that you cannot mistake a valid result for an error. To make sure we don't end up with an erroneous value somewhere along the line, we must ensure that we cannot get a value from our new type without specifying a default if it is the `Nothing` value. And, concerning our second goal, we will see if we can write something nice:

```
<?php

abstract class Maybe
{
    public static function just($value): Just
    {
        return new Just($value);
    }

    public static function nothing(): Nothing
    {
        return Nothing::get();
    }

    abstract public function isJust(): bool;

    abstract public function isNothing(): bool;

    abstract public function getOrElse($default);
}
```

Our class has two static helper methods to create two instances of our soon-to-come child classes representing our two possible states. The Nothing value will be implemented as a singleton for performance reasons; since it will never hold any values, it is safe to do it this way.

The most important part of our class is an abstract `getOrElse` function,

which will force anyone wanting to get a value to also pass a default that will get returned if we have none. This way, we can enforce that a valid value will be returned even in the case of error. Obviously, you could pass the value null as the default, since PHP has no mechanism to enforce something else, but this would be akin to shooting yourself in the foot:

```
<?php
final class Just extends Maybe
{
    private $value;

    public function __construct($value)
    {
        $this->value = $value;
    }

    public function isJust(): bool
    {
        return true;
    }

    public function isNothing(): bool
    {
        return false;
    }

    public function getOrElse($default)
    {
        return $this->value;
    }
}
```

Our `Just` class is pretty simple; a constructor and a getter:

```
<?php
final class Nothing extends Maybe
{
    private static $instance = null;
    public static function get()
    {
        if(is_null(self::$instance)) {
            self::$instance = new static();
        }
```

```
        return self::$instance;
    }

    public function isJust(): bool
    {
        return false;
    }

    public function isNothing(): bool
    {
        return true;
    }

    public function getOrElse($default)
    {
        return $default;
    }
}
```

If you don't take the part about being a singleton into account, the `Nothing` class is even simpler because the `getOrElse` function will always return the default value no matter what. For those wondering, it is a deliberate choice to keep the constructor public. It has absolutely no consequences if someone wants to create a `Nothing` instance directly, so why bother?

Let's test our new `Maybe` type:

```
<?php

$hello = Maybe::just("Hello World !");
$nothing = Maybe::nothing();

echo $hello->getOrElse("Nothing to see...");
// Hello World !
var_dump($hello->isJust());
// bool(true)
var_dump($hello->isNothing());
// bool(false)

echo $nothing->getOrElse("Nothing to see...");
// Nothing to see...
var_dump($nothing->isJust());
```

```
// bool(false)
var_dump($nothing->isNothing());
// bool(true)
```

Everything seems to be working great. The need for boilerplate can be improved though. At this point, every time you want to instantiate a new `Maybe` type, you need to check the value you have and choose between the `Some` and `Nothing` values.

Also, it might happen that you need to apply some functions to the value before passing it further without knowing at this point what default value is best. As it would be cumbersome to get the value with some temporary default before creating a new `Maybe` type right behind, let's try to fix this aspect as well:

```
<?php

abstract class Maybe
{
    // [...]

    public static function fromValue($value, $nullValue = null)
    {
        return $value === $nullValue ?
            self::nothing() :
            self::just($value);
    }

    abstract public function map(callable $f): Maybe;
}

final class Just extends Maybe
{
    // [...]

    public function map(callable $f): Maybe
    {
        return new self($f($this->value));
    }
}

final class Nothing extends Maybe
{
    // [...]
```

```
    public function map(callable $f): Maybe
    {
        return $this;
    }
}
```

In order to have a somewhat coherent naming for utility methods, we use the same name as for functions working with collections. In a way, you can consider a `Maybe` type like a list with either one or no value. Let's add some other utility methods based on the same assumption:

```
<?php abstract class Maybe
{
    // [...]
    abstract public function orElse(Maybe $m): Maybe;
    abstract public function flatMap(callable $f): Maybe;
    abstract public function filter(callable $f): Maybe;
}

final class Just extends Maybe
{
    // [...]

    public function orElse(Maybe $m): Maybe
    {
        return $this;
    }

    public function flatMap(callable $f): Maybe
    {
        return $f($this->value);
    }

    public function filter(callable $f): Maybe
    {
        return $f($this->value) ? $this : Maybe::nothing();
    }
}

final class Nothing extends Maybe
{
    // [...]

    public function orElse(Maybe $m): Maybe
```

```
    {
        return $m;
    }

    public function flatMap(callable $f): Maybe
    {
        return $this;
    }

    public function filter(callable $f): Maybe
    {
        return $this;
    }
}
```

We have added three new methods to our implementation:

- The `orElse` method returns the current value if there is one, or the given value if it was `Nothing`. This allows us to easily get data from multiple possible sources.
- The `flatMap` method applies a callable to our value but does not wrap it inside a Maybe class. It is the responsibility of the callable to return a Maybe class itself.
- The `filter` method applies the given predicate to the value. If the predicate returns true value, we keep the value; otherwise, we return the value `Nothing`.

Now that we have implemented a working `Maybe` type, let's see how we can use it to get rid of error and null management easily. Imagine we want to display information about the connected user in the upper-right corner of our application. Without a `Maybe` type, you do something like the following:

```
<?php
$user = getCurrentUser();

$name = $user == null ? 'Guest' : $user->name;

echo sprintf("Welcome %s", $name);
// Welcome John
```

Here, we only use the name, so we can limit ourselves to one null check.

If we need more information from the user, the usual method is to use a pattern that is sometimes called the **Null object** pattern. In our case, our Null object will be an instance of `AnonymousUser` method:

```
<?php

$user = getCurrentUser();

if($user == null) {
   $user = new AnonymousUser();
}

echo sprintf("Welcome %s", $user->name);
// Welcome John
```

Now let's try to do the same with our `Maybe` type:

```
<?php

$user = Maybe::fromValue(getCurrentUser());

$name = $user->map(function(User $u) {
  return $u->name;
})->getOrElse('Guest');

echo sprintf("Welcome %s", $name);
// Welcome John

echo sprintf("Welcome %s", $user->getOrElse(new
AnonymousUser())->name);
// Welcome John
```

The first version might not be much better, as we have had to create a new function to extract the name. But let's keep in mind that you could do any number of treatments on your object before needing to extract a final value. Also, most functional libraries we present later provide helper methods to get value from objects in a simpler way.

You can also easily call a chain of methods until one of them returns a value. Say you want to display a dashboard, but those can be redefined on a per-group and per-level basis. Let's compare how our two methods fare.

First, the null value check approach:

```
<?php

$dashboard = getUserDashboard();
if($dashboard == null) {
    $dashboard = getGroupDashboard();
}
if($dashboard == null) {
    $dashboard = getDashboard();
}
```

And now, using `Maybe` type:

```
<?php

/* We assume the dashboards method now return Maybe instances
*/
$dashboard = getUserDashboard()
             ->orElse(getGroupDashboard())
             ->orElse(getDashboard());
```

I think the more readable one is easier to determine!

Finally, let us demonstrate a little example on how we could chain multiple calls on a `Maybe` instance without having to check whether we currently have a value or not. The chosen example is probably a bit silly, but it shows what is possible:

```
<?php

$num = Maybe::fromValue(42);

$val = $num->map(function($n) { return $n * 2; })
         ->filter(function($n) { return $n < 80; })
         ->map(function($n) { return $n + 10; })
         ->orElse(Maybe::fromValue(99))
         ->map(function($n) { return $n / 3; })
         ->getOrElse(0);
echo $val;
// 33
```

The power of our `Maybe` type is that we have never had to consider whether the instance contained a value. We were just able to apply

functions to it until finally, extracting the final value with the `getOrElse` method.

Lifting functions

We have seen the power of our new `Maybe` type. But the fact is, you either don't have time to rewrite all your existing functions to support it or you simply cannot because those are in an external third party.

Fortunately, you can **lift** a function to create a new function that takes a `Maybe` type as a parameter, applies to the original function to its value, and returns the modified `Maybe` type.

For this, we will need a new helper function. In order to keep things more or less simple, we will also assume that, if any of the parameters of the lifted function evaluate to the value `Nothing`, we will simply return nothing:

```
<?php

function lift(callable $f)
{
    return function() use ($f)
    {
        if(array_reduce(func_get_args(), function(bool $status,
Maybe $m) {
            return $m->isNothing() ? false : $status;
        }, true)) {
            $args = array_map(function(Maybe $m) {
                // it is safe to do so because the fold above
checked
                // that all arguments are of type Some
                return $m->getOrElse(null);
            }, func_get_args());
            return Maybe::just(call_user_func_array($f,
$args));
        }
        return Maybe::nothing();
    };
}
```

Let's try it:

```
<?php
function add(int $a, int $b)
{
    return $a + $b;
}

$add2 = lift('add');

echo $add2(Maybe::just(1), Maybe::just(5))-
>getOrElse('nothing');
// 6

echo $add2(Maybe::just(1), Maybe::nothing())-
>getOrElse('nothing');
// nothing
```

You can now lift any function so that it can accept our new `Maybe` type. The only consideration to have is that it will not work if you want to rely on any optional parameter of your function.

We could use reflection or other means to determine whether the function has an optional value or pass some default to the lifted functions, but this will only complicate things and make our function slower. If you need to use a function with optional parameters and `Maybe` types, you can rewrite it or make a custom wrapper for it.

As a closing note, the process of lifting is not reserved to Maybe types. You can lift any function to accept any kind of container type. A better name for our helper will probably be **liftMaybe** or we could add it as a static method on our `Maybe` class to make things clearer.

The Either type

The `Either` type is a generalization of our `Maybe` type. Instead of having a value and nothing, you have a left and right value. As it is also a union type, only one of these two possible values can be set at any given time.

The `Maybe` type works well when there is only a few sources of errors or when the error in itself does not matter. With the `Either` type, we can provide any kind of information we want in case of error through the left

value. The right value is used for success because of the obvious wordplay.

Here is a simple implementation of `Either` type. Since the code in itself is pretty boring, only the base class is presented in the book. You can access both child classes on the Packt website:

```
<?php
abstract class Either
{
    protected $value;

    public function __construct($value)
    {
        $this->value = $value;
    }

    public static function right($value): Right
    {
        return new Right($value);
    }

    public static function left($value): Left
    {
        return new Left($value);
    }

    abstract public function isRight(): bool;
    abstract public function isLeft(): bool;
    abstract public function getRight();
    abstract public function getLeft();
    abstract public function getOrElse($default);
    abstract public function orElse(Either $e): Either;
    abstract public function map(callable $f): Either;
    abstract public function flatMap(callable $f): Either;
    abstract public function filter(callable $f, $error):
Either;
}
```

The implementation proposes the same API as the one we have for `Maybe` class, assuming that the right value is the valid one. You should be able to use the `Either` class instead of the `Maybe` class, everywhere without having to change your logic. The only difference is the methods

to check which case we are in, and change the method to the new `getRight` or `getLeft` method.

It is also possible to write lift for our new type:

```
<?php
function liftEither(callable $f, $error = "An error occured")
{
    return function() use ($f)
    {
        if(array_reduce(func_get_args(), function(bool $status,
Either $e) {
            return $e->isLeft() ? false : $status;
        }, true)) {
            $args = array_map(function(Either $e) {
                // it is safe to do so because the fold above
checked
                // that all arguments are of type Some
                return $e->getRight(null);
            }, func_get_args());
            return Either::right(call_user_func_array($f,
$args));
        }
        return Either::left($error);
    };
}
```

This function is, however, a bit less useful than a custom wrapper because you cannot specify an error message that is specific to the possible errors.

Libraries

Now that we covered the basics of the functional techniques with the various functions already available in PHP, it's time to have a look at the various libraries that will allow us to concentrate on our business code, instead of writing helpers and utility functions, as we did with our new `Maybe` and `Either` types.

The functional-php library

The `functional-php` library is probably one of the oldest libraries related to functional programming for PHP, as its first release dates back to June 2011. It evolved nicely with the newest PHP version and even switched to Composer last year for distribution.

The code is available on GitHub at https://github.com/lstrojny/functional-php. It should be fairly easy to install if you are accustomed to using Composer by writing the following command:

```
composer require lstrojny/functional-php.
```

The library used to be implemented both in PHP and as part of a C extension for performance reasons. But recent improvements of the PHP core regarding speed and the burden of maintaining two codebases made the extension obsolete.

A lot of helper functions are implemented-we won't have enough space to go into the details of each of them right now. If you are interested, you can have a look at the documentation. We will, however, quickly present the important ones and the rest of the book will contain examples using more of them.

Also, we haven't yet discussed some concepts covered by the library-related functions will be presented as we tackle those topics.

How to use the functions

As already discussed in Chapter 1, *Functions as First-Class Citizens in PHP*, since PHP 5.6, you can import a function from a namespace. This is the easiest way to use the library. You can also import the whole namespace and prefix all functions when calling them:

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

use function Functional\map;

map(range(0, 4), function($v) { return $v * 2; });

use Functional as F;

F\map(range(0, 4), function($v) { return $v * 2; });
```

It is also important to note that most functions accept arrays and anything implementing the `Traversable` interface, such as iterators.

General helpers

Those functions can help you in a variety of contexts, not only functional ones:

- The `true` and `false` functions check whether all elements in a collection are either strictly True or strictly False.
- The `truthy` and `falsy` functions are same as before but the comparison is not strict.
- The `const_function` function returns a new function that will always return the given value. This could be used to simulate immutable data.

Extending PHP functions

PHP functions have a tendency to work only on *real* arrays. The following functions extend their behavior to anything that can be iterated over using a `foreach` loop. The order of parameters is also kept the same across all functions:

- The contains method checks whether the value is contained in the given collection. The third parameter controls whether the

comparison should be strict or not.
- The `sort` method sorts a collection but returns a new array instead of sorting by reference. You can decide to preserve the keys or not.
- The `map` method extends the `array_map` method behavior to all collections.
- The sum, maximum, and minimum methods perform the same job as their PHP counterparts but on any type of collection. Besides those, the library also contains product, ratio, difference, and average.
- The zip method performs the same work as the `array_map` method when you don't pass it a function. You can, however, also pass a callback to determine how the various items should be merged.
- The `reduce_left` and `reduce_right` methods fold collections either from the left or the right.

Working with predicates

When working with collections, you often want to check whether some, all, or no elements verify a certain condition and act accordingly. In order to do so, you can use the following functions:

- The `every` function returns the true value if all elements of a collection are valid for the predicate
- The `some` function returns the value true if at least one element is valid for the predicate
- The `none` function returns the value true if no elements at all are valid for the predicate

Those functions won't modify the collections. They are only to check whether the elements match a certain condition or not. If you need to filter some elements, you can use the following helpers:

- The `select` or `filter` functions return only the elements that are valid for the predicate.
- The `reject` function returns only the elements that are invalid for the predicate.
- The first or `head` function return the first element that is valid for the predicate.
- The last function returns the last element that is valid for the

predicate.
- The `drop_first` function removes elements from the beginning of the collection until the given callback is `true`. As soon as the callback returns false, stop removing elements.
- The `drop_last` function is the same as the previous function, but starts at the end.

All those functions return a new array, leaving the original collection untouched.

Invoking functions

It is cumbersome to declare an anonymous function as soon as you want to invoke a function in a callback. Those helpers will do exactly that for you with a simpler syntax:

- The `invoke` helper invokes a method on all objects in a collection and returns a new collection with the result
- The `invoke_first` and `invoke_last` helpers invoke a method on the first and last object of a collection, respectively
- The `invoke_if` helper invokes the given method on the first parameter if it is a valid object. You can pass the method parameters and a default value.
- The `invoker` helper returns a new callable that invokes the given method with the given parameters to its parameter.

You might also want to call a function until you obtain a value or some threshold is reached. The library's got you covered:

- The `retry` library calls a function until it stops returning an exception or the number of tries is reached
- The `poll` library calls the function until it returns the truthy value or a given timeout is reached

Manipulating data

The previous functions group was about invoking functions with helpers; this one is about getting and manipulating data without having to resort to an anonymous function each time:

- The `pluck` function fetches a property from all objects in a given collection and returns a new collection with the values.
- The `pick` function selects an element from an array based on the given key. You can provide a default value if the element does not exist.
- The `first_index_of` and `last_index_of` functions return the first, respectively last, index of an element matching the given value.
- The `indexes_of` function returns all indexes matching the given value.
- The `flatten` function reduces the depth of nested collection to a single flat collection.

Sometimes, you also want to separate a collection into multiple parts, either given a predicate or some grouping value:

- The `partition` method accepts a list of predicates-each item of the collection is put in a given group based on the first predicate for which it is valid
- The `group` method creates multiple groups based on each different value returned by the callback for each element

Wrapping up

As you can see, the `functional-php` library offers a lot of different helpers and utility functions. It might not be obvious how you can get the most of all of them right now, but I hope the remainder of the book will give you a glimpse of what you can achieve.

Also, do not forget that we didn't present all functions as some of them need a bit of theoretical explanation first. All in due time.

The php-option library

We created our own version of the `Maybe` type earlier. This library proposes a more complete implementation. The naming used by Scala was chosen, however. The source code is on GitHub at https://github.com/schmittjoh/php-option. The easiest way to install is to by writing the following command using Composer:

```
composer require phpoption/phpoption
```

An interesting addition is the `LazyOption` method, which takes a callback instead of a value. The callback will be executed only when a value is needed. This is particularly interesting when you use the `orElse` method to give alternatives in case the previous one is an invalid value. By using the `LazyOption` method in this case, you avoid doing unnecessary computation as soon as one value is valid.

You also have various helpers to help you call methods only if the value is valid, for example, and there are multiple instantiation possibilities offered. The library also provides an API even more akin to the one you are accustomed to for a collection.

Laravel collections

As already mentioned in the first chapter, Laravel offers a great library to manage collections. It declares a class called `Collection` which is used internally by their ORM, **Eloquent**, and most of the other parts relying on collections.

Internally, a simple array is used, but it is wrapped in a way that promotes immutability of the data and a functional approach manipulating data. To achieve this goal, between 60 and 70 methods are proposed to the developer.

If you are already using Laravel, you are probably already familiar with the possibilities offered by this support class. If you are using any other framework, you can still benefit from it by getting the extracted part from https://github.com/tightenco/collect.

The documentation is available on Laravel's official website at https://laravel.com/docs/collections. We won't describe each method in detail as there are a lot of them. If you are using Laravel and want to learn more about all the possibilities offered by its collections, you can head over to https://adamwathan.me/refactoring-to-collections/.

Working with Laravel's Collections

The first step is to transform your array or `Traversable` interface to an instance of the `Collection` class using the collect utility function. You will then have access to all the various methods the class provides. Let's make a quick list of those that we have already encountered in another form so far:

- The `map` method applies a function to all elements and returns the new value
- The `filter` method filters the collection using a predicate
- The `reduce` method folds the collection using the given callback
- The `pluck` gets a given property from all elements
- The `groupBy` method partitions the collection using the given value from each element

All those methods return a new instance of the `Collection` class, preserving the values of your original instance.

Once you are done manipulating, you can get the current values as an array using all method.

The immutable-php library

This library proposing an immutable data structure is born due to various gripes with the `SplFixedArray` method from the **Standard PHP** Library, mostly with its hard-to-use API. At its core, the `immutable-php` library uses the aforementioned data structure, but with a nice set of methods to wrap it.

The `SplFixedArray` method is a specific implementation of an array with a fixed size and which allows only numerical indexes. Those constraints allow for a really fast array structure.

You can have a look on the GitHub project page at https://github.com/jkoudys/immutable.php or install it by writing the following command using Composer:

```
composer require qaribou/immutable.php.
```

Using immutable.php

Creating a new instance is really easy using the dedicated static helpers `fromArray` or `fromItems` for any instance of the `Traversable` class. Your newly created `ImmArray` instance can be accessed like any array, iterated over using `foreach` loop, and counted using the `count` method. However, if you try to set a value, you will be rewarded with an exception.

Once you have your immutable array, you can have various methods to apply the transformations you should now be accustomed to:

- The `map` method to apply a function to all items and return the new value
- The `filter` method to create a new array with only items valid for the predicate
- The `reduce` method to fold items using a callback

You also have other helpers:

- The `join` method concatenates a collection of strings
- The `sort` method returns a collection sorted using the given callback

Your data can also easily be retrieved as a traditional array or encoded to a JSON format.

All in all, this library provides fewer methods than Laravel's Collection but you will have better performance and a much lower memory footprint.

Other libraries

Since PHP core is lacking a lot of utility functions and features to do proper functional programming, a lot of people started working on libraries that implement the missing pieces. This is the reason why you will find a lot of those if you start looking.

Here is an incomplete and unordered list of such libraries if those that were presented previously do not suit your needs.

The Underscore.php library

A variety of ports based on the API of the `Underscore.js` library exists for PHP. I am personally not a big fan of the `Underscore.js` libraries because the function parameters are often in the wrong order to perform efficient function composition. This point is well explained in this video at https://www.youtube.com/watch?v=m3svKOdZijA.

However, if you are accustomed to using it, here is a short list of various ports:

- https://github.com/brianhaveri/Underscore.php: The oldest port as far as I can tell. It has not seen any activity since 2012, but a lot of forks exist to improve compatibility with newer PHP versions and apply bug fixes.
- https://github.com/wikiHow/Underscore.php: One of the most maintained forks of the preceding library.
- https://github.com/Anahkiasen/underscore-php: Originally a port of its JS counterpart. It now contains some different features trying to respect the original philosophy.
- https://github.com/Im0rtality/Underscore: The more recent try to have something akin to the `Underscore.js` library. At the time of writing, the documentation is lacking some important topics and the library differs from the JavaScript version in quite a few places.

Saber

Saber is strictly following the latest PHP version as a requirement. It makes use of strong typing, immutable objects, and lazy evaluation. In order to use its various methods, you have to *box* your values inside classes supplied by the library. It can be cumbersome but it provides safety and reduces bugs.

It seems to be inspired by C# and, principally, F#, the latter being the functional language running on the .NET virtual machine, or **CLR** to call it by its real name. You can find the source code and documentation on GitHub at https://github.com/bluesnowman/fphp-saber.

Rawr

Rawr is not only a functional library. It tries to fix shortcomings of the PHP language in a more general manner. Like Saber, it provides a new class to box your scalar values; however, the types are used in a way that is closer to what Haskell does. You can also wrap your anonymous functions inside a class to have better typing safety around them.

The library also adds a more **Smalltalk** flavored object-orientation, Monads, and allows you to perform some kind of prototype-based programming as you can do with JavaScript.

Sadly, the library seems to be at a standstill and the documentation is not up to date with the source code. You can, however, find some inspiration there. You can find the code on GitHub at https://github.com/haskellcamargo/rawr.

PHP Functional

This library revolves mostly around the concept of Monads we will see in Chapter 5, *Real-life Monads*. The acknowledged inspiration is Haskell, from which the library implements:

- State Monad
- IO Monad
- Collection Monad
- Either Monad
- Maybe Monad

Through the `Collection` Monad, the library offers the various methods we expect the `map`, `reduce`, and `filter` methods.

As it is inspired by Haskell, you might find it a bit more difficult to use in the beginning. However, it should prove more powerful in the end. You can find the code on GitHub at https://github.com/widmogrod/php-functional.

Functional

Originally created as a learning playground, this library has grown into something that could prove useful if you are looking for something

relatively small. The main idea is to provide a framework so that you can remove all loops in your code.

The most interesting feature is that all functions can be partially applied without doing anything special. Partial application is really important for function composition. We will discover both topics in Chapter 4, *Compositing functions*.

The library also has all the traditional contenders such as mapping and reducing. The code and documentation are available on GitHub at https://github.com/sergiors/functional.

PHP functional programming Utils

This library tries to walk the same path as the `functional-php` library, which we presented in the previous pages. As far as I can tell, it has, however, slightly fewer features as of now. It can be an interesting library for people wanting something a tad smaller and maybe easier to learn. The code is on GitHub at https://github.com/daveross/functional-programming-utils.

Non-standard PHP library

This library is not strictly a functional one. The idea is more to extend the standard library with various helpers and utility functions to make working with the collections easier.

It contains useful features such as helpers to validate function parameters with ease, either with already defined constraints or custom ones. It also extends existing PHP functions so that they can work on anything that is `Traversable` interface and not just arrays.

The library was created in 2014 but was nearly dead until work started going strong again at the end of 2015. Now it is a possible replacement for any of the libraries we presented previously. If you are interested, grab the code on GitHub at https://github.com/ihor/Nspl.

Summary

In this long chapter, we presented all the practical building blocks that we will use throughout the book. I hope the few examples didn't seem too dry. There was a lot to cover and only a limited set of pages. The following chapters will build on what we have learned with better examples.

You first read some general advice about programming in general that is especially important for a functional codebase. We then discovered basic functional techniques such as mapping, folding, filtering, and zipping, all of which are available directly within PHP.

The next part was a brief introduction to recursion, both a technique to solve a particular set of problems and to avoid using loops. In a book about a functional language, the topic might have deserved a whole chapter, but since PHP has various loop structures, it's a bit less important. Also, we will see more recursion examples in the following chapters.

We also discussed exceptions and why they pose issues in a functional codebase, and we wrote implementations for the Maybe and Either types as a better way to manage errors after discussing the pros and cons of other methods.

Finally, we presented some libraries that provide functional constructs and helpers so that we don't have to write our own.

Chapter 4. Composing Functions

In previous chapters, we talked a lot about building blocks and small pure functions. But so far, we haven't even hinted at how those can be used to build something bigger. What good is a building block if you cannot use it? The answer partly lies in function composition.

Although this chapter completes the previous one, this technique is such an integral and important part of any functional program that it warranted its own chapter.

In this chapter, we will cover the following topics:

- Function composition
- Partial application
- Currying
- Parameter order importance
- Real-life application of those concepts

Composing functions

As is often the case in functional programming, the concept of function composition is borrowed from mathematics. If you have two functions, `f` and `g`, you can create a third function by composing them. The usual notation in mathematics is $(f \circ g)(x)$, which is equivalent to calling them one after the other as $f(g(x))$.

You can compose any two given functions very easily with PHP, using a wrapper function. Say you want to display a title in all caps and only safe HTML characters:

```
<?php

function safe_title(string $s)
{
    $safe = htmlspecialchars($s);
    return strtoupper($safe);
}
```

You can also avoid the temporary variable altogether:

```
<?php

function safe_title2(string $s)
{
    return strtoupper(htmlspecialchars($s));
}
```

This works well when you want to compose only a few functions. But creating a lot of those wrapper functions can become really cumbersome. What if you could simply use `$safe_title = strtoupper htmlspecialchars` line of code? Sadly, this operator does not exist in PHP, but the `functional-php` library we presented earlier contains a `compose` function which does exactly that:

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

use function Functional\compose;

$safe_title2 = compose('htmlspecialchars', 'strtoupper');
```

The gain may not seem that important, but let's compare using such an approach in a bit more context:

```
<?php

$titles = ['Firefly', 'Buffy the Vampire Slayer', 'Stargate
Atlantis', 'Tom & Jerry', 'Dawson's Creek'];

$titles2 = array_map(function(string $s) {
    return strtoupper(htmlspecialchars($s));
}, $titles);

$titles3 = array_map(compose('htmlspecialchars', 'strtoupper'),
$titles);
```

Personally, I find the second approach a lot easier to read and understand. And it gets better, as you can pass more than two functions to the `compose` function:

```
<?php
```

```
$titles4 = array_map(compose('htmlspecialchars', 'strtoupper',
'trim'), $titles);
```

One thing that can be misleading is the order of application of the functions. The mathematical notation `f ∘ g` first applies `g` and then the result is passed to `f`. However, the `compose` function from `functional-php` library applies the functions in the order they are passed in the `compose('first', 'second', 'third')` parameters.

This might be easier to understand depending on your personal preferences, but beware when you use another library, as the order of application might be reversed. Always make sure you've read the documentation carefully.

Partial application

You might want to set some parameters of a function but leave some of them unassigned for later. For example, we might want to create a function that returns an excerpt of a blog post.

The dedicated term for setting such a value is **bind a parameter** or **bind an argument**. The process itself is called **partial application** and the new function is set to be partially applied.

The naive way to do this is by wrapping the function in a new one:

```
<?php
function excerpt(string $s)
{
    return substr($s, 0, 5);
}

echo excerpt('Lorem ipsum dolor si amet.');
// Lorem
```

As with composition, always creating new functions can quickly become cumbersome. But once again, the `functional-php` library has us covered. You can decide to bind parameters either from the left, the right, or in any particular location in the function signature, using respectively the `partial_left`, `partial_right`, or `partial_any` function.

Why three functions? Mostly for performance reasons, as the left and right versions will perform a lot faster because the parameters will be replaced once and for all, whereas the last one will use placeholders evaluated upon each call to the new function.

In the last example, the placeholder is defined using the function `...` which is the ellipsis unicode character. If you don't have an easy way to type it on your computer, you can also use the `placeholder` function from the `Functional` namespace which is an alias.

Currying

Currying is often used as a synonym for partial application. Although both concepts allow us to bind some parameters of a function, the core ideas are a bit different.

The idea behind currying is to transform a function, taking multiple arguments into a sequence of functions taking one argument. As this might be a bit hard to grasp, let's try to curry `substr` function. The result is called a **curryied function**:

```
<?php

function substr_curryied(string $s)
{
    return function(int $start) use($s) {
        return function(int $length) use($s, $start) {
            return substr($s, $start, $length);
        };
    };
}

$f = substr_curryied('Lorem ipsum dolor sit amet.');
$g = $f(0);
echo $g(5);
// Lorem
```

As you can see, each call returns a new function that takes the next parameter. This illustrates the principal difference with partial application. When you call a partially applied function, you will obtain a result. But, when you call a curryied function, you will get a new function until you pass the last parameter. Also, you can only bind the arguments in order starting from the left.

If the call chain seems overly lengthy, you can greatly simplify it starting from PHP 7. This is because the RFC *Uniform variable syntax* was implemented (see https://wiki.php.net/rfc/uniform_variable_syntax for details):

```
<?php
```

```
echo substr_curryied('Lorem ipsum dolor sit amet.')(0)(5);
// Lorem
```

The advantages of currying might not seem evident when presented like this. But, as soon as you start working with higher-order functions such as `map` or `reduce` function, the idea becomes really powerful.

You might remember the `pluck` function from the `functional-php` library. The idea is to retrieve a given property from a collection of objects. If the `pluck` function was implemented as a curryied function, it could be used in a variety of ways:

```
<?php

function pluck(string $property)
{
    return function($o) use($property) {
        if (is_object($o) && isset($o->{$propertyName})) {
            return $o->{$property};
        } elseif ((is_array($o) || $o instanceof ArrayAccess)
&&  isset($o[$property])) {
            return $o[$property];
        }

        return false;
    };
}
```

We could get a value from any kind of object or array easily:

```
<?php

$user = ['name' => 'Gilles', 'country' => 'Switzerland',
'member'  => true];
pluck('name')($user);
```

We could extract a property from a collection of objects, as does the version from the `functional-php` library:

```
<?php

$users = [
```

```
    ['name' => 'Gilles', 'country' => 'Switzerland', 'member'
=>  true],
    ['name' => 'Léon', 'country' => 'Canada', 'member' =>
false],
    ['name' => 'Olive', 'country' => 'England', 'member' =>
true],
];
pluck('country')($users);
```

As our implementation returns `false` when nothing is found, we could use it to filter arrays that contain a certain value:

```
<?php

array_filter($users, pluck('member'));
```

We could combine multiple use cases to get the names of all the members:

```
<?php

pluck('name', array_filter($users, pluck('member')));
```

Without currying, we would have needed to either write a wrapper around a more traditional `pluck` function, or create three specialized functions.

Let's go a step further and combine multiple curryied functions. First, we will need to create a wrapper function around the `array_map` and `preg_replace` functions:

```
<?php

function map(callable $callback)
{
    return function(array $array) use($callback) {
        return array_map($callback, $array);
    };
}

function replace($regex)
{
    return function(string $replacement) use($regex) {
```

```
        return function(string $subject) use($regex,
$replacement)
{
            return preg_replace($regex, $replacement,
$subject);
        };
    };
}
```

Now we can use those to create multiple new functions, for example, a function that replaces all spaces in a string with underscores or all vowels with a star:

```
<?php function map(callable $callback)
{
    return function(array $array) use($callback) {
        return array_map($callback, $array);
    };
}

function replace($regex)
{
    return function(string $replacement) use($regex) {
        return function(string $subject) use($regex,
$replacement)
{
            return preg_replace($regex, $replacement,
$subject);
        };
    };
}
```

Currying functions in PHP

I hope you are now convinced of the power of currying. If not, I hope the examples to follow will do it for you. In the meantime, you are probably thinking it is really cumbersome to write a new utility function around existing ones to create a new curryied version, and you would be right.

In languages such as Haskell, all functions are curryied by default. Sadly, this is not the case in PHP, but the process is easy and repetitive enough that we can write a helper function.

Due to the possibility of having optional parameters in PHP, we will first create a function `curry_n` that takes the number of parameters you want to curry. This way, you will also be able to decide if you want to curry all parameters, or only some of them. It can also be used for functions that have a variable number of parameters:

```
<?php

function curry_n(int $count, callable $function): callable
{
    $accumulator = function(array $arguments) use($count,
$function, &$accumulator) {
        return function() use($count, $function, $arguments,
$accumulator) {
            $arguments = array_merge($arguments,
func_get_args());

            if($count <= count($arguments)) {
                return call_user_func_array($function,
$arguments);
            }

            return $accumulator($arguments);
        };
    };
    return $accumulator([]);
}
```

The idea is to use an inner helper function, taking the already passed values as parameters, and then creating a closure with those. When called, the closure will decide, based on the actual number of values, whether we can call the original function, or whether we need to create a new function using our helper.

Be aware that if you give a parameter count higher than the real count, all extraneous parameters will be passed along to the original function but will probably be ignored. Also, giving a smaller count will result in the last step expecting more than just one parameter to correctly complete.

We can now create our second function that will determine the number

of parameters using `reflection` variable:

```
<?php

function curry(callable $function, bool $required = true):
callable
{
    if(is_string($function) && strpos($function, '::', 1) !==
false) {
        $reflection = new \ReflectionMethod($f);
    }
    else if(is_array($function) && count($function) == 2)
    {
        $reflection = new \ReflectionMethod($function[0],
$function[1]);
    }
    else if(is_object($function) && method_exists($function,
'__invoke'))
    {
        $reflection = new \ReflectionMethod($function,
'__invoke');
    }
    else
    {
        $reflection = new \ReflectionFunction($function);
    }

    $count = $required ?
        $reflection->getNumberOfRequiredParameters() :
        $reflection->getNumberOfParameters();

    return curry_n($count, $function);
}
```

As you can see, there is no easy way to determine the number of parameters a function is expecting. We also had to add a parameter to determine whether we should consider all parameters, including those with a default value, or only the mandatory ones.

You might have noticed that we don't create functions that take strictly one parameter; instead, we used the `func_get_args` function to get all passed parameters. This allows the use of currying functions more naturally and is also on a par with what is done in functional languages.

Our definition of currying is now more along the lines of *A function that returns a new function until it receives all its arguments*.

The examples in the remainder of the book will assume that this curry function is available and ready to use.

At the time of writing, there is an open pull request on the `functional-php` library to incorporate this function.

Parameter order matters a lot!

As you might remember from the first chapter, `array_map` and `array_filter` functions have their parameters in different orders. For sure, it makes them a bit more difficult to use, as you are more prone to getting it wrong, but it is not the only issue this poses. To illustrate why parameter order matters, let's create curryied versions of both of them:

```
<?php

$map = curry_n(2, 'array_map');
$filter = curry_n(2, 'array_filter');
```

We are using `curry_n` functions for two different reasons here:

- The `array_map` function accepts a variable number of arrays, so we enforce the value to 2 to be on the safe side
- The `array_filter` function has a third parameter named `$flag` for which the optional value is fine

Remember the order parameters of our newly curryied functions? The `$map` parameter will take the callback first, and the `$filters` parameter expects the collection first. Let's try to create a new useful function knowing this:

```
<?php

$trim = $map('trim');
$hash = $map('sha1');

$oddNumbers = $filter([1, 3, 5, 7]);
$vowels = $filter(['a', 'e', 'i', 'o', 'u']);
```

Our mapping examples are really basic but serve some purpose, whereas our filtering examples are just static data. I bet you can find some way to use `$trim` and `$hash` parameters, but what are the chances you will need a list of odd numbers or vowels ready to be filtered?

Another example can be taken from a bit earlier in this chapter-

remember our curryied example of `substr` function?

```
<?php

function substr_curryied(string $s)
{
    return function(int $start) use($s) {
        return function(int $length) use($s, $start) {
            return substr($s, $start, $length);
        };
    };
}

$f = substr_curryied('Lorem ipsum dolor sit amet.');
$g = $f(0);
echo $g(5);
// Lorem
```

I can guarantee you it would be a lot more useful if we could first define the start and length to create. For example, a `$take5fromStart` function; instead of having this awkward `$substrOnLoremIpsum` parameters, we simply called the `$f` parameter in the example.

The important thing here is that the data you want to work upon, your "subject", must come last because it greatly increases reuse of your curryied functions and lets you use them as parameters to other higher-order functions.

As in the last example, let's say we want to create a function that takes the first two letters of all elements of a collection. We will try to do it with a set of two functions, where the arguments are in different orders.

The implementation of the function is left as an exercise, as it does not really matter to drive the point home.

In example one, the subject is the first argument:

```
<?php

$map = curry(function(array $array, callable $cb) {});
$take = curry(function(string $string, int $count) {});
```

```
$firstTwo = function(array $array) {
    return $map($array, function(string $s) {
        return $take($s, 2);
    });
}
```

The parameter order forces us to create wrapper functions. In fact, it doesn't even matter that the functions are curryied because we cannot use this fact.

In example two, the subject is at the end:

```
<?php

$map = curry(function(callable $cb, array $array) {});
$take = curry(function(int $count, string $string) {});

$firstTwo = $map($take(2));
```

As a matter of fact, a well-chosen order also helps a lot with function composition, as we will see in the following section.

As a final note on the subject and to be completely fair, the version using functions with backward parameters could have been written using the `partial_right` function from the `functional-php` library, and you could use the `partial_any` function for functions with more parameters in strange orders. But even so, the solution is not as simple as the one with the arguments in the right order:

```
<?php

use function Functional\partial_right;

$firstTwo = partial_right($map, partial_right($take, 2));
```

Using composition to solve real issues

As an example, let's say that your boss comes in and wants you to produce a new report with the phone numbers of all users that have registered in the last 30 days. We assume that we have the following class representing our users. Obviously, a real class will store and return real data, but let us just define our API:

```
<?php

class User {
    public function phone(): string
    {
        return '';
    }

    public function registration_date(): DateTime
    {
        return new DateTime();
    }
}

$users = [new User(), new User(), new User()]; // etc.
```

Without any knowledge of functional programming, you might write something like this:

```
<?php

class User {
    public function phone(): string
    {
        return '';
    }
    public function registration_date(): DateTime
    {
        return new DateTime();
    }
}
```

```
$users = [new User(), new User(), new User()]; // etc.
```

A first look at our function tells us that it is not pure, as the limit is computed inside the function, thus subsequent calls could result in a different user list. We could also leverage the `map` and `filter` functions:

```
<?php

function getUserPhonesFromDate($limit, $users)
{
    return array_map(function(User $u) {
        return $u->phone();
    }, array_filter($users, function(User $u) use($limit) {
        return $u->registration_date()->getTimestamp() >
$limit;
    }));
}
```

Depending on your preferences, the code might now be a bit easier to read, or not at all, but at least we have a pure function and our concerns are a bit more separated. We can, however, do better. Firstly, the `functional-php` library has a function that allows us to create a helper calling a method on an object:

```
<?php

use function Functional\map;
use function Functional\filter;
use function Functional\partial_method;

function getUserPhonesFromDate2($limit, $users)
{
    return map(
        filter(function(User $u) use($limit) {
            return $u->registration_date()->getTimestamp()
>$limit;
        }, $users),
        partial_method('phone')
    );
}
```

It is a bit better, but if we accept having to create some new helper functions, we can improve the solution even more. Also, those helper

functions are new building blocks we will be able to reuse:

```
<?php

function greater($limit) {
    return function($a) {
        return $a > $limit;
    };
}

function getUserPhonesFromDate3($limit, $users)
{
    return map(
        filter(compose(
            partial_method('registration_date'),
            partial_method('getTimestamp'),
            greater($limit)
          ),
          $users),
        partial_method('phone')
    );
}
```

If we have curryied versions of `filter` and `map` functions, we can even create a function that only takes a date and returns a new function that can be further composed and reused:

```
<?php

use function Functional\partial_right;

$filter = curry('filter');
$map = function($cb) {
    return function($data) use($cb) {
        return map($data, $cb);
    };
};

function getPhonesFromDate($limit)
{
    return function($data) use($limit) {
        $function = compose(
            $filter(compose(
            partial_method('getTimestamp'),
                partial_method('registration_date'),
```

```
                greater($limit)
            )),
            $map(partial_method('phone'))
        );
        return $function($data);
    };
}
```

As a good reminder about the necessity of having a good parameter order, since the `map` function from the `functional-php` library has the same signature as the original one from PHP, we had to curry it manually.

Our resulting function is a tad longer than the original imperative one, but, in my opinion, it is easier to read. You can easily follow what is happening:

1. Filter the data using:
    1. The registration date
    2. From this, you get the timestamp.
    3. Check whether it's greater than the given limit.
2. Map the `phone` method on the result.

I totally agree if you find that the name `partial_method` is not ideal and that the presence of the calls to the `compose` function somehow makes it a bit difficult on the eyes. As a matter of fact, in a hypothetical language with a `compose` operator, auto-currying, and some syntactic sugar to defer a call to a method, this could look like this:

```
getFromDate($limit) = filter(
  (->registration_date) >>
  (->getTimestamp) >>
  (> $limit)
) >> map(->phone)
```

Now that we have our function, your boss walks right back in your office with new requirements. In fact, he only wants the three most recent registrations from the last 30 days. Easy, let's just compose our new function with some more building blocks:

```
<?php

use function Functional\sort;
use function Functional\compare_on;

function take(int $count) {
    return function($array) use($count) {
        return array_slice($array, 0, $count);
    };
};

function compare($a, $b) {
    return $a == $b ? 0 : $a < $b ? -1 : 1;
}

function getAtMostThreeFromDate($limit)
{
    return function($data) use($limit) {
        $function = compose(
            partial_right(
                'sort',
                compare_on('compare',
partial_method('registration_date'))
            ),
            take(3),
            getPhonesFromDate($limit)
        );
        return $function($data);
    };
}
```

In order to take a certain number of items from the beginning of an array, we need to create a `take` function around `array_slice` function. We also need a function to compare values, which we can do simply because `DateTime` function overloads the comparison operators.

Again, the `functional-php` library gets argument order wrong for `sort` function, so we need to partially apply instead of curry. And the `compare_on` function creates a comparator given a comparison function and a "reducer" which is called on each item being compared. In our case, we want to compare the registration date, so we reuse our different method application.

We need to perform the sorting operation before filtering because our `getPhonesFromDate` method returns only the phone numbers as the name suggests. Our resulting function is itself a curryied composition of other functions, thus allowing easy reuse.

I hope this small example has finished convincing you of the power of using small functions as building blocks and composing them to solve issues. If that is not the case, maybe one of the more advanced techniques we'll see in the following chapters will do it for you.

As a final note, as you perhaps gathered from the examples, PHP is sadly missing a lot of utility functions to make the life of a functional programmer easy. Also, even the `functional-php` library, which is widely used, gets some parameter orders wrong and is missing some important pieces of code, such as currying.

By combining multiple libraries, we could have a better coverage of the required features, but it would also add a lot of useless code and some mismatched function names that wouldn't really make your life easier.

What I can recommend is to keep a file with all the little gems you create along the way and soon you will have your own compilation of helpers that really suits your need and coding style. This advice might go against best practices concerning reusable packages with a large community around them, but until someone creates the right library, it helps a lot. And who knows, you might be the one who has enough energy to create the missing pearl in the functional PHP ecosystem.

Summary

This chapter revolved around function composition, which is a really powerful idea once you get used to it. By using small building blocks, you can create complex processes while keeping the readability and maintainability offered by short functions.

We also talked about the partial application and the most powerful concept of currying, which allow us to easily create more specialized versions of existing functions and rewrite our code to be more readable.

We tackled argument order, a topic which is often brushed off but is really important as soon as you want to use higher-order functions. The combination of currying and a correct parameter succession allowed us to reduce the need for boilerplate code and wrapper functions, a process sometimes referred to as eta-reduction.

Finally, with all the aforementioned tools, we tried to demonstrate solutions to some issues and problems that you could encounter in your day-to-day programming to help you write better code.

Chapter 5. Functors, Applicatives, and Monads

The previous chapter introduced the first real functional techniques, such as function composition and currying. In this chapter, we will delve into more theoretical concepts again by presenting the concept of monads. There won't be a lot of practical application as we have a lot of ground to cover. However, Chapter 6, *Real-life Monads* will use everything we learn here to solve real issues.

You might already have heard the term **monad**. Usually, it is associated with a sense of dread by non functional programmers. Monads are usually described as hard to understand despite countless tutorials on the subject. The fact is, they are hard to understand and people writing those tutorials often forget how much time it took them to understand the idea correctly. This is a common pedagogic pitfall, probably much better described in this article at https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/.

You will probably not get everything the first time. Monads are a highly abstract concept and, even if the topic seems clear to you at the end of the chapter, you will probably stumble upon something later on that will baffle your intuition of what a monad really is.

I will try to do my best to explain things clearly, but if you find my explanations lacking, I added references to other material about the topic in the *Further reading* section at the end of this chapter. In this chapter, we will cover the following topics:

- Functors and related laws
- Applicative functors and related laws
- Monoids and related laws
- Monads and related laws

There will be a lot of theoretical content only with implementation of the concepts. Don't expect a lot of examples until Chapter 6, *Real-life Monads*.

Functors

Before speaking directly of monads, let us start at the beginning. In order to understand what a monad is, we need to introduce a few related concepts. The first one is functors.

To complicate matters a bit, the term **functor** is used in imperative programming to describe a function object, which is something totally different. In PHP, an object with the `__invoke` method, as we saw in Chapter 1, *Functions as First Class Citizens*, is such a function object.

However, in functional programming, a functor is a concept taken and adapted from the mathematical field of category theory. Details aren't that important for our purpose; it suffice it to say that a functor is a pattern allowing us to map a function to one or more values contained in a context. Also, in order for the definition to be as complete as possible, our functor must respect some laws, which we will describe and verify later.

We have already used map multiple times on collections, which makes them de facto functors. But if you remember correctly, we also named our method to apply a function to the value contained in Maybe in the same way. The reason is that functors can be seen as containers with a means to apply a function to the contained value.

In a sense, any class that implements the following interface could be called a `Functor`:

```
<?php

interface Functor
{
    public function map(callable $f): Functor;
}
```

However, describing it like that is a bit reductive. A simple PHP array is also a functor (because the `array_map` function exists), as is anything implementing the `Traversable` interface as soon as you use the `functional-php` library and its map function.

Why such a fuss for such a simple idea? Because, although the idea is simple in itself, it allows us to reason about what is happening in a different light and possibly helps understand and refactor the code.

Also, the `map` function can do much more than blindly apply the given `callable` type, as is the case with an array. If you remember our `Maybe` type implementation, in the case of a value `Nothing`, the `map` function simply kept returning the `Nothing` value in order to allow for a simpler management of null values.

We could also imagine having much more complicated data structures in our functors, such as trees, where the function given to the `map` function is applied to all nodes.

Functors allow us to share a common interface, our `map` method or function, to perform a similar operation on a variety of data types while hiding the complexity of the implementation. As often with functional programming, the cognitive burden is reduced because you don't have multiple names for an identical operation. Function and method names such as "apply", "perform", and "walk" are often seen to describe the same thing for example.

Identity function

Our final concern is the two functor laws that come with the concept. But before introducing them, we need to take a small detour about the identity function, or often `id`. It is a really simple function that simply returns its parameter:

```
<?php

function id($value)
{
```

```
    return $value;
}
```

Why would anyone need a function doing as little as this? First of all, we will need it later to prove the laws of the various abstractions presented in this chapter. But real-world applications also exist.

For example, when you are doing a fold with numbers, say summing them, you will use the initial value `0`. The `id` function will have the same role when folding over functions. As a matter of fact, the compose function is implemented using the `id` function in the `functional-php` library.

Another use might be some function from another library that performs an operation you are interested in, but also calls a callback on the resulting data. If the callback is mandatory, but you don't want to do anything else to your data, just pass `id` and you will get them unaltered.

Let's use our new function to declare a property of our `compose` function for any function `f`, taking only one argument:

```
compose(id, f) == compose(f, id)
```

What this basically says is that, if you first apply the argument `id` then `f`, you will get the exact same result as when first applying `f` and then `id`. At this point, this should be evident to you. If not, I encourage you to revisit the last chapter until you clearly understand why this is the case.

Functor laws

Now that we have our identity function covered, let's get back to our laws. They are important for two reasons:

- They give us a set of constraints to guarantee the validity of our functors
- They allow us to perform refactoring that is proven correct

Without further ado, here they are:

1. *map(id) == id*

2. *compose(map(f), map(g)) == map(compose(f, g))*

The first law states that mapping the `id` function on the contained value is exactly the same as calling the `id` function directly on the functor itself. When this law holds, this guarantees us that our map function only applies the given function to the data without performing any other kind of treatment.

The second law states that first mapping the `f` function then the g one on our value is the same as first composing `f` and `g` together and then mapping the resulting function. Knowing this, we can perform all kinds of optimization. For example, instead of looping three times over our data for three different methods, we can compose them together and perform only one loop.

I can imagine not everything being crystal clear for you right now, so instead of wasting time trying to explain them further, let's verify if they hold for the `array_map` method. This will probably help you to get the gist of it; the following code expects the `id` function defined previously to be in scope :

```
<?php

$data = [1, 2, 3, 4];

var_dump(array_map('id', $data) === id($data));
// bool(true)

function add2($a)
{
    return $a + 2;
}

function times10($a)
{
    return $a * 10;
}

function composed($a) {
    return add2(times10($a));
}
```

```
var_dump(
array_map('add2', array_map('times10', $data)) ===
array_map('composed', $data)
);
// bool(true)
```

The composition is performed manually; as in my opinion resorting to currying here would only have made things more complicated.

As we can see, both laws hold for the `array_map` method, which is a good sign because it means there is no hidden data processing done in the shadow and we can avoid looping two or more times on our array when only once is enough.

Let's try the same with our `Maybe` type defined earlier:

```
<?php

$just = Maybe::just(10);
$nothing = Maybe::nothing();

var_dump($just->map('id') == id($just));
// bool(true)

var_dump($nothing->map('id') === id($nothing));
// bool(true)
```

We had to switch to a non-strict equality for the `$just` case because we would have a result of false otherwise as PHP compares object instances and not their values. A `Maybe` type wraps the resulting value in a new object and PHP only performs internal value comparison in the case of a non-strict equality; the `add2`, `times10`, and `composed` functions defined above are expected to be in scope:

```
<?php

var_dump($just->map('times10')->map('add2') == $just-
>map('composed'));
// bool(true)

var_dump($nothing->map('times10')->map('add2') === $nothing-
>map('composed'));
```

```
// bool(true)
```

Great, our `Maybe` type implementation is a valid functor.

Identity functor

As we discussed in the section about the identity function, an identity functor also exists. It acts as a really simple functor that does nothing to the value besides holding it:

```
<?php

class IdentityFunctor implements Functor
{
    private $value;

    public function __construct($value)
    {
        $this->value = $value;
    }

    public function map(callable $f): Functor
    {
        return new static($f($this->value));
    }

    public function get()
    {
        return $this->value;
    }
}
```

As with the identity function, the use for such a functor is not immediately evident. However, the idea is the same-you can use it when you have some function having a functor as a parameter, but you don't want to modify your actual value.

This should begin to make more sense in the following chapters of this book. In the meantime, we will use the identity functor to explain some more advanced concepts.

Closing words

Let me reiterate again that functors are a really simple abstraction to understand, but also a really powerful one. We've seen only two of them, but an infinity of data structures that can be transformed to functors really easily.

Any function or class that allows you to map a given function to one or more values held in a context can be considered a functor. Identity functors or arrays are simple example of such a context; other examples would be the `Maybe` and `Either` types we discussed earlier or any class having a `map` method that allows you to apply a function to the contained valued.

I cannot encourage you enough to try implementing this map pattern and verify whether both laws hold wherever you create a new class or data structure. This will allow you to have an easier understanding of what your code can perform and you will be able to perform optimization using composition with the guarantee that your refactoring is correct.

Applicative functors

Let's take an instance of our identity functor, holding some integer and a curryied version of an `add` function:

```
<?php

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

$id = new IdentityFunctor(5);
```

Now, what happens when we try to map the `$add` parameter on our functor? Consider the following code:

```
<?php

$hum = $id->map($add);

echo get_class($hum->get());
// Closure
```

As you may have guessed, our functor now contains a closure representing a partially applied `add` parameter with the value `5` as the first parameter. You can use the `get` method to retrieve the function and use it, but it is not really that useful.

Another possibility would be to map another function, taking our function as a parameter, and doing something with it:

```
<?php

$result = $hum->map(function(callable $f) {
    return $f(10);
});
echo $result->get();
// 15
```

But I imagine we can all agree that this is not a really effective way to perform such an operation. What would be great is to be able to simply pass the value `10` or maybe another functor to `$hum` and achieve the same result.

Enters the applicative functor. As the name suggests, the idea is to apply functors. More precisely, to apply functors to other functors. In our case, we could apply `$hum`, which is a functor containing a function, to another functor containing the value `10` and obtain the value `15` we are after.

Let's create an extended version of our `IdentityFunctor` class to test our idea:

```
<?php

class IdentityFunctorExtended extends IdentityFunctor
{
    public function apply(IdentityFunctorExtended $f)
    {
        return $f->map($this->get());
    }
}

$applicative = (new IdentityFunctorExtended(5))->map($add);
$ten = new IdentityFunctorExtended(10);
echo $applicative->apply($ten)->get();
// 15
```

You can even create your `Applicative` class containing only your function and apply the values afterwards:

```
<?php

$five = new IdentityFunctorExtended(5);
$ten = new IdentityFunctorExtended(10);
$applicative = new IdentityFunctorExtended($add);

echo $applicative->apply($five)->apply($ten)->get();
// 15
```

The applicative abstraction

We are now able to use our `IdentifyFunctor` class as a curryied function holder. What if we could abstract this idea away and create something on top of the `Functor` class?

```
<?php
```

```
abstract class Applicative implements Functor
{
    public abstract static function pure($value): Applicative;
    public abstract function apply(Applicative $f):
Applicative;
    public function map(callable $f): Functor
    {
        return $this->pure($f)->apply($this);
    }
}
```

As you can see, we created a new abstract class instead of an interface. The reason is because we can implement the `map` function using the `pure` and `apply` methods so it makes no sense to force everyone wanting to create an `Applicative` class to implement it.

The `pure` function is called so because anything stored inside an `Applicative` class is considered pure as there is no way to modify it directly. The term is taken from the Haskell implementation. Other implementations sometimes use the name *unit* instead. pure is used to create a new applicative from any `callable`.

The `apply` function applies the stored function to the given parameter. The parameter must be of the same type so that the implementation knows how to access the inner value. Sadly, the PHP type system does not allow us to enforce this rule and we must resign ourselves to defaulting to `Applicative`.

We have the same issue for the definition of map that has to keep the return type as `Functor`. We need to do this as the PHP type engine does not support a feature called **return type covariance**. If it did, we could specify a more specialized type (that is a child type) as a return value.

The `map` function is implemented using the aforementioned functions. First we encapsulate our `callable` using the `pure` method and we apply this new applicative on the actual value. Nothing really fancy.

Let's test our implementation:

```
<?php

$five = IdentityApplicative::pure(5);
$ten = IdentityApplicative::pure(10);
$applicative = IdentityApplicative::pure($add);

echo $applicative->apply($five)->apply($ten)->get();
// 15

$hello = IdentityApplicative::pure('Hello world!');

echo IdentityApplicative::pure('strtoupper')->apply($hello)-
>get();
// HELLO WORLD!

echo $hello->map('strtoupper')->get();
// HELLO WORLD!
```

Everything seems to work fine. We were even able to verify that our map implementation seems to be correct.

As with the functor, we can create the simplest of the `Applicative` class abstraction:

```
<?php

class IdentityApplicative extends Applicative
{
    private $value;

    protected function __construct($value)
    {
        $this->value = $value;
    }

    public static function pure($value): Applicative
    {
        return new static($value);
    }

    public function apply(Applicative $f): Applicative
    {
        return static::pure($this->get()($f->get()));
    }
```

```
    public function get()
    {
        return $this->value;
    }
}
```

Applicative laws

The first important property of applicative is that they are *closed under composition*, meaning an applicative will return a new applicative of the same type. Also, the apply method takes an applicative of its own type. We cannot enforce this using the PHP type system, so you will need to be careful otherwise something might break at some point.

The following rules also need to be respected to have a proper applicative functor. We will first present them all in detail and then verify that they hold for our `IdentityApplicative` class later.

Map

pure(f)->apply == map(f)

Applying a function using an applicatives is the same as mapping this function over. This law simply tells us that we can use an applicative anywhere we used a functor before. We lose no power by switching to an applicative.

In fact, this is not really a law as it can be inferred from the following four laws. But as it is not at all evident, and in order to make things clearer, let's state it.

Identity

pure(id)->apply($x) == id($x)

Applying the identity function results in no change to the value. As with the identity law for functors, this ensure that the `apply` method does nothing but apply the function. No hidden transformation is happening behind our backs.

Homomorphism

pure(f)->apply($x) == pure(f($x))

Creating an applicative functor and applying it to a value has the same effect as first calling the function on the value and boxing it inside the functor afterwards.

This is an important law because our first motivation to dive into applicative is usage of the curryied function instead of unary functions. This law ensures that we can create our applicative at any stage instead of needing to box our function right away.

Interchange

pure(f)->apply($x) == pure(function($f) { $f($x); })->apply(f)

This one is a bit tricky. It states that applying a function on a value is the same as creating an applicative functor with a lifted value and applying it to the function. In this case, a lifted value is a closure around the value that will call the given function on it. The law guarantees that the pure function performs no modification besides boxing the given value.

Composition

pure(compose)->apply(f1)->apply(f2)->apply($x) == pure(f1)->apply(pure(f2)->apply($x))

A simpler version of this law could be written with *pure(compose(f1, f2))->apply($x)* on the left-hand side. It simply states, as the composition law for functors, that you can apply a composed version of two functions to your value or call them separately. This ensures that you can perform the same optimizations for functors.

Verifying that the laws hold

As we saw for functors, it is more than recommended to test whether your implementations hold for all laws. This can be a really tedious process, especially if you have four of them. So instead of performing

the check manually, let's write a helper:

```
<?php

function check_applicative_laws(Applicative $f1, callable $f2,
$x)
{
    $identity = function($x) { return $x; };
    $compose = function(callable $a) {
        return function(callable $b) use($a) {
            return function($x) use($a, $b) {
                return $a($b($x));
            };
        };
    };

    $pure_x = $f1->pure($x);
    $pure_f2 = $f1->pure($f2);

    return [
        'identity' =>
            $f1->pure($identity)->apply($pure_x) ==
            $pure_x,
        'homomorphism' =>
            $f1->pure($f2)->apply($pure_x) ==
            $f1->pure($f2($x)),
        'interchange' =>
            $f1->apply($pure_x) ==
            $f1->pure(function($f) use($x) { return $f($x); })-
>apply($f1),
        'composition' =>
            $f1->pure($compose)->apply($f1)->apply($pure_f2)-
>apply($pure_x) ==
            $f1->apply($pure_f2->apply($pure_x)),
        'map' =>
            $pure_f2->apply($pure_x) ==
            $pure_x->map($f2)
    ];
}
```

The `identity` and `compose` functions are declared inside the helper so that it is completely self-contained and you can use it in a variety of situations. Also the `compose` function from the `functional-php` library is not adapted as it is not curryied and it takes a variable number of arguments.

Also, in order to avoid having a lot of arguments, we take an instance of the `Applicative` class so we can have a first function and the type to check and then a `callable` and a value that will be lifted to the applicative and used whenever necessary.

This choice limits the functions we can use as the value must match the type of the parameter for both functions; the first function must also return a parameter of the same type. If this is too constraining for you, you can decide to extend the helper to take two more arguments, a second applicative and a lifted value, and use those when necessary.

Let's verify our `IdentityApplicative` class:

```
<?php

print_r(check_applicative_laws(
IdentityApplicative::pure('strtoupper'),
    'trim',
    ' Hello World! '
));
// Array
// (
//     [identity] => 1
//     [homomorphism] => 1
//     [interchange] => 1
//     [composition] => 1
//     [map] => 1
// )
```

Great, everything seems to be fine. If you want to use this helper, you need to choose functions that are compatible as you might encounter some error messages that lack clarity as we cannot ensure that the type of the return value for the first function matches the type of the first parameter for the second function.

Since this kind of automatic checking can greatly help, let us quickly write the same kind of function for functors:

```
<?php

function check_functor_laws(Functor $func, callable $f,
```

```
callable $g)
{
    $id = function($a) { return $a; };
    $composed = function($a) use($f, $g) { return $g($f($a));
};

    return [
        'identity' => $func->map($id) == $id($func),
        'composition' => $func->map($f)->map($g) == $func-
>map($composed)
    ];
}
```

And check our never tested `IdentityFunctor` with it:

```
<?php

print_r(check_functor_laws(
    new IdentityFunctor(10),
    function($a) { return $a * 10; },
    function($a) { return $a + 2; }
));
// Array
// (
//     [identity] => 1
//     [composition] => 1
// )
```

Good, everything is fine.

Using applicatives

As we already saw, arrays are functors, because they have a `map` function. But a collection can also easily be an applicative. Let's implement a `CollectionApplicative` class:

```
<?php

class CollectionApplicative extends Applicative implements
IteratorAggregate
{
    private $values;

    protected function __construct($values)
```

```
    {
        $this->values = $values;
    }

    public static function pure($values): Applicative
    {
        if($values instanceof Traversable) {
            $values = iterator_to_array($values);
        } else if(! is_array($values)) {
            $values = [$values];
        }

        return new static($values);
    }

    public function apply(Applicative $data): Applicative
    {
        return $this->pure(array_reduce($this->values,
            function($acc, callable $function) use($data) {
                return array_merge($acc, array_map($function,
$data->values) );
            }, [])
        );
    }

    public function getIterator() {
        return new ArrayIterator($this->values);
    }
}
```

As you can see, this is all fairly easy. To simplify our life we just wrap anything that is not a collection inside an array and we transform instances of the `Traversable` interface to a real array. This code would obviously need some improvement to be used in production, but it will suffice for our little demonstration:

```
<?php

print_r(iterator_to_array(CollectionApplicative::pure([
  function($a) { return $a * 2; },
  function($a) { return $a + 10; }
])->apply(CollectionApplicative::pure([1, 2, 3]))));
// Array
// (
//     [0] => 2
```

```
//     [1] => 4
//     [2] => 6
//     [3] => 11
//     [4] => 12
//     [5] => 13
// )
```

What is happening here? We have a list of functions in our applicative and we apply that to a list of numbers. The result is a new list with each function applied to each number.

This small example is not really useful, but the idea can be applied to anything. Imagine you have some kind of image gallery application where the user can upload some images. You also have various processing you want to make on those images:

- Limiting the size of the final image as the user tends to upload them too big
- Creating a thumbnail for the index page
- Creating a small version for mobile devices

The only thing you need to do is create an array of all your functions, an array of the uploaded images, and apply the same pattern we just did to our numbers. You can then use the group function from the `functional-php` library to regroup your images together:

```
<?php

use function Functional\group;

function limit_size($image) { return $image; }
function thumbnail($image) { return $image.'_tn'; }
function mobile($image) { return $image.'_small'; }

$images = CollectionApplicative::pure(['one', 'two', 'three']);

$process = CollectionApplicative::pure([
  'limit_size', 'thumbnail', 'mobile'
]);

$transformed = group($process->apply($images), function($image,
$index) {
```

```
    return $index % 3;
});
```

We use the index in the transformed array to regroup the images together. Every third image is the limited one, every fourth the thumbnail, and finally we have the mobile version. The result can be seen as follows:

```
<?php

print_r($transformed);
// Array
// (
//     [0] => Array
//         (
//             [0] => one
//             [3] =>one_tn
//             [6] =>one_small
//         )
//
//     [1] => Array
//         (
//             [1] => two
//             [4] =>two_tn
//             [7] =>two_small
//         )
//
//     [2] => Array
//         (
//             [2] => three
//             [5] =>three_tn
//             [8] =>three_small
//         )
//
//)
```

You might be hungry for more at this stage, but you will need to be patient. Let's finish first with the theory in this chapter and we will soon see more potent examples in the next.

Monoids

Now that we have an understanding of applicative functors, we need to add one last piece to the puzzle before speaking about monads, the monoid. Once again, the concept is taken from the mathematical field of category theory.

A **monoid** is a combination of any type and a binary operation on this type with an associated identity element. For example, here are some combinations that you probably never expected were monoids:

- Integer number and the addition operation, the identity being 0 because*$i + 0 == $i*
- Integer number and the multiplication operation, the identity being 1 because*$i * 1 == $i*
- Arrays and the merge operation, the identity being the empty array because *array_merge($a, []) == $a*
- String and the concatenate operation, the identity being the empty string because *$s . '' == $s*

For the remainder of the chapter, let's call our operation *op* and the identity element *id*. The `op` call comes from operation or operator and is used in some `Monoid` implementation across multiple languages. Haskell uses the terms **mempty** and **mappend** to avoid clashes with other function names. Sometimes zero is used instead of *id* or identity.

A monoid must also respect a certain number of laws, two to be precise.

Identity law

$a op id == id op $a == $a

The first law ensures that the identity can be used on both sides of the operator. An identity element can work only when applied as the right-hand or left-hand side of the operator. This is, for example, the case for operations on matrices. In this case, we speak of left and right identity elements. In the case of a `Monoid`, we need a two-sided identity, or

simply identity.

As for most identity laws, verifying it for a `Monoid` implementation ensures that we correctly apply the operator with no other side-effects.

Associativity law

($a op $b) op $c == $a op ($b op $c)

This law guarantees that we can regroup our call to the operator in any order we want, as long as some other operations are not interleaved. This is important because it allows us to reason about possible optimization with the insurance that the result will be the same.

Knowing that a sequence of operations is associative; you can also separate the sequence into multiple parts, distribute the computation across multiple threads, cores, or computers, and when all the intermediary results come in, apply the operation between them to get the final result.

Verifying that the laws hold

Let's verify those laws for the monoids we spoke about earlier. First, the integer addition:

```
<?php

$a = 10; $b = 20; $c = 30;

var_dump($a + 0 === $a);
// bool(true)
var_dump(0 + $a === $a);
// bool(true)
var_dump(($a + $b) + $c === $a + ($b + $c));
// bool(true)
```

Then, the integer multiplication:

```
<?php

var_dump($a * 1 === $a);
```

```
// bool(true)
var_dump(1 * $a === $a);
// bool(true)
var_dump(($a * $b) * $c === $a * ($b * $c));
// bool(true)
```

Then the array merge as follows:

```
<?php

$v1 = [1, 2, 3]; $v2 = [5]; $v3 = [10];

var_dump(array_merge($v1, []) === $v1);
// bool(true)
var_dump(array_merge([], $v1) === $v1);
// bool(true)
var_dump(
array_merge(array_merge($v1, $v2), $v3) ===
array_merge($v1, array_merge($v2, $v3))
);
// bool(true)
```

And finally, the string concatenation:

```
<?php

$s1 = "Hello"; $s2 = " World"; $s3 = "!";

var_dump($s1 . '' === $s1);
// bool(true)
var_dump('' . $s1 === $s1);
// bool(true)
var_dump(($s1 . $s2) . $s3 == $s1 . ($s2 . $s3));
// bool(true)
```

Great, all our monoids respect both laws.

What about subtraction or division? Are they also a monoid? It's pretty obvious that 0 is the identity of subtraction and 1 is the identity for division, but what about associativity?

Consider the following for checking the associativity of subtraction or division:

```php
<?php

var_dump(($a - $b) - $c === $a - ($b - $c));
// bool(false)
var_dump(($a / $b) / $c === $a / ($b / $c));
// bool(false)
```

We clearly see that neither subtraction nor division is associative. When working with such abstraction, it is always important to test our assumption using the law. Otherwise a refactoring or calling some function expecting a `Monoid` could go really wrong. Obviously, the same applies for functors and applicatives.

What are monoids useful for?

Honestly, monoids themselves are not really useful, especially in PHP. Eventually, in a language where you can either declare new operators or redefine existing ones, you could ensure their associativity and the existence of an identity using monoids. But even so, there would be no real advantages.

Also, if the language could automatically distribute work done with operators that are part of a `Monoid`, it would be a great way to speed up lengthy computation. But I am not aware of any language, even academic ones, that are currently capable of doing so. Some languages perform operation reordering to improve efficiency, but that's about it. Obviously, PHP can't do any of that since the concept of monoid is not in the core.

Why bother then? Because monoids can be used with higher-order functions and some constructs that we will discover later can take full advantage of their laws. Also, since PHP does not allow us to use existing operators as functions, as is the case with Haskell, for example, we've had to define functions such as `add` before. Instead, we can define a `Monoid` class. It would have the same utility as our simple function with some nice properties added.

At the risk of sounding like a broken record, explicitly stating that an

operation is a monoid reduces cognitive burden. When using a monoid, you have the assurance that the operation is associative and that it respects a two-sided identity.

A monoid implementation

PHP does not support generics, so we have no way to encode the type information of our `Monoid` formally. You will have to choose a self-explanatory name or document that is the type clearly.

Also, as we want our implementation to replace functions such as `add`, we need some additional methods on our class to allow for this usage. Let's see what we can do:

```
<?php

abstract class Monoid
{
    public abstract static function id();
    public abstract static function op($a, $b);

    public static function concat(array $values)
    {
        $class = get_called_class();
        return array_reduce($values, [$class, 'op'], [$class,
'id']());
    }

    public function __invoke(...$args)
    {
        switch(count($args)) {
            case 0: throw new RuntimeException("Except at least
1 parameter");
            case 1:
                return function($b) use($args) {
                    return static::op($args[0], $b);
                };
            default:
                return static::concat($args);
        }
    }
}
```

Obviously, we have our `id` and `op` functions declared abstract as those will be the specific parts for each of our monoids.

One of the main advantages of having a `Monoid` is easily being able to fold a collection of values having the type of the `Monoid` class. This is why we create the `concat` method as a helper to do exactly that.

Finally, we have an `__invoke` function so that our `Monoid` can be used like a normal function. The function is curryied in a specific way. If you pass more than one parameter on the first call, the `concat` method will be used to return a result immediately. Otherwise, with exactly one parameter, you will get a new function waiting for a second parameter.

Since we are at it, let's write a function to check the laws:

```
<?php

function check_monoid_laws(Monoid $m, $a, $b, $c)
{
    return [
        'left identity' => $m->op($m->id(), $a) == $a,
        'right identity' => $m->op($a, $m->id()) == $a,
        'associativity' =>
            $m->op($m->op($a, $b), $c) ==
            $m->op($a, $m->op($b, $c))
    ];
}
```

Our first monoids

Let's create monoids for the cases we've seen previously and demonstrate how we can use them:

```
<?php

class IntSum extends Monoid
{
    public static function id() { return 0; }
    public static function op($a, $b) { return $a + $b; }
}

class IntProduct extends Monoid
```

```
{
    public static function id() { return 1; }
    public static function op($a, $b) { return $a * $b; }
}

class StringConcat extends Monoid
{
    public static function id() { return ''; }
    public static function op($a, $b) { return $a.$b; }
}

class ArrayMerge extends Monoid
{
    public static function id() { return []; }
    public static function op($a, $b) { return array_merge($a,
$b); }
}
```

Let's validate the laws on them:

```
<?php

print_r(check_monoid_laws(new IntSum(), 5, 10, 20));
// Array
// (
//     [left identity] => 1
//     [right identity] => 1
//     [associativity] => 1
// )

print_r(check_monoid_laws(new IntProduct(), 5, 10, 20));
// Array
// (
//     [left identity] => 1
//     [right identity] => 1
//     [associativity] => 1
// )

print_r(check_monoid_laws(new StringConcat(), "Hello ",
"World", "!"));
// Array
// (
//     [left identity] => 1
//     [right identity] => 1
//     [associativity] => 1
// )
```

```
print_r(check_monoid_laws(new ArrayMerge(), [1, 2, 3], [4, 5],
[10]));
// Array
// (
//     [left identity] => 1
//     [right identity] => 1
//     [associativity] => 1
// )
```

For the example, let's try to create a monoid for subtraction and check the laws:

```
<?php

class IntSubtraction extends Monoid
{
    public static function id() { return 0; }
    public static function op($a, $b) { return $a - $b; }
}

print_r(check_monoid_laws(new IntSubtraction(), 5, 10, 20));
// Array
// (
//     [left identity] =>
//     [right identity] => 1
//     [associativity] =>
// )
```

As expected, the associativity laws fail. We also have an issue with the left identity because of *0 - $a == -$a*. So let's not forget to test our monoids regarding the laws to ensure that they are correct.

Two interesting monoids can be created regarding the Boolean type:

```
<?php

class Any extends Monoid
{
    public static function id() { return false; }
    public static function op($a, $b) { return $a || $b; }
}

class All extends Monoid
{
```

```
    public static function id() { return true; }
    public static function op($a, $b) { return $a && $b; }
}

print_r(check_monoid_laws(new Any(), true, false, true));
// Array
// (
//     [left identity] => 1
//     [right identity] => 1
//     [associativity] => 1
// )

print_r(check_monoid_laws(new All(), true, false, true));
// Array
// (
//     [left identity] => 1
//     [right identity] => 1
//     [associativity] => 1
// )
```

These two monoids allow us to verify whether at least one or all conditions are met. These are the monoidic versions of the every and some functions in the `functional-php` library. These two monoids serve the same purpose as the sum and product ones since PHP does not allow us to use Boolean operators as functions:

```
<?php

echo Any::concat([true, false, true, false]) ? 'true' :
'false';
// true

echo All::concat([true, false, true, false]) ? 'true' :
'false';
// false
```

They can prove useful when you need to create a series of conditions programmatically. Instead of iterating over all of them to generate results, just feed them to the `Monoid`. You can also write a *none* monoid as an exercise to see if you understood the concept well.

Using monoids

One of the most obvious ways to use our new monoids is to fold a collection of values:

```
<?php

$numbers = [1, 23, 45, 187, 12];
echo IntSum::concat($numbers);
// 268

$words = ['Hello ', ', ', 'my ', 'name is John.'];
echo StringConcat::concat($words);
// Hello , my name is John.

$arrays = [[1, 2, 3], ['one', 'two', 'three'], [true, false]];
print_r(ArrayMerge::concat($arrays));
// [1, 2, 3, 'one', 'two', 'three', true, false]
```

This property is so interesting that most functional programming languages implement the idea of a `Foldable` type. Such a type needs to have an associated monoid. With the help of the property we have just seen, the type can then be easily folded. However, the idea is difficult to port to PHP as we will miss the syntactic sugar needed to improve over using the `concat` method like we just did.

You can also use them as the `callable` type and pass them to higher order functions:

```
<?php

use function Functional\compose;

$add = new IntSum();
$times = new IntProduct();

$composed = compose($add(5), $times(2));
echo $composed(2);
// 14
```

Obviously, this is not limited to the compose function. You could rewrite all of the previous examples from this book that used an `add` function and use our new `Monoid` instead.

As we progress in this book, we will see more ways to use monoids associated with functional techniques that we have yet to discover.

Monads

We started learning about functors, which are a collection of values that can be mapped over. We then introduced the idea of applicative functors, which allow us to put those values in a certain context and apply functions to them while preserving the context. We also made a quick detour to talk about monoids and their properties.

With all this prior knowledge, we are finally ready to start with the idea of monads. As James Iry humorously noted in *A Brief, Incomplete, and Mostly Wrong History of Programming Languages*:

> *A monad is a monoid in the category of endofunctors, what's the problem?*

This fictional quote, attributed to Philip Wadler, one of the original people involved in the Haskell specification and a proponent of the use of monads, can be found in context at http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html.

Without some knowledge of category theory, it would be hard to clearly explain what the quote is all about, especially since it is fictional and voluntarily vague enough to be funny. Suffice to say that monads are akin to monoids as they share roughly the same set of laws. Also, they are directly linked to the concept of functors and applicative.

A monad, like a functor, acts as some kind of container for a value. Also, like for applicative, you can apply functions to the encapsulated values. All three patterns are a way to put some data inside a context. However, there are a few differences between the two:

- The applicative encapsulates a function. The monad, and the functor, encapsulate a value.
- The applicative uses functions that return non-lifted values. The monad uses functions that return a monad of the same type.

As functions are also valid values, this doesn't mean both are

incompatible, it only means we will need to define a new API for our monad type. However, we can freely extend Applicative as it contains totally valid methods in a monadic context:

```
<?php

abstract class Monad extends Applicative
{
    public static function return($value): Monad
    {
        return static::pure($value);
    }

    public abstract function bind(callable $f): Monad;
}
```

Our implementation is pretty simple. We alias pure with return as Haskell uses this term in monads so that people don't get lost. Be aware that it has nothing to do with the return keyword you are accustomed to; it's really just to put the value inside the context of the monad. We also define a new bind function that takes a `callable` type as parameter.

As we don't know how the internal value will be stored and because of the limitations of the PHP type system, we cannot implement neither the `apply` nor the `bind` function although they should be pretty similar:

- The `apply` method takes a value wrapped in the `Applicative` class and applies the stored function to it
- The `bind` method takes a function and applies it to the stored value

The difference between the two is that the `bind` method needs to return the value directly, whereas the `apply` method first wraps the value again using the `pure` or `return` functions.

As you may have come to understand, people using different languages tend to name things a bit differently. This is why you will sometimes see the `bind` method called **chain** or **flatMap**, depending on the implementation that you are looking at.

Monad laws

You know the drill by now; there are also some laws a monad must respect to be called a monad. The laws are the same as for a monoid-identity and associativity. So all the same useful properties of a monoid are also true for a monad.

However, as you will see, the laws we will describe seem to have nothing in common with the identity and associativity laws we've seen for monoids. This is linked to the way we defined the `bind` and `return` functions. Using something called the **Kleisli** composition operator we can transform the laws so that they read a bit like the ones we saw before. However, this is a bit complicated and not at all useful for our purpose. If you want to read more about it, I can direct you to https://wiki.haskell.org/Monad_laws.

Left identity

return(x)->bind(f) == f(x)

The law states that, if you take a value, wrap it in the context of the monad, and bind it to *f*, the result has to be the same as if you call the function directly on the value. It ensures that the `bind` method has no side effects on the function and the value besides applying it.

This can only be true because, contrary to the `apply` method, the `bind` method does not wrap the return value of the function inside the monad again. This is the work of the function.

Right identity

m->bind(return) == m

This law states that, if you bind the returned value to a monad, you will get your monad back. It ensures that return has no effect than putting the value inside the context of the monad.

Associativity

m->bind(f)->bind(g) == m->bind((function($x) { f($x)->bind(g); })

This laws states that you can either bind the value inside the monad first to *f* then to *g* or you can bind it to the composition of the first function with the second. We need an intermediary function to simulate this like we needed one in the interchange law for applicatives.

This law allows us the same benefits as previous associative and composition laws. The form is a bit strange because the monad holds the value and not the function or operator.

Validating our monads

Let's write a function to check a monad validity:

```
<?php

function check_monad_laws($x, Monad $m, callable $f, callable
$g)
{
    return [
        'left identity' => $m->return($x)->bind($f) == $f($x),
        'right identity' => $m->bind([$m, 'return']) == $m,
        'associativity' =>
            $m->bind($f)->bind($g) ==             $m-
>bind(function($x) use($f, $g) { return $f($x)->bind($g); }),
    ];
}
```

We also need an identity monad:

```
class IdentityMonad extends Monad
{
    private $value;

    private function __construct($value)
    {
        $this->value = $value;
    }

    public static function pure($value): Applicative
    {
        return new static($value);
    }

    public function get()
```

```
    {
        return $this->value;
    }

    public function bind(callable $f): Monad
    {
        return $f($this->get());
    }

    public function apply(Applicative $a): Applicative
    {
        return static::pure($this->get()($a->get()));
    }
}
```

And we can finally verify that everything holds:

```
<?php

print_r(check_monad_laws(
    10,
IdentityMonad::return(20),
    function(int $a) { return IdentityMonad::return($a + 10);
},
    function(int $a) { return IdentityMonad::return($a * 2); }
));
// Array
// (
//     [left identity] => 1
//     [right identity] => 1
//     [associativity] => 1
// )
```

Why monads?

The first reason is a practical one. When you apply a function using an applicative, the result is automatically put inside the context of the applicative. This means that, if you have a function returning an applicative and you apply it, the result will be an applicative inside an applicative. Anyone who has seen the film *Inception* knows it is not always a great idea to put something, inside something, inside something.

Monads are a way to avoid this unnecessary nesting. The `bind` function delegates the task of encapsulating the return value to the function, meaning you will only have one level of depth.

Monads are also a way to perform flow control. As we've seen, functional programmers tend to avoid using loops or any other kind of control flow, such as the `if` conditions that make your code harder to reason upon. Monads are a powerful way to sequence transformation in a really expressive way while keeping your code clean.

Languages such as Haskell also have specific syntactic sugar to work with monads, such as the `do` notation, which makes your code even easier to read. Some people have tried implementing such a thing in PHP without much success in my opinion.

However, to truly understand the power of the monad abstraction you have to look at some specific implementations, as we will do in the next chapter. They will allow us to perform *IO* operation in a purely functional way, pass log messages from one function to another, and even compute random numbers with a pure function.

Another take on monads

We decided to implement our `Monad` class leaving both the `apply` and `bind` methods abstract. We had no other choice as the way the value is stored inside the `Monad` class will only be decided in the `child` class.

However, as we already said, The `bind` method is sometimes called flatMap in Scala, for example. As the name implies, this is nothing but the combination of a map and a function called `flatten`.

Do you get where I am going with this? Remember the issue with nested applicatives? We could add a `flatten` function, or join as Haskell calls it, method to our `Monad` class, and instead of having `bind` as an abstract method, we could implement it using `map` and our new method.

We will still have two methods to implement, but instead of both doing

roughly the same job, calling a function with a value, one will continue to do that and the other one will be in charge of un-nesting the `Monad` instances.

As such a function has only limited use for the outside world, I decided to go with the presented implementation. Doing it with the `flatten` function instead is a nice exercise that you can try to solve to get a better understanding of how monad works.

A quick monad example

Imagine that we need to read the content of a file using the `read_file` function and then send it to a **webservice** using the post function. We will create two versions, as follows, of the upload function to do just that:

- The first version will use traditional functions, returning a Boolean `false` in case of an error.
- The functional version will assume curryied functions returning an instance of the `Either` monad. We will describe this monad further in the next chapter; let's just assume it works like the `Either` type we already saw earlier.

In the case of success, the given callback must be called with the status code returned by the `post` method:

```
<?php

function upload(string $path, callable $f) {
    $content = read_file(filename);
    if($content === false) {
        return false;
    }

    $status = post('/uploads', $content);
    if($status === false) {
        return $false;
    }

    return $f($status);
}
```

And now the functional version, as follows:

```
<?php

function upload_fp(string $path, callable $f) {
    return Either::pure($path)
      ->bind('read_file')
      ->bind(post('/uploads'))
      ->bind($f);
}
```

I don't know which one you prefer, but my choice is clear. The choice of using `Either` instead of `Maybe` is also not innocent. It means the functional version can also return a detailed error message instead of just `false` in case of error.

Further reading

If after completing this chapter you still find yourself a bit lost, as this is such an important topic, don't hesitate to read one of the following articles or some others you found yourself:

- A small introduction to monads in PHP also with a related library at http://blog.ircmaxell.com/2013/07/taking-monads-to-oop-php.html.
- A good introduction to Scala, which anyone who has written some Java should understand, at https://medium.com/@sinisalouc/demystifying-the-monad-in-scala-cc716bb6f534.
- A video with a more mathematical approach at https://channel9.msdn.com/Shows/Going+Deep/Brian-Beckman-Dont-fear-the-Monads.
- A humorous JavaScript tutorial on monads. You either love or hate the style. If you are fluent with JavaScript, I can only recommend you read the whole book: https://drboolean.gitbooks.io/mostly-adequate-guide/content/ch9.html.
- A really complete, if somewhat difficult, introduction to monads. Some basic Haskell knowledge is needed to understand the explanations at https://wiki.haskell.org/All_About_Monads.

Summary

This chapter was certainly a mouthful, but fear not, it was the last of its kind. From now on, we will tackle more practical topics with real-life applications. Chapter 6, *Real life monads* will present some useful uses of the abstractions we just learned about.

Abstractions, such as functors, applicative, and monads, are the design patterns of the functional world. They are high-level abstractions that can be found in a lot of different places and you will need some time to be able to discern them. However, once you will get the feel for them, you will probably realize that they are everywhere and it will greatly help you to think in terms of manipulating data.

The laws that preside over our abstractions are really prevalent. You probably assume them already when writing code without noticing it. Being able to recognize the patterns we've learned about will bring you more confidence when doing refactoring or writing algorithms because what your instinct always suspected will be backed up by facts.

If you want to play around with this chapter's concepts, I can only recommend you start playing with the `functional-php` library we presented in Chapter 3, *Functional basis in PHP*. It contains a lot of interfaces defining various algebraic structures, the fancy name given by mathematicians to functors, monads, and such. Some of the method names will not be exactly the ones we used, but you should be well equipped to understand the idea behind them. As the library name makes it a bit hard to find, here is the link once more, https://github.com/widmogrod/php-functional.

Chapter 6. Real-Life Monads

In the previous chapter, we covered a lot of theoretical ground concerning various abstractions, leading us to the concept of monads. It is now time to apply this knowledge by presenting a few instances of monads that will prove useful in your day-to-day coding.

Each part will start with an introduction to the issue solved by the given monad, followed by some examples of usage so that you can gain some practice. As explained at the end of this introduction, the implementation itself won't be present in the book to concentrate on usage.

As you will see, once you get the theory out of the way, most implementation will seem pretty natural to you. Also, the usefulness extends beyond the realm of functional programming. Most of what we will learn in this chapter can be applied in any development context.

Most of the monads that will be presented relate to the management of side effects, or rather affects once they are explicitly contained inside a Monad. When doing functional programming, a side effect is unwanted. Once contained, we can control them so that they are merely effects of our program.

Monads are used for mainly two reasons. The first is that they are great to perform flow control, as already explained in the last chapter. The second is that their structure allows you to easily encapsulate effects and protect the rest of your code from the impurity.

However, let's keep in mind that this is only one of the possible uses of a monad. You can do much more with the concept. But let's not get ahead of ourselves; we will discover that along the way.

In this chapter, we will cover the following topics:

- Monadic helper methods
- The Maybe and Either monads

- The List monad
- The Writer monad
- The Reader monad
- The State monad
- The IO monad

In order to concentrate on using monads, and since the implementation is often not the most important part, we will use the ones provided by the **PHP Functional** library. Obviously, important implementation details will be highlighted in the book. You can install it in your project using the `composer` invocation:

```
composer require widmogrod/php-functional
```

It is important to note that the `php-functional` library's author made some other choices concerning the naming of the methods and some implementation details:

- The `apply` method is simply `ap`
- The `unit` and `return` keywords are replaced by `of` in the class
- The inheritance tree is a bit different, for example, there are `Pointed` and `Chain` interfaces
- The library uses traits for shared code
- Some helper functions are implemented outside of the class and need to be imported separately

Monadic helper methods

In the previous chapter, we spoke about the `flatten` method and how it can be used to compress multiple nested levels of the same monad instance. This function is often spoken about because it can be used to rewrite the monad in another light. There are, however, other helpful helpers.

The filterM method

Filtering is a key concept in functional programming, but what if our filter functions return a monad instead of a simple Boolean value? This

is what the `filterM` method is for. Instead of expecting a predicate that returns a simple Boolean value, the `filterM` method will use any predicate that can be converted to a Boolean and also wrap the resulting collection in the same monad:

```
<?php

use function Functional\head;
use function Functional\tail;

use Monad\Writer;

function filterM(callable $f, $collection)
{
    $monad = $f(head($collection));

    $_filterM = function($collection) use($monad, $f,
&$_filterM){
        if(count($collection) == 0) {
            return $monad->of([]);
        }

        $x = head($collection);
        $xs = tail($collection);

        return $f($x)->bind(function($bool) use($x, $xs,
$monad, $_filterM) {
            return $_filterM($xs)->bind(function(array $acc)
use($bool, $x, $monad) {
                if($bool) {
                    array_unshift($acc, $x);
                }

                return $monad->of($acc);
            });
        });
    };
    return $_filterM($collection);
}
```

The implementation is a bit hard to follow, so I'll try to explain what is going on:

1. First, we need to have the information about the monad we are working with, so we extract the first element of our collection and

get the monad from it by applying the callback.
2. We then declare a closure around the monad and the predicate.
3. The closure first tests whether the collection is empty. If this is the case, we return an instance of the monad containing an empty array. Otherwise, we run the predicate on the first element of the collection.
4. We bind a closure holding the current value to the resulting monad containing a Boolean.
5. The second closure recursively traverses the whole array, if needed.
6. Once we are on the last element, we bind a new closure that will use the Boolean to add the value to the accumulator or not.

This is not easy stuff, but as it is mostly internal plumbing, made more difficult by PHP's lack of syntactic sugar, understanding everything is not necessary at all. For comparison, here is the same code implemented using Haskell pattern matching and *do notation* features:

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
filterM _ []     = return []
filterM f (x:xs) = do
    bool <- f x
    acc  <- filterM p xs
    return (if bool then x:acc else acc)
```

As you can see, this is a bit easier to read. I think anyone would be able to understand what is going on. Unfortunately, in PHP, we have to create nested inner functions to achieve the same results. This is, however, not really a concern, since the resulting function is fairly easy to use. The inner working of some functional patterns might, however, sometimes a bit gruesome in PHP and not perfectly functional themselves.

Examples will follow as we discover some monads. An implementation of this helper is available in the `php-functional` library.

The foldM method

The `foldM` method is the monadic version of the `fold` method. It takes a function that returns a monad and then produces a value that is also a

monad. The accumulators and collection are, however, simple values:

```
<?php

function foldM(callable $f, $initial, $collection)
{
    $monad = $f($initial, head($collection));

    $_foldM = function($acc, $collection) use($monad, $f,
&$_foldM){
        if(count($collection) == 0) {
            return $monad->of($acc);
        }

        $x = head($collection);
        $xs = tail($collection);

        return $f($acc, $x)->bind(function($result)
use($acc,$xs,$_foldM) {
            return $_foldM($result, $xs);
        });
    };

    return $_foldM($initial, $collection);
}
```

The implementation is a tad smaller than the one for the `filterM` method because we only need to recurse; no transformation from Boolean to the value needs to happen. Again, we will show a few examples in the following parts of the chapter and an implementation is also available in the `php-funcational` library.

Closing words

There exist multiple other functions that can be enhanced to be used with monadic values. For example, you can have the `zipWithM` method, which merges two collections using a merge function returning a monad. The `php-functional` library has an implementation of `mcompose` which allows you to compose functions returning the same monad instance.

When you discover some kind of recurring pattern when you are using monads, don't hesitate to factor it into a helper function. It will probably

come in handy more often than not.

Maybe and Either monads

You should already be well aware of the Maybe and Either types we have discussed multiple times already. We first defined them, then we learned that they are in fact perfect examples of a functor.

We will now go even further and define them as monads, so we will be able to use them in even more situations.

Motivation

The `Maybe` monad represents the idea that a sequence of computation can, at anytime, stop returning a meaningful value using the `Nothing` class we defined in an earlier chapter. It's particularly useful when chain of transformations depend on one another and where some step may fail to return a value. It allows us to avoid the dreaded `null` checks that often come with such a situation.

The `Either` monad has mostly the same motivation. The slight difference is that the steps usually either throw an exception or return an error instead of an empty value. The fact that the operation fails entails that we need to store an error message symbolized by the `Left` value instead of the `Nothing` value.

Implementation

The code for both Maybe and Either types can be found in `php-functional` library. The implementation is pretty straightforward-the major difference from our own previous implementation is that methods such as `isJust` and `isNothing` are missing, and that instances are constructed using helper functions instead of static factories.

It is important to note that the Either monad as implemented in `php-functional` library sadly does not take care of catching exceptions itself. The functions you are either applying or binding to it must take care to do so correctly themselves. You can also use the `tryCatch` helper

function to do so for you.

Examples

To get a better grasp of how `Maybe` monad works, let's have a look at a few examples. `php-functional` library uses helper functions instead of static methods on the class to create new instances. They live in the `Widmogrod\Monad\Maybe` namespace.

Another really useful helper is the `maybe` method, which is a curryied with the following signature-the `maybe($default, callable $fn, Maybe $maybe)` namespace. When called, it will first try to extract the value from `$maybe` variable, defaulting to a `$default` variable. It will then be passed as a parameter to `$fn` variable:

```
<?php

use Widmogrod\Monad\Maybe as m;
use Widmogrod\Functional as f;

$just = m\just(10);
$nothing = m\nothing();

$just = m\maybeNull(10);
$nothing = m\maybeNull(null);

echo maybe('Hello.', 'strtoupper', m\maybe('Hi!'));
// HI!

echo maybe('Hello.', 'strtoupper', m\nothing());
// HELLO.
```

Now that the helpers are out of the way, we will demonstrate how `Maybe` monad can be used in combination with the `foldM` method:

```
<?php

$divide = function($acc, $i) {
    return $i == 0 ? nothing() : just($acc / $i);
};

var_dump(f\foldM($divide, 100, [2, 5, 2])->extract());
```

```
// int(5)

var_dump(f\foldM($divide, 100, [2, 0, 2])->extract());
// NULL
```

Implementing this using a traditional function and the `array_reduce` method would mostly result in something really similar, but it demonstrates nicely how the `foldM` method works. Since the folding function is bound to the current monadic value on each iteration, as soon as we have a null value, the following steps will just continue returning nothing until the end. The same function could be used to return some other kind of monad to also hold information about the failure.

We already saw before how the monad type can be used to chain multiple functions together on a value that may or may not exist. However, if we need to use this value to get another value that could be nothing, we will have nested `Maybe` instances:

```
<?php

function getUser($username): Maybe {
  return $username == 'john.doe' ? just('John Doe') :
nothing();
}

var_dump(just('john.doe')->map('getUser'));
// object(Monad\Maybe\Just)#7 (1) {
//     ["value":protected]=> object(Monad\Maybe\Just)#6 (1) {
//         ["value":protected]=> string(8) "John Doe"
//     }
// }

var_dump(just('jane.doe')->map('getUser'));
// object(Monad\Maybe\Just)#8 (1) {
//     ["value":protected]=> object(Monad\Maybe\Nothing)#6 (0)
{ }
// }
```

In this case, you could use the `flatten` method, or simply the `bind` method instead of the `map` method:

```php
<?php

var_dump(just('john.doe')->bind('getUser'));
// object(Monad\Maybe\Just)#6 (1) {
//     ["value":protected]=> string(8) "John Doe"
// }

var_dump(just('jane.doe')->bind('getUser'));
// object(Monad\Maybe\Nothing)#8 (0) { }
```

I agree that the examples for `Maybe` monad are a bit anticlimactic as most of the uses were already described earlier monads are only a pattern, thus creating a `Maybe` monad does not add feature in itself, it will only allow us with other patterns expecting a monad; the features stay the same as before.

A similar case can be made for `Either` monad; this is why there won't be any new examples for it here. Just make sure to have a look at the helper functions instead of rewriting the plumbing yourself when you want to use the monad.

List monad

The List or Collection monad represents the category of all functions taking a collection as a parameter and returning zero, one, or several values. The function is applied to all possible values in the input list and the results are concatenated to product a new collection.

An important thing to understand is that a list monad does not really represent a simple list of values, but rather a list of all different possible values for the monad. This idea is often described a *non-determinism*. As we saw with the `CollectionApplicative` function, this can lead to interesting results when you apply a collection of functions with a collection of values. We will try to expand on this topic in the examples to clarify this.

Motivation

The List monad embodies the idea that you cannot know the best result until the end of the full computation. It allows us to explore all possible solutions until we have a final one.

Implementation

The monad is implemented in the `php-functional` library under the name the `Collection` method. It is done in a pretty straightforward way. Two new methods are, however, available in comparison to our own previous implementation:

- The `reduce` method will perform a fold operation on the values stored inside the monad.
- The `traverse` method will map a function returning an applicative to all values stored inside the monad. The applicative is then applied to the current accumulator.

Examples

Let's start with something hard, using the `filterM` method that we

previously discovered. We will create something that is called the `powerset` of a set. The `powerset` collection is all possible subsets of a given set, or, if you like, all possible combination of its members:

```
<?php

use Monad\Collection;
use Functional as f;

$powerset = filterM(function($x) {
    return Collection::of([true, false]);
}, [1, 2, 3]);

print_r($powerset->extract());
// Array (
//     [0] => Array ( [0] => 1 [1] => 2 [2] => 3 )
//     [1] => Array ( [0] => 1 [1] => 2 )
//     [2] => Array ( [0] => 1 [1] => 3 )
//     [3] => Array ( [0] => 1 )
//     [4] => Array ( [0] => 2 [1] => 3 )
//     [5] => Array ( [0] => 2 )
//     [6] => Array ( [0] => 3 )
//     [7] => Array ( ) // )
```

Note

This currently doesn't work with the actual implementation of Collection/filterM due to the fact that the constructor does not wrap an actual array inside another. See https://github.com/widmogrod/php-functional/issues/31.

What is happening here? It may seem as if it is some kind of dark magic. In fact, it is pretty simple to explain. Binding a function to a collection results in this function being applied to all its members. In this particular case, our filtering function returns a collection containing both `true` and `false` values. This means the inner closure of the `filterM` method responsible for replacing the Boolean with the value is run twice and the result is then appended to all previously created collections. Let's see the first steps to make things clearer:

1. The filter is first applied to the value `1`, creating two collections `[]` and `[1]`.

2. The filter is now applied to the value `2`, creating two new collections (`[]` and `[2]`) that need to be appended to the ones we created earlier, creating four collections `[]`, `[1]`, `[2]`, `[1, 2]`.
3. Each new step creates two collections that are appended to the previous ones, making the number of collections grow exponentially.

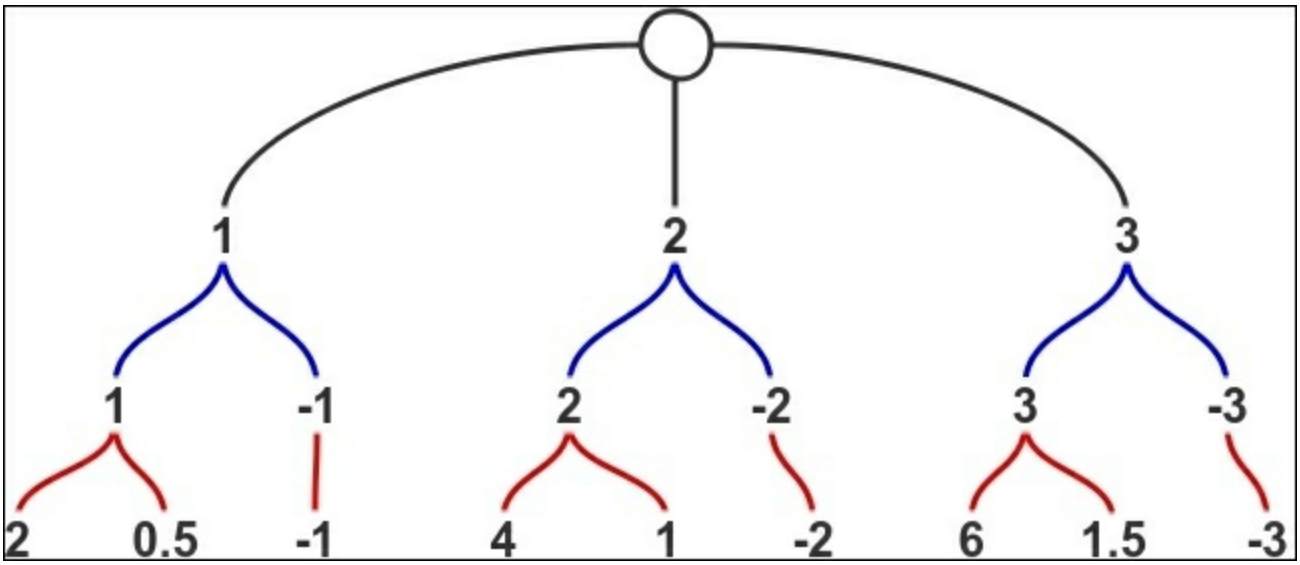
Still not clear? Let's look at another example. This time, try imagining the collection as a tree where each initial value is a branch. When you bind a function, it is applied to each branch and, if the result is another collection, it creates new branches:

```
<?php
use Monad\Collection;
use Functional as f;

$a = Collection::of([1, 2, 3])->bind(function($x) {
    return [$x, -$x];
});
print_r($a->extract());
// Array (
//     [0] => 1
//     [1] => -1
//     [2] => 2
//     [3] => -2
//     [4] => 3
//     [5] => -3
// )

$b = $a->bind(function($y) {
    return $y > 0 ? [$y * 2, $y / 2] : $y;
});
print_r($b->extract());
// Array (
//     [0] => 2
//     [1] => 0.5
//     [2] => -1
//     [3] => 4
//     [4] => 1
//     [5] => -2
//     [6] => 6
//     [7] => 1.5
//     [8] => -3
// )
```

In order to make matters a bit more complicated for you, the second function returns a variable number of elements based on the given value. Let's visualize this as a tree:



Where can the knight go?

Now that we have a good understanding of how the `Collection` monad works, let's tackle a more difficult challenge. Given a starting position on a chessboard, we want to know all possible valid positions a knight piece can reach in three moves.

I want you to take a moment to imagine how you would implement that. Once you are done, let's try using our monad. We first need a way to encode our knight position. A simple class will suffice. Also, a chessboard has eight columns and eight rows, so let's add a method to check whether the position is valid:

```
<?php

class ChessPosition {
    public $col;
    public $row;

    public function __construct($c, $r)
    {
```

```
        $this->col = $c;
        $this->row = $r;
    }

    public function isValid(): bool
    {
        return ($this->col > 0 && $this->col < 9) &&
               ($this->row > 0 && $this->row < 9);
    }
}

function chess_pos($c, $r) { return new ChessPosition($c, $r);
}
```

Now we need a function that returns all valid moves for a knight, given a starting position:

```
<?php

function moveKnight(ChessPosition $pos): Collection
{
    return Collection::of(f\filter(f\invoke('isValid'),
Collection::of([
        chess_pos($pos->col + 2, $pos->row - 1),
        chess_pos($pos->col + 2, $pos->row + 1),
        chess_pos($pos->col - 2, $pos->row - 1),
        chess_pos($pos->col - 2, $pos->row + 1),
        chess_pos($pos->col + 1, $pos->row - 2),
        chess_pos($pos->col + 1, $pos->row + 2),
        chess_pos($pos->col - 1, $pos->row - 2),
        chess_pos($pos->col - 1, $pos->row + 2),
    ])));
}

print_r(moveKnight(chess_pos(8,1))->extract());
// Array (
//     [0] => ChessPosition Object ( [row] => 2 [col] => 6 )
//     [1] => ChessPosition Object ( [row] => 3 [col] => 7 )
// )
```

Nice, it seems to be working well. Now all we need to do is bind this function three times in a row. And, while we are at it, we will also create a function that checks whether a knight can reach a given position in three steps:

```php
<?php

function moveKnight3($start): array
{
    return Collection::of($start)
        ->bind('moveKnight')
        ->bind('moveKnight')
        ->bind('moveKnight')
        ->extract();
}

function canReach($start, $end): bool
{
    return in_array($end, moveKnight3($start));
}

var_dump(canReach(chess_pos(6, 2), chess_pos(6, 1)));
// bool(true)

var_dump(canReach(chess_pos(6, 2), chess_pos(7, 3)));
// bool(false)
```

The only thing left to do is to check on a real chessboard whether our functions work correctly. I don't know how you imagined doing this in an imperative way, but my own solution for once was a lot less elegant than the one we've got here.

If you want to play a bit more, you can try to parametrize the number of moves or implement this for other chess pieces. As you will see, it requires only minimal changes.

Writer monad

If you remember, pure functions cannot have any side effects, meaning you cannot put a debug statement in them, for example. If you are like me, the `var_dump` method is your debugging tool of choice, so you are left with breaking your purity rule or using some other debugging techniques. Since all outputs of a function must go through its return value, one of the first ideas that comes to mind is to return a tuple of values-the original return value and any kind of debug statement you need.

This solution is, however, pretty complex to put in place. Imagine you have a function that halves a numerical value which returns the halved value and the received input for debugging purposes. Now, if you want to compose this function by itself to create a new function that returns the value divided by four, you also need to modify the inputs so that they can accept your new return format. And this goes on and on until you've modified all your functions accordingly. This also poses some issues with currying, as you now have an extraneous parameter which is not really useful if you don't care about the debug statements.

The solution you are looking for is the Writer monad. Sadly, there are no implementations in `php-functional` library at the time of writing.

Motivation

The Writer monad is used to encapsulate some kind of associated statement alongside the principal return value of a function. This statement can be anything. It is often used to store generated debugging output or tracing information. Doing so manually is cumbersome and can lead to complex management code.

The Writer monad provides a clean way to manage such side output and allows you to interleave functions returning such information alongside functions returning simple values. At the end of the computation sequence, the supplementary values can be either discarded, displayed,

or treated in any kind of way depending on the mode of operation.

Implementation

Since the monad needs to concatenate the output values, any instance of a monoid can be used as such. To simplify string-based logging, any string is also managed out-of-the-box. Obviously, using a monoid with a slow operation will result in a performance cost.

The `php-functional` library includes an implementation of a `StringMonoid` class to which each string will be lifted. However, the `runWriter` method will always return a `StringMonoid` class, so there is no surprise for people using it. Besides that, the implementation is pretty straightforward.

Examples

As we just saw, the `Writer` is great for logging. Coupled with the `filter` method, this can be leveraged to understand what is happening in a filtering function without having to resort to dumping values:

```
<?php

$data = [1, 10, 15, 20, 25];
$filter = function($i) {
    if ($i % 2 == 1) {
        return new Writer(false, "Reject odd number $i.\n");
    } else if($i > 15) {
      return new Writer(false, "Reject $i because it is bigger
than 15\n");
    }

    return new Writer(true);
};

list($result, $log) = filterM($filter, $data)->runWriter();

var_dump($result);
// array(1) {
//   [0]=> int(10)
// }
```

```
echo $log->get();
// Reject odd number 1.
// Reject odd number 15.
// Reject 20 because it is bigger than 15
// Reject odd number 25.
```

As we can see, `Writer` monad allows us to have exact information about why certain numbers were filtered out. It may seem like nothing in such a simple example, but conditions are not always as easy to understand.

You can also use `Writer` to add more traditional debug information:

```
<?php

function some_complex_function(int $input)
{
    $msg = new StringMonoid('received: '.print_r($input,
true).'.');

    if($input > 10) {
        $w = new Writer($input / 2, $msg->concat(new
StringMonoid("Halved the value. ")));
    } else {
        $w = new Writer($input, $msg);
    }

    if($input > 20)
    {
        return $w->bind('some_complex_function');
    }

    return $w;
}

list($value, $log) = (new Writer(15))-
>bind('some_complex_function')->runWriter();
echo $log->get();
// received: 15. Halved the value.

list($value, $log) = some_complex_function(27)->runWriter();
echo $log->get(); // received: 27. Halved the value. received:
13. Halved the value.

list($value, $log) = some_complex_function(50)->runWriter();
echo $log->get();
```

```
// received: 50. Halved the value. received: 25. Halved the
value. received: 12. Halved the value.
```

This monad is great keeping track of useful information. Also, it often avoids leaving some unwanted `var_dump` or `echo` methods in your function and library code. Once you are done debugging, leave the messages there-they might prove useful to someone else, and just remove the use of the `$log` value returned by the `runWriter` method.

Obviously, you can also use `Writer` monad to keep track of any kind of information. One good use could be to back profiling right into your function by always returning the execution time via a `Writer` instance.

If you need to store multiple kinds of data, the `Writer` monad is not limited to string values, any monoid will do. You can, for example, declare a specific monoid containing execution time, stack trace, and debug messages in an array and use that with your Writer. This way, each of your functions will be able to pass useful information to anyone calling them.

We could argue that it slows your program down always having that kind of information. This is probably correct, but I'd imagine that those kind of optimizations are not needed in most applications.

Reader monad

It so happens that you have a bunch of functions that should all take the same parameter, or a subset of a given list of values. For example, you have a configuration file and various parts of your application need to have access to values stored in it. One solution is to have some kind of global object or singleton to store that information, but as we already discussed, this leads to some issues. A more common approach in modern PHP frameworks is to use a concept called **Dependency Injection** (**DI**). The Reader monad allows you to do exactly that in a purely functional way.

Motivation

Provide a way to share a common environment, such as configuration information or class instances, across multiple functions. This environment is read-only for the computation sequence. However, it can be modified or extended for any sub-computation local to the current step.

Implementation

The `Reader` class performs function evaluation lazily because the content of the environment is not yet known when the function is bound. This means all functions are wrapped inside closures inside the monad and everything is run when the `runReader` method is called. Besides that, the implementation available in `php-functional` library is pretty straightforward.

Examples

Using the `Reader` monad is a bit different than what we have seen until now. The bound function will receive the value from the previous step in the computation and must return a new reader that holds a function receiving the environment. If you just want to process the current value, it is easier to use the `map` function, as it does not require a `Reader`

instance to be returned. You will, however, not receive the context:

```
<?php
function hello()
{
    return Reader::of(function($name) {
        return "Hello $name!";
    });
}

function ask($content)
{
    return Reader::of(function($name) use($content) {
        return $content.
               ($name == 'World' ? '' : ' How are you ?');
    });
}

$r = hello()
      ->bind('ask')
      ->map('strtoupper');

echo $r->runReader('World');
// HELLO WORLD!

echo $r->runReader('Gilles');
// HELLO GILLES! HOW ARE YOU ?
```

This not-so-interesting example just poses the basics of what you can do. The next example will show how you can perform DI using this monad.

Note

If you've used a modern web framework, you probably already know what dependency injection, or DI, is. Otherwise, here is a real quick explanation, for which I could probably get burned at the stake. DI is a pattern to avoid using singletons or globally available instances. Instead, you declare your dependencies as functions or constructor parameters and a **Dependency Injection Container** (**DIC**) is tasked with providing them to you.

Usually, this involves letting the DIC instantiate all your objects instead of using the `new` keyword, but the method varies from one framework to

another.

How do we do that using the `Reader` monad? It's pretty simple. We need to create a container that will hold all our services and then we will use our reader to pass those around.

For the sake of the example, let's say we have an `EntityManager` for our users that connects to the database and a service to send e-mails. Also, to keep things simple, we won't do any encapsulation, and we will use simple functions instead of classes:

```
<?php

class DIC
{
    public $userEntityManager;
    public $emailService;
}

function getUser(string $username)
{
    return Reader::of(function(DIC $dic) use($username) {
        return $dic->userEntityManager->getUser($username);
    });
}

function getUserEmail($username)
{
    return getUser($username)->map(function($user) {
        return $user->email;

    });
}

function sendEmail($title, $content, $email)
{
    return Reader::of(function(DIC $dic) use($title, $content,
$email) {
        return $dic->emailService->send($title, $content,
$email);
    });
}
```

Now we want to write the controller that gets called after a user registers

on our application. We will need to send them an e-mail and display some kind of confirmation. For now, let's assume the user is already saved in the database and that our theoretical framework provides the use of the `POST` method values as a parameter:

```
<?php

function controller(array $post)
{
    return Reader::of(function(DIC $dic) use($post) {
        getUserEmail($post['username'])
            ->bind(f\curry('sendEmail', ['Welcome', '...']))
            ->runReader($dic);

        return "<h1>Welcome !</h1>";
    });
}
```

OK, we have everything in place for a quick test. We will create some face service classes to see whether the plumbing works correctly:

```
<?php

$dic = new DIC();
$dic->userEntityManager = new class() {
    public function getUser() {
      return new class() {
          public $email = 'john.doe@email.com';
      };
    }
};

$dic->emailService = new class() {
    public function send($title, $content, $email) {
        echo "Sending '$title' to $email";
    }
};

$content = controller(['username' => 'john.doe'])-
>runReader($dic);
// Sending 'Welcome' to john.doe@email.com

echo $content;
// <h1>Welcome !</h1>
```

Obviously, we don't have a usable framework yet, but I think this demonstrates nicely the possibilities offered by the `Reader` monad when it comes to DI.

Concerning the IO operations that need to be done to store the newly created user in the database and the mail sending, we will see how it can be achieved using the IO monad that we will present later.

State monad

The State monad is a generalization of the reader monad in the sense that each step can modify the current state before the next step is called. As a referentially transparent language cannot have a shared global state, the trick is to encapsulate the state inside the monad and pass it explicitly to each part of the sequence.

Motivation

It provides a clean and easy-to-use process to pass a shared state across multiple steps in a sequence. This can obviously be done manually but the process is error prone and leads to less readable code. The monad hides the complexity so that you can simply write functions taking a state as input and returning a new state.

Implementation

The implementation available in the `php-functional` library is nearly identical to the one we just discussed for the `Reader` monad, with just one key difference-the state can be updated by each bound function. This leads to a difference in the functions that are bound to the monad-instead of returning a value, they need to return an array containing the value as first element and the new state as second element.

Examples

As we already discussed, it is impossible for a function to return the current time or some kind of random value. The `state` monad can help us do exactly this by providing a clean way to pass the `state` variable around, exactly as we did with our `Reader` environments earlier:

```
function randomInt()
{
    return s\state(function($state) {
        mt_srand($state);
        return [mt_rand(), mt_rand()];
    });
```

```
}

echo s\evalState(randomInt(), 12345);
// 162946439
```

Another use of the `state` monad is to implement a caching system:

```
<?php

function getUser($id, $current = [])
{
    return f\curryN(2, function($id, $current) {
        return s\state(function($cache) use ($id, $current) {
            if(! isset($cache[$id])) {
                $cache[$id] = "user #$id";
            }

            return [f\append($current, $cache[$id]), $cache];
        });
    })(...func_get_args());
}

list($users, $cache) = s\runState(
  getUser(1, [])
    ->bind(getUser(2))
    ->bind(getUser(1))
    ->bind(getUser(3)),
  []
);

print_r($users);
// Array (
//     [0] => user #1
//     [1] => user #2
//     [2] => user #1
//     [3] => user #3
// )

print_r($cache);
// Array (
//     [1] => user #1
//     [2] => user #2
//     [3] => user #3
// )
```

As we can see, the user list contains the `user 1` two times, but the cache

only once. This is a pretty basic cache mechanism, but it can come in handy.

There are many other uses for the `state` monad, but to be honest, without syntactic sugar like the do notation and such, I am not quite sure it is a good fit for PHP programming. If you are interested, I am sure you will find many other resources online but we will stop there with the examples.

IO monad

Inputs and outputs are the quintessence of side effects. There is no way to guarantee purity when you get your function output from an external source as those change without relation to the inputs. And as soon as you output something, be it to the screen, a file, or anywhere else, you changed an external state unrelated to your function outputs.

Some people in the functional community argue that, for example, logging outputs or debugging statements should not necessarily be considered as side-effects as usually they have no consequences on the outcome of running your application. The end user doesn't care whether something was written to a log file or not as long as it gets the wanted result back and the operation is repeatable at will. Truthfully, my opinion on the subject is not quite made, and honestly I don't really care as the writer monad lets us take care of logging and debugging statements in a clever way.

However, there are some times when you need to have information from the outside and usually, if your application is doing anything worthy, you need to display or write the final result somewhere.

We could imagine getting all values before beginning any computation and passing them around using some kind of clever data structure. This could work for some simpler applications, but as soon as you need to perform database access based on some computed values, reality starts to set in and you realize that this isn't at all viable in the long term.

The trick proposed by the IO monad is to do what we just proposed but in reverse. You start by describing all computational steps needed by your program. You encapsulate them in an instance of the IO monad and when everything is cleanly defined in terms of referentially transparent function calls, you start the beast which will finally perform all needed IO actions, and call each described step.

This way, you have an application composed only of pure functions,

which you can easily test and understand. All actions related to inputs and outputs are performed at the end, the complexity being hidden inside the IO monad. In order to enforce this, the IO monad is said to be a one-way monad, meaning there is no way to get any value out of it. You have only two options:

- Binding computations, or actions, to the monad so that they get executed later
- Running said computations to get the final result of your application

I imagine this may be pretty confusing if you have never seen an application created like this. The examples will try to give you a first impression of how it can be done and we will dive deeper into the topic in Chapter 11, *Designing a Functional Application*.

Motivation

The IO monad solves the issue of inputs and outputs breaking referential transparency and function purity by confining all IO operations within the monad. All computational steps needed for the application are first described in a functional way. Once this is done, we accept that the final step cannot be side-effect-free and we run all the sequences stored inside the monad.

Implementation

The implementation provided by `php-functional` library is pretty simple, as there are no real subtleties. There is only one little trick needed as the computation needs to be made when `run` method is called and not when the function is bound.

Besides that, the library comes with helper functions under the `Widmogrod\Monad\IO` namespace to help you use the monad. You can easily read input from the user on the command line, print text on the screen, and read and write files and environment variables.

Examples

We will take this opportunity to use the `mcompose` method in order to compose multiple `IO` operations together:

```
<?php

use Widmogrod\Functional as f;
use Widmogrod\Monad\IO;
use Widmogrod\Monad\Identity;

$readFromInput = f\mcompose(IO\putStrLn, IO\getLine,
IO\putStrLn);
$readFromInput(Monad\Identity::of('Enter something and press
<enter>'))->run();
// Enter something and press <enter>
// Hi!
// Hi!
```

So we first create a function that displays the current content of the monad using `putStrLn`, ask for some input, and display the result back.

The `IO` monad needs to wrap the whole computation of your application if you want to maintain referential transparency. This is because your inputs need to be retrieved through it and any output must also be done through the monad. This means we could show a lot of examples without really capturing the real essence of its use. This is why we will stop here and wait until Chapter 11, *Designing a Functional Application*, to see how it can be achieved.

Summary

In this chapter, we have looked at multiple monads and their implementation. I hope the examples made it clear how you can use them and what their benefits are:

- The Maybe monad can be used when a computation might return nothing
- The Either monad can be used when a computation might error
- The List monad can be used when a computation has multiple possible results
- The Writer monad can be used when some side information needs to be passed alongside the return value
- The Reader monad can be used to share a common environment between multiple computations
- The State monad is a beefed-up version of the Reader monad where the environment can be updated between each computation
- The IO monad can be used to perform IO operations in a referentially transparent way

There are, however, multiple other computations that can be simplified using monads. When writing code, I encourage you to take a step back and look at the structure to see if you recognize a monadic pattern. If so, you should probably implement it using our `Monad` class to benefit from what we've learned so far.

Also, those various monads can be used in combination to achieve complex transformations and computations. We will approach this topic in Chapter 10, *PHP Frameworks and FP*, where we will discuss monad transformers, and Chapter 11, *Designing a Functional Application*.

At this point in the book, you are perhaps impressed by some functional techniques but I imagine most of the things we've seen so far are a bit awkward and functional programming might seem tedious. The feeling is totally normal for two main reasons.

First, this awkwardness often results from some kind of missing

abstraction or technique waiting to be discovered. If this were a book about Haskell, you would learn about all of these and you would have a handful of other books to look them up. However, this book is about PHP; we will learn a few more concepts in the later chapters, but after that, you will mostly be on your own, like a pioneer.

I can only encourage you to make your way through when you encounter those situations and look for patterns and ways to factor out commonalities in your code. Step by step, you will forge a great toolbox and things will get easier.

Second, all of this is probably new to you. Switching programming paradigm is really hard and it can be really frustrating. But fear not, with time, practice, and experience, you will gain confidence and the benefits will start to outweigh the cost. The steeper the learning curve, the greater the reward.

In the next chapter, we will discover some new functional concepts and patterns that will permit us to fully use the various techniques we have learned so far.

Chapter 7. Functional Techniques and Topics

We've covered the foundational techniques relative to functional programming. But, as you can probably imagine, there are a lot more topics to cover. In this chapter, you will learn about some of those patterns and ideas.

Some topics will be covered in depth and others will be mentioned with pointers to external resources if you want to learn more. As this is an introductory book to functional programming in PHP, the most advanced ideas are out of scope. If you encounter an article about such a topic somewhere, you should, however, have enough understanding to grasp at least the gist of it.

The sections of this chapter are not necessarily linked with one another. Some content might be novel to you and some is linked to excerpts presented earlier.

In this chapter, we will cover the following topics:

- Type systems, type signature, and their uses
- Point-free style
- Using the `const` keyword to facilitate anonymous function usage
- Recursion, stack overflows, and trampolines
- Pattern matching
- Type classes
- Algebraic structures and category theory
- Monad transformers
- Lenses

Type systems

Disclaimer: I have no intention to launch quarrels between static and dynamic typing aficionados. Discussing which one is better and why is

not at all the objective of this book and I'll let each and every one of you decide what you prefer. If the subject interests you, I can recommend reading http://pchiusano.github.io/2016-09-15/static-vs-dynamic.html, which is a good summary, even if a bit biased in favor of static typing.

That is said, types and typing systems are an important topic in functional programming, even if said types aren't enforced by the language. The types of a function signature are a meta language enabling succinct and effective communication about the purpose of a function.

As we will see, clearly declaring the expected types of the input and output of a function is an important part of its documentation. Not only does it reduce the cognitive burden by allowing you to skip reading the function code, it also allows you to deduce important facts about what is happening and derive *free theorems*.

The Hindley-Milner type system

Hindley-Milner, also known as Damas-Milner or Damas-Hindley-Milner, from the name of the people whom first theorized it, is a type system. A type system is a set of rules that define what type a variable or parameter can have and how different types interact with each other.

One of the main features of Hindley-Milner is that it allows type inference. This means that you often don't have to explicitly define the type; it can be inferred from other sources such as the context or the types of surrounding elements. To be exact, the type inference is done by an algorithm called **Algorithm W** and is related to, but not the same as, the Hindley-Milner type system.

It also allows polymorphism; this means that if you have a function returning the length of a list, the type of the elements of the list doesn't need to be known as it doesn't matter for the computation. It is an idea akin to the Generics that you can find in C++ or Java, but not quite the same as it is more powerful.

Most statically typed functional languages, such as Haskell, **OCaml**, and

F#, use Hindley-Milner as their type system, often with extensions to manage some edge cases. Scala is notable for using its own type system.

Besides the theory about type systems, there is a commonly accepted way to describe the input and output parameters of a function and this is what we are interested in here. It is quite different from the type hints you can use with PHP and it is often put in a comment at the top of a function when the language does not use this particular syntax. As of now we will refer to such type annotation as the *type signature* of a function.

Type signatures

As a first and simple example, we will start with the type signatures of the `strtoupper` and `strlen` PHP functions:

```
// strtoupper :: string -> string
// strlen :: string -> int
```

It's pretty simple to understand: we start with the function name, followed by the type of the parameter, an arrow, and the type of the return value.

What about functions with multiple arguments? Consider the following:

```
// implode :: string -> [string] -> string
```

Why are there multiple arrows? If you think of currying this might help you to get the answer. If we write the same function signature using parentheses, this will probably help you further:

```
// implode :: string -> ([string] -> string)
```

Basically, we have a function that takes a `string` type and returns a new function taking an array of strings and returning a string. The right most type is always the type of the return value. All other types are the various parameters in order. Parentheses are used to denote a function. Let's see how it looks with more parameters:

```
// number_format :: float -> (int -> (string -> (string -> string)))
```

Or without parentheses:

```
// number_format :: float -> int -> string -> string -> string
```

I don't know what your opinion is, but I personally prefer the latter as it is less noisy and once you get used to it, the parentheses don't bring much information.

If you are familiar with the `number_format` function, you might have noted that the type signature I proposed contains all parameters, even the optional ones. This is because there is no canonical way to convey this information, because functional languages usually do not allow such parameters. However, Haskell has an `Optional` data type that is used to emulate this. With this information in mind, arguments with a default value are sometimes displayed like this:

```
// number_format :: float -> Optional int -> Optional string -> Optional string -> string
```

This works fairly well and is self-explanatory, until you have a type called the `Optional` data type. There is also no common way to express what the default value is.

As far as I know, there is no way to convey the information that a function takes a variable number of arguments. Since PHP 7.0 introduced a new syntax for this, I propose we use that for the remainder of this book:

```
// printf :: string -> ...string -> int
```

We saw earlier that parentheses are used to express the idea of functions, but that they were usually not used for readability reasons. However, this is not true when using functions as parameters. In this case, we need to retain the information that the function expects or returns another function:

```
// array_reduce :: [a] -> (b -> a -> b) -> Optional a -> b
```

```
// array_map :: (a -> b) -> ...[a] -> [b]
```

You might be asking yourself, What are those `a` and `b` variables? This is the polymorphism feature we were talking about earlier. The `array_reduce` and `array_map` functions do not care about what the real types of the elements contained in the array are, they don't need to know that information to perform their job. We could have used `mixed` function like we did earlier with the `printf` method, but then we would have lost some useful data.

The `a` variable is a certain type, and the `b` variable is another, or it could also be the same. Variables like `a` and `b` are sometimes called **type variables**. What those type signatures are saying is that we have an array of values having a certain type (the type `a`), a function that takes such a value and returns one having another type (the type `b`); obviously, the final value is the same as the one of the callback.

The names `a` and `b` are a convention, but you are free to use anything you want. In some occasions, using a longer name or some particular letter might help convey more information.

Note

If you have trouble understanding the signature for the `array_reduce` function, this is perfectly normal. You are not familiar with the syntax yet. Let's try to take the arguments one by one:

- An array containing elements of type `a` type
- A function, taking a type `b` (the accumulator), a type `a` (the current value), and returning a type `b` (the new accumulator content)
- An optional initial value of the same type as the array elements
- The return value is of type `b`, the same type as the accumulator

What this signature does not tell us is the exact types of `a` and `b`. For all we care, `b` could be an array itself, a class, a Boolean, really anything. The types `a` and `b` can also be of the same type.

You can also have a unique type variable, as is the case for the

`array_filter` function:

```
// array_filter :: (a -> bool) -> [a] -> [a]
```

Since the type signature only uses the `a` type, this means that the elements in the input array and the returned array will have exactly the same type. Since the `a` type is not a specific type, this type signature also informs us that the `array_filter` function works for all types, meaning it cannot transform the values. The elements inside of the list can only be rearranged or filtered.

One final feature of type signatures is that you can narrow down the possible types for a given type variable. For example, you can specify that a certain type `m` should be a child of a given class:

```
// filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

We just introduced a new double arrow symbol. You will always find it at the beginning of the type signature, never in the middle of one. It means that what comes before defines some kind of specificity.

In our case, we constraint the `m` type variable to be a descendant of the `Monad` class. This allows us to declare that the `filterM` method first takes a function returning a Boolean value enclosed in a monad as a first argument and that its return value will be enclosed in the same monad.

You can specify multiple constraints if you'd like. If we imagine having two types, the `TypeA` and the `Typeb` types, we could have the following type signature:

```
// some_function :: TypeA a TypeB b => a -> b -> string
```

What this function does is not clear just by looking at the type signature, but we know that it expects one instance of the `TypeA` type and one instance of the `TypeB` type. The return value will be a string, obviously the result of a computation based on the parameters.

In this case, we cannot make the same assumption as for the `array_filter` method, namely that no transformation will happen,

because we have a constraint on the type variables. Our functions may very well know how to operate on our data since they are an instance of a certain type or its children.

I know there are a lot of things to take in, but as the preceding the `array_reduce` function example proves, type signatures allow us to encode a lot of information in a concise way. They are also more precise than PHP type hints as they, for example, allow us to say that the `array_map` method can transform from one type to another, whereas the `array_filter` method will maintain the types in the array.

If you've perused the code of the `php-functional` library, you might have noticed that the author used such type signatures in the doc blocks of most of the functions. You will also find that some other functional libraries do the same and the habit is also spreading in the JavaScript world for example.

Free theorems

Not only do type signatures give us insight in to what the function does, they also allow us to deduce useful theorems and rules based on the type information. These are called **free theorems** because they come for free with the type signature. The idea was developed by Philip Walder in the paper *Theorems for free!* published in 1989.

By using a free theorem, we can affirm the following:

```
// head :: [a] -> a
// map :: (a -> b) -> [a] -> [b]
head(map(f, $x)) == f(head($x)
```

This might seem obvious to you using a bit of common sense and knowing what the functions do, but a computer lacks common sense. So, in order to optimize the left hand of our equality to the right hand, our compiler or interpreter must rely on the free theorem to do so.

How can the type signatures prove this theorem? Remember when we said that the type `a` is a generic type that our functions know nothing

about? The corollary is that they cannot modify the values inside the array because such a generic function does not exist. The only function that knows how to transform something is `f` because it needs to conform to the `(a -> b)` type signature enforced by map.

Since nor `head` nor `map` functions modify the elements, we can deduce that first applying the function and then taking the first element is exactly the same as taking the first element and applying the function. Only the second way is much faster:

```
// filter :: (a -> Bool) -> [a] -> [a]
map(f, filter(compose(p, f), $x)) === filter(p, map(f, $x))
```

A bit more complicated, this free theorem says that if your predicate needs a value transformed using the function `f` and then you apply the `f` function on the result it is exactly the same as first applying `f` to all elements and filtering afterward. Again the idea is to optimize for performances by applying `f` only one time instead of two.

When composing functions together or calling them after one another, try to look at the type signatures to see if you cannot improve your code by deducing some free theorems.

Haskell users even have a free theorem generator at their disposition at http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi.

Closing words

Type signatures bring so much to the table that you can find function search engines based on them. Sites such as https://www.haskell.org/hoogle/ and http://scala-search.org/ lets you search for functions based solely on their type signature.

When working with functional techniques, it also often happens that you have a certain structure for your data and you need to transform that to something else. Since most functions are totally generic, it is often complicated coming up for the right keyword to search for what you are looking for. This is also where type signatures and search engines like

Hoogle come in handy. Just enter the type structure of your input, the wanted type of the output, and peruse the function list returned by the search engine.

PHP being a dynamically typed language and only having recently introduced scalar typing, useful tools around type signatures obviously don't yet exist. But maybe it is only a matter of time before people will come up with something.

Point-free style

Point-free style, also called tacit programming is a way of writing functions where you don't explicitly define the parameters or points, hence the name. Depending on the language, it is possible to apply this particular style on different levels. How can you have functions without identified arguments? By using function composition or currying.

In fact, we already did some point-free style programming before in this book. Let's use an example from Chapter 4, *Composing Function*, to illustrate what it is about:

```
<?php
// traditional
function safe_title(string $s)
{
    return strtoupper(htmlspecialchars($s));
}

// point-free
$safe_title = compose('htmlspecialchars', 'strtoupper');
```

The first part is a traditional function where you declare an input parameter. In the second case, however, you can see no explicit parameter declarations; you rely on the definitions of the composed functions. The second function is said to be in point-free style.

PHP syntax forces us to assign our composed or curryied function to a variable, but there is no such clear separation in some other languages. Here are three examples in Haskell:

```
-- traditional
sum (x:xs) = x + sum xs
sum [] = 0

-- using foldr
sum xs = foldr (+) 0 xs

-- point-free
sum = foldr (+) 0
```

As we can see, the structure of the function definition is the same in the three cases. The first one is how you would define the `sum` method without using folding. The second example acknowledges that we can simply fold on the data, but still keeps the argument explicitly declared. Finally, the last example is point-free as there is no trace at all of the arguments.

Another language where the difference between functions and variables is more tenuous than in PHP is JavaScript. In fact, all functions are variables and since there is no special syntax for variables, there is no distinction between a traditional function and an anonymous one assigned to a variable:

```
// traditional
function snakeCase(word) {
    return word.toLowerCase().replace(/\s+/ig, '_');
};

// point-free
var snakeCase = compose(replace(/\s+/ig, '_'), toLowerCase);
```

Obviously, this is not valid JavaScript because there is no native `compose` function and both functions working on strings cannot be called like that so simply. There are, however, multiple libraries allowing you to write such code with ease, such as **Ramda**, which I highly recommend. The point of the example is just to demonstrate that you cannot distinguish a traditional function and an anonymous function in JavaScript, as is the case in PHP.

There are some benefits to using such a style:

- You usually have a more concise code, which some consider cleaner and easier to read.
- It helps thinking in terms of abstraction. The parameter name `word` in the JavaScript examples hints that the function works on single words only whereas it can work on any string. This is especially true for more generic functions such as those working on lists, for example.
- It helps developers think in terms of function composition instead of

data structures, which often results in better code.

There are, however, also some possible drawbacks:

- Removing explicit arguments from the definition might make things harder to understand; having no parameter name, for example, sometimes removes useful information.
- Long chains of composed functions can lead to losing sight of the data structure and types.
- Code can be more difficult to maintain. When you have an explicit function you can easily add new lines, debugging, and so on. This is made nearly impossible when functions are composed together.

Some opponents to the paradigm sometimes use the term *pointless style* to describe this technique as a result of those issues.

Reading and using point-free code definitely takes some time getting used to. I have personally no strong opinion about it. I advise you to use the style that suits you best and also, even if you prefer one, there are situations where the other is arguably better so don't hesitate to mix the two.

As a closing word, I would like to remind you that "Parameter order matters, a lot!" as we discussed in Chapter 4, *Composing functions*. And it is especially true if you want to use a point-free style. If the data you need to work on isn't the last parameter, you won't be able to.

Using const for functions

This technique is not related to functional programming, but rather it is a neat trick in PHP itself. It will, however, probably help you a lot, so here we go.

If you've had a look at the code of the `functional-php` library, you might have noticed that there are definitions of constants at the top of nearly all functions. Here is a small example:

```
<?php
const push = 'Widmogrod\Functional\push';

function push(array $array, array $values)
{
    // [...]
}
```

The idea behind this is to allow a simpler usage of functions as parameters. We've seen earlier that the way you pass a function or method as an argument is to use something called a `callable`, which usually is either a string or an array for methods composed of an object instance and a string for the method to call.

Using a `const` keyword allows us to have something much more akin to what you can find in languages where functions are not a separate construct from variables:

```
<?php const increment = 'increment';

function increment(int $i) { return $i + 1; }

// using a 'callable'
array_map('increment' [1, 2, 3, 4]);

// using our const
array_map(increment, [1, 2, 3, 4]);
```

Exit the awkward quotes around our function name. It really looks like you are passing the function itself, as would be the case in other

languages such as Python or JavaScript.

If you are using an IDE it gets even better. You can use the *Go to declaration* or equivalent function and the file where you defined the `const` will open on the line where it is defined. If you declared it either just on top or at the bottom of the *real* function, you will have quick access to its definition.

Some IDE, I am not aware of, might offer the same feature for `callable` in the form of strings, but it is at least not the case for the one that I am using. Whereas if I press *Ctrl* + click on the `increment` function in the second example, it focuses on the `const` declaration, which is a real time-saver.

When declaring constants like that, you are not limited to shadow functions; it also works with static object methods. You can also use **DocBlock** annotations to declare that your constant represents a `callable` type:

```
<?php
class A {
    public static function static_test() {}
    public function test() {}
}

/** @var callable */
const A_static = ['A', 'static_test'];
```

Sadly, this trick will not work for anonymous functions stored inside a variable or calling methods on object instances. If you try to do so, PHP will gratify you with a resounding `Warning: Constants may only evaluate to scalar values or arrays` warning.

Although not a silver bullet and accompanied by limitations, this little technique will help you write code that is a bit cleaner and easier to navigate in if you are using an IDE.

Recursion, stack overflows, and trampolines

We first presented recursion as a possible solution to programming problems in Chapter 3, *Functional basis in PHP*. Some memory complications were hinted at; it is time to investigate those further.

Tail-calls

A call to a function that is the last statement done before returning a value is called a **tail-call**. Let's have a look at some examples to get a grasp of what it means:

```
<?php
function simple() {
    return strtoupper('Hello!');
}
```

This is without any doubt a tail-call. The last statement of the function returns the result of the `strtoupper` function:

```
<?php

function multiple_branches($name) {
    if($name == 'Gilles') {
        return strtoupper('Hi friend!');
    }
    return strtoupper('Greetings');
}
```

Here, both calls to the `strtoupper` function are tail-calls. The position inside the function does not matter; what does matter is if there are any kinds of operations made after the call to the function is made. In our case, if the argument value is `Gilles` the last thing the function will do is call the `strtoupper` function, making it a tail-call:

```
<?php
function not_a_tail_call($name) {
    return strtoupper('Hello') + ' ' + $name;
```

```
}

function also_not_a_tail_call($a) {
    return 2 * max($a, 10);
}
```

None of those two functions have a tail-call. In both cases, the return value of the call is used to compute a final value before the function returns. The order of operation does not matter, the interpreter needs to first get the value of the `strtoupper` and the `max` functions in order to compute the result.

As we just saw, spotting a tail-call is not always easy. You can have one in the first few lines of a very long function, and being on the last line is not a sufficient criterion.

If the tail-call is made to the function itself, or in other words it is a recursive call, the term of **tail recursion** is often used.

Tail-call elimination

Why bother you are maybe asking yourself? Because compilers and parsers can perform something called tail-call elimination or sometimes **Tail-Call Optimization** (**TCO**) in short.

Instead of performing a new function call and suffering from all the related overhead, the program simply jumps to the next function without adding more information to the stack and wasting precious time passing parameters around.

This particularly matters in cases of tail recursion as it allows the stack to stay flat, not using any more memory than the first function call.

It sounds great, but as with most advanced compilation techniques, the PHP engine does not implement tail-call elimination. However, other languages do:

- Any **ECMAScript** 6 compliant JavaScript engine
- Python after installing the tco module

- Scala, you even have an annotation (`@tailrec`) to trigger a compiler error if your method is not tail recursive
- **Elixir**
- **Lua**
- **Perl**
- Haskell

There is also ongoing proposition and work to perform tail-call elimination at the **Java Virtual Machine** (**JVM**) level, but no concrete implementation so far has landed in Java 8, as this is not considered a priority feature.

Tail recursive functions are usually simpler to work with especially with regards to folding; as we saw in this section, techniques exist to alleviate the growing stack issue at the cost of some processing power.

From recursion to tail recursion

Now that we have a clearer understanding of what we are talking about, let's learn how we can transform a recursive function in to a tail recursive one if that is not already the case.

We declined to use the computation of a factorial as a good recursion example in Chapter 3, *Functional Basis in PHP*, but as it is probably the simplest recursive function that could be written, we will start with that example:

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

Is this function tail-recursive? No, we multiply a value with the result of our recursive call to `fact` method. Let's see the various steps in more detail:

```
fact(4)
4 * fact(3)
```

```
4 * 3 * fact(2)
4 * 3 * 2 * fact(1)
4 * 3 * 2 * 1
-> 24
```

Any idea how we can transform this to a tail-recursive function? Take some time to play around with some ideas before reading further. If you need a hint, think about how functions used for folding operate.

The usual answer when it comes to tail recursion revolves around using accumulators:

```
<?php
function fact2($n)
{
    $fact = function($n, $acc) use (&$fact) {
        return $n <= 1 ? $acc : $fact($n - 1, $n * $acc);
    };

    return $fact($n, 1);
}
```

Here we wrote it using an inner helper to hide the implementation detail of the accumulator, but we could just have written it using a unique function:

```
<?php

function fact3($n, $acc = 1)
{
    return $n <= 1 ? $acc : fact3($n - 1, $n * $acc);
}
```

Let's have a look at the steps once again:

```
fact(4)
fact(3, 4 * 1)
fact(2, 3 * 4)
fact(1, 2 * 12) -> 24
```

Great, no more pending operation after each recursive call; we truly have a tail recursive function. Our `fact` function was pretty simple. What about the *Tower of Hanoi* solver we wrote earlier? Here it is so

you don't have to search for it:

```
<?php
function hanoi(int $disc, string $source, string $destination,
string $via)
{
    if ($disc === 1) {
        echo("Move a disc from the $source rod to the
$destination  rod\n");
    } else {
        // step 1 : move all discs but the first to the "via"
rod
        hanoi($disc - 1, $source, $via, $destination);
        // step 2 : move the last disc to the destination
        hanoi(1, $source, $destination, $via);
        // step 3 : move the discs from the "via" rod to the
destination
        hanoi($disc - 1, $via, $destination, $source);
    }
}
```

Like for our factorial computation, take some time to try transforming the function into a tail recursive one yourself:

```
<?php
use Functional as f;

class Position
{
    public $disc;
    public $src;
    public $dst;
    public $via;

    public function __construct($n, $s, $d, $v)
    {
        $this->disc = $n;
        $this->src = $s;
        $this->dst = $d;
        $this->via = $v;
    }
}

function hanoi(Position $pos, array $moves = [])
{
    if ($pos->disc === 1) {
```

```
        echo("Move a disc from the {$pos->src} rod to the
{$pos- >dst} rod\n");

        if(count($moves) > 0) {
            hanoi(f\head($moves), f\tail($moves));
        }
    } else {
        $pos1 = new Position($pos->disc - 1, $pos->src, $pos-
>via,  $pos->dst);
        $pos2 = new Position(1, $pos->src, $pos->dst, $pos-
>via);
        $pos3 = new Position($pos->disc - 1, $pos->via, $pos-
>dst,  $pos->src);

        hanoi($pos1, array_merge([$pos2, $pos3], $moves));
    }
}
hanoi(new Position(3, 'left', 'right', 'middle'));
```

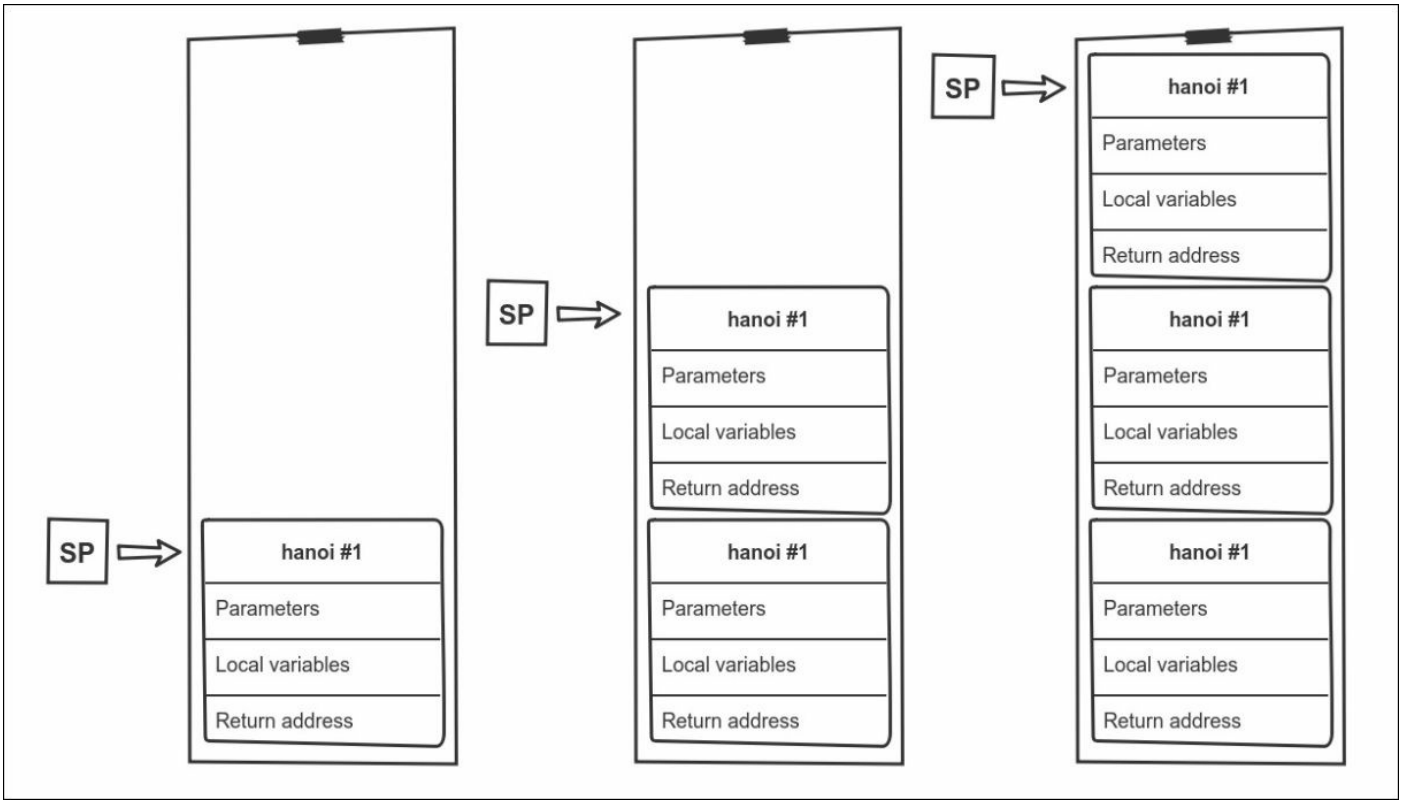
As you can see, the solution is pretty similar, even when there are multiple recursive calls inside the function, we can still use an accumulator. The trick is to use an array instead of storing only the current value. In most cases, the accumulator will be a **stack**, meaning you can only add elements at the beginning and remove them from the beginning. A stack is said to be a **Last In, First Out** (**LIFO**) structure.

If you don't quite get why this refactoring works, I encourage you to write down the steps for both variants as we did for the `fact` method so that you can get a better feel for the involved mechanics.

In fact, taking the time to write down the steps of a recursive algorithm is often a great way to clearly understand what is happening and how it can be refactored to be tail recursive or fix a bug if there is one.

Stack overflows

The fact we used a stack-like data structure for our tail recursive Hanoi solver is no coincidence. When you call functions, all needed information is also stored in a stack-like structure in memory. In the case of recursion, it will look something like this:



This stack has a limited size. It is bound through the `memory_limit` configuration option and even if you remove the limit you won't be able to go beyond the available memory in the system. Also, extensions such as **Xdebug** introduce specific mechanisms to avoid having too many nested recursive calls. For example, you have a configuration option named `xdebug.max_nesting_level`, which defaults to 256, meaning that if you call a function recursively more than that, an error will be raised.

If PHP performed tail-call optimization, instead of stacking the pieces of function information on top of one another, the call would replace the current information in the stack. It is safe to do so because the final result of a tail-call does not depend on any variable local to the function.

Since PHP does not perform this kind of optimization, we need to find another solution to avoid blowing up the stack. If you encounter this issue and you are willing to sacrifice some processing power to limit the memory usage, you can use **trampolines**.

Trampolines

The only way we can avoid stack growth is to return a value instead of calling a new function. This value can hold information needed to perform a new function call, which will continue the computation. This also means we need some cooperation from the caller of the function.

This helpful caller is the trampoline and here is how it works:

- The trampoline calls our function `f`
- Instead of making a recursive call, the `f` function returns the next call encapsulated inside a data structure with all the arguments
- The trampoline extracts the information and performs a new call to the `f` function
- Repeat the two last steps until the `f` function returns a *real* value
- The trampoline receives a value and it returns to the *real* caller

These steps should also explain where the name of the technique comes from, each time the function returns to the trampoline, it bounced back with the next arguments.

To perform these steps, we need a data structure holding both the function to call in the form of a `callable` and the arguments. We also need a helper function that will continue calling whatever is stored inside the data structure until it gets a real value:

```
<?php
class Bounce
{
    private $f;
    private $args;

    public function __construct(callable $f, ...$args)
    {
        $this->f = $f;
        $this->args = $args;
    }

    public function __invoke()
    {
        return call_user_func_array($this->f, $this->args);
    }
}
```

```
function trampoline(callable $f, ...$args) {
    $return = call_user_func_array($f, $args);

    while($return instanceof Bounce) {
        $return = $return();
    }
    return $return;
}
```

Simple enough, let's try it:

```
<?php

function fact4($n, $acc = 1)
{
    return $n <= 1 ? $acc : new Bounce('fact4', $n - 1, $n *
$acc);
}

echo trampoline('fact4', 5)
// 120
```

Works fine, and the code isn't that much harder to read. There is, however, a performance hit when using trampolines. In the case of computing a factorial, the trampoline version is roughly five times slower on my laptop. This is explained by the fact that the interpreter has to do to a lot more work than simply calling the next function.

Knowing this, if your recursive algorithm has a bound depth and you are certain that you are at no risk of a stack overflow, I recommend you to just perform a traditional recursion instead of using a trampoline. In the case of doubt, however, don't hesitate, as a stack overflow error can be critical on a production system.

Multi-step recursion

Trampolines even have their usefulness for languages performing incomplete tail-call elimination. For example, Scala is unable to perform such an optimization when there are two functions involved. Instead of trying to explain what I am talking about, let's see some code:

```
<?php

function even($n) {
    return $n == 0 ? 'yes' : odd($n - 1);
}

function odd($n) {
    return $n == 0 ? 'no' : even($n - 1);
}

echo even(10);
// yes

echo odd(10);
// no
```

This might not be the best and most efficient way to determine if a number is odd or even, but it has the merits to simply illustrate what I am talking about. Both functions are calling themselves until the number reaches 0, at which time we can decide if it is odd or even.

Depending on whom you ask, this might or might not be recursion. It respects the academic definition we gave in Chapter 3, *Functional basis in PHP*:

> *Recursion is the idea to divide a problem into smaller instance of the same problem.*

However, the function does not call itself, so this is why some people will try to define what is happening here with other terms. In the end, what we call that does not matter; we will have a stack overflow on a big number.

As I was saying, Scala performs an incomplete tail-call elimination as it will only do so if the function calls itself as its last instruction, leading to a stack overflow error as PHP will do. This is why trampolines are used even in some functional languages.

As a real simple exercise, I invite you to rewrite both `odd` and `even` functions using trampolines.

The trampoline library

If you want to use trampolines in your own project, I invite you to install the following library using the `composer` command, as it offers some helpers compared to our crude implementation:

```
composer require functional-php/trampoline
```

The data structure and features have been conflated inside the same class called `Trampoline`. Helpers are available in the form of functions:

- The `bounce` helper is used to create a new function wrapper. It takes a `callable` and the arguments.
- The `trampoline` helper runs a callable until completion. It takes a `callable` and its arguments. The method also accepts a `Trampoline` class instance as parameter, but in this case, the arguments will be ignored as they are already wrapped inside the instance.

The class also defines `__callStatic`, which allows us to call any function from the global namespace on the class directly.

Here are some examples taken from the documentation:

```
<?php

use FunctionalPHP\Trampoline as t;
use FunctionalPHP\Trampoline\Trampoline;

function factorial($n, $acc = 1) {
    return $n <= 1 ? $acc : t\bounce('factorial', $n - 1, $n *
$acc);
};

echo t\trampoline('factorial', 5);
// 120

echo Trampoline::factorial(5);
// 120

echo Trampoline::strtoupper('Hello!');
// HELLO!
```

Another helper returning a callable with all the trampoline capabilities turned on also exists, and it is called the `trampoline_wrapper` helper:

```
<?php
$fact = t\trampoline_wrapper('factorial');

echo $fact(5);
// 120
```

As an exercise, you can try to transform our *Tower of Hanoi* solver to use the `trampoline` library and see if you get the same results.

Alternative method

Instead of using trampolines to solve stack overflow issues, it is also possible to use a queue to store all the arguments of the successive recursive calls to our function.

The original function needs to be wrapped inside a helper that will hold the queue and call the original function with all the arguments in a chain. In order for this to work, the recursive call needs to be made to the wrapper instead:

- Create the wrapper function
- First call to the wrapper with the first arguments
- The wrapper enters a loop that calls the original function while there are arguments in the queue
- Each subsequent call to the wrapper adds arguments to the queue instead of calling the original function
- Once the loop is finished (that is, all recursive calls have been made), the wrapper returns the final value

For it to work, it is really important that the original function calls the wrapper when making a recursive call. This can be done either by using an anonymous function that you use inside of your wrapped function or using the `bindTo` method on the `Closure` class, as we discussed in Chapter 1, *Functions as First Class Citizen in PHP*.

The `trampoline` library has an implementation of this method using the

later technique. Here is how you can use it instead of a trampoline:

```
<?php

use FunctionalPHP\Trampoline as t;

$fact = T\pool(function($n, $acc = 1) {
    return $n <= 1 ? $acc : $this($n - 1, $n * $acc);
});

echo $fact(5);
// 120
```

The wrapper created by the `pool` function binds an instance of the `Pool` class to `$this`. This class has an `__invoke` method, which is callable inside our original function. Doing so will call the wrapper again, but this time it will add the arguments to the queue instead of calling the original function.

From a performance standpoint there is no difference between this method and the trampoline, both should perform roughly the same way. However, you cannot do multi-step recursion using a `pool` function as the wrapper is only aware of one function.

Also, until PHP 7.1 is released, this method is also limited to anonymous functions due to some difficulties transforming a callable in the form of a string to an instance of `Closure` class in order to bind the class to it. PHP 7.1 will introduce a new `fromCallable` method on `Closure`, which allows lifting this limitation.

Closing words

As a final note, both the trampoline and queue techniques we've seen to solve the stack overflow problem will only work if the recursive function is tail recursive. This is a mandatory condition as the function needs to fully return in order for the helper to continue the computation.

Also, as the trampoline method has fewer drawbacks, I would recommend this instead of the `pool` function implementation.

Pattern matching

Pattern matching is a really powerful feature of most functional languages. It is embedded inside the language at various levels. In Scala, for example, you can use it as a beefed up `switch` statement, and in Haskell it is an integral part of function definition.

Pattern matching is the process of checking a series of tokens against a pattern. It is different from pattern recognition as the match needs to be exact. The process does not only match as a switch statement does, it also assigns the value as with the `list` construct in PHP, a process called **destructuring assignment**.

It is not to be confused with regular expressions. Regular expressions can only operate on the content of strings, where as pattern matching can also operate on the structure of the data. For example, you can match on the number of elements of an array.

Let's see some examples in Haskell so we get a feel for it. The simplest form of pattern matching is to match to a specific value:

```
fact :: (Integral a) => a -> a
fact 0 = 1
fact n = n * fact (n-1)
```

This is how you could define `fact` function in Haskell:

- The first line is the type signature, which should remind you something we saw earlier in this chapter; `Integral` is a type that is a bit less restrictive than the `Integer` type without entering in the details.
- The second line is the function body if the argument has the value 0.
- The last line is executed in all other cases. The value is assigned to the `n` variable.

If you are not interested in some values, you can ignore them using the _ (underscore) wildcard pattern. For example, you can easily define functions to get the first, second, and third value from a tuple:

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Note

A tuple is a data structure with a fixed number of elements, as opposed to an array, which can change size. The `(1, 2)` and `('a', 'b')` tuples are both of size two. The advantage of using tuples over arrays when there is a known number of elements lies in enforcing the correct size and performance.

Languages such as Haskell, Scala, Python, C#, and Ruby have a tuple type inside their core or standard library.

You can destruct the values into more than just one variable. In order to understand the following examples, you need to know that ":" (colon) is the operator to prepend an element to a list. This means that the `1:[2, 3]` tuple will return the list `[1, 2, 3]`:

```
head :: [a] -> a
head [] = error "empty list"
head (x:_) = x

tail :: [a] -> [a]
tail [] = error "empty list"
tail (_:xs) = xs

sum :: (Num a) => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

The `head` and `tail` variable have the same structure, if the list is empty, they return an error. Otherwise either return `x`, which is the element at the beginning of the list, or `xs`, which is the rest of the list. The `sum` variable is also similar, but instead it uses both `x` and `xs`. By the way, Haskell would disallow defining those two functions as they already

exist:

```
firstThree :: [a] -> (a, a, a)
firstThree (x:y:z:_) = (x, y, z)
firstThree _ = error "need at least 3 elements"
```

The `firstThree` variable is a bit different. It first tries to match a list with at least three elements, `x`, `y`, and `z`. In this case the _ pattern can be the empty list or not, the pattern will match. If it doesn't succeed, we know that the list has fewer than three elements and we display an error.

You can also use pattern matching as a beefed up switch statement. For example, this would also be a valid implementation of `head`:

```
head :: [a] -> a
head xs = case xs of []    -> error "empty lists"
(x:_) -> x
```

If you want to use both the destructuring data and the whole value, you can use **as patterns**:

```
firstLetter :: String -> String
firstLetter "" = error "empty string"
firstLetter all@(x:_) = "The first letter of " ++ all ++ " is "
++ [x]
```

Finally, you can also use constructors when doing pattern matching. Here is a small example using the `Maybe` type:

```
increment :: Maybe Int -> Int
increment Nothing = 0
increment (Just x) = x + 1
```

Yes, you can get the value inside a monad just like that, using destructuring.

You can have overlapping patterns; Haskell will use the first one that matches. And if it isn't able to find a matching one, it will raise an error saying `Non-exhaustive patterns in function XXX`.

We could demonstrate roughly the same kind of features for Scala,

Clojure, or other functional languages, but as this was just an example to understand what pattern matching is about, I would rather advise you to read tutorials on the subject if the topic is of interest to you. Instead, we will try to emulate part of this powerful feature in PHP.

Pattern matching in PHP

Obviously, we will never be able to declare functions the same way we just saw in Haskell as this needs to be implemented at the heart of the language. However, a library tries to emulate pattern matching as best as possible to create a more powerful version of the switch statement with automatic destructuring.

You can install the library using `composer` command in Composer:

```
composer require functional-php/pattern-matching
```

In order to be as expressive as possible to what is available in Haskell, it uses a string to hold the patterns. Here is a table defining the various possible syntaxes:

Name	Format	Example
Constant	Any scalar value (integer, float, string, Boolean)	`1.0`, `42`, "test"
Variable	identifier	`a`, `name`, `anything`
Array	[<pattern>, ..., <pattern>]	`[]`, `[a]`, `[a, b, c]`
Cons	(identifier:list-identifier)	`(x:xs)`, `(x:y:z:xs)`
Wildcard	_	_
As	identifier@(<pattern>)	all@(x:xs)

At the time of writing, there is no support for automatic destructuring of values inside a Monad, or other types, nor the possibility to constrain the type of a particular item we match upon. There are, however, opened

issues concerning those two features https://github.com/functional-php/pattern-matching/issues.

As there is no possibility to use named parameters in PHP, the parameters will be passed in the order they are defined in the pattern and there will be no matching done based on their names. This makes using the library a bit tedious at times.

Better switch statements

The library can be used to perform more advanced `switch` statements by also using the structure and extracting data instead of just equating on the value. Since the functions are curryied, you can also map them over arrays contrary to a `switch` statement:

```
<?php

use FunctionalPHP\PatternMatching as m;

$users = [
    [ 'name' => 'Gilles', 'status' => 10 ],
    [ 'name' => 'John', 'status' => 5 ],
    [ 'name' => 'Ben', 'status' => 0],
    [],
    'some random string'
];

$statuses = array_map(m\match([
    '[_, 10]' => function() { return 'admin'; },
    '[_, 5]' => 'moderator',
    '[_, _]' => 'normal user',
    '_' => 'n/a',
]), $users);

print_r($statuses);
// Array (
//     [0] => Gilles - admin
//     [1] => John - moderator
//     [2] => Ben - normal user
//     [3] => incomplete array
//     [4] => n/a
// )
```

The first pattern in the list that matches is used. As you can see, the callback can either be a function as for the first pattern or a constant that will be returned. Obviously in this case all of them could have been constants, but this was for the sake of the example.

One of the benefits over a traditional `switch` statement as you can see is that you are not constrained by the structure of the data to perform your match. In our case, we created a catch-all pattern at the end for erroneous data. Using a `switch` statement you would have needed to either filter out the data or perform some kind of other data normalization.

This example could also use the destructuring to avoid having three patterns with constants (and while we are at it, we will also use the name from the array):

```
<?php

$group_names = [ 10 => 'admin', 5 => 'moderator' ];

$statuses = array_map(m\match([
    '[name, s]' => function($name, $s) use($group_names) {
        return $name.
            ' - '.
            (isset($group_names[$s]) ? $group_names[$s] :
'normal user');
    },
    '[]' => 'incomplete array',
    '_' => 'n/a',]), $users);
print_r($statuses);
// Array (
//     [0] => admin
//     [1] => moderator
//     [2] => normal user
//     [3] => incomplete array
//     [4] => n/a
// )
```

You could also write patterns that match a wide variety of different structures and use that to determine what to do based on it. It could also be used to perform some kind of basic routing inside a web application:

```
$url = 'user/10';

function homepage() { return "Hello!"; }
function user($id) { return "user $id"; }
function add_user_to_group($group, $user) { return "done."; }

$result = m\match([
    '["user", id]' => 'user',
    '["group", group, "add", user]' => 'add_user_to_group',
    '_' => 'homepage',
], explode('/', $url));

echo $result;
// user 10
```

Obviously a more specialized library would do a better job at routing with a better performances, but keeping the possibility in mind can come in handy and it demonstrates the versatility of pattern matching.

Other usages

If you are just interested in destructuring your data, the `extract` function has you covered:

```
<?php
$data = [
  'Gilles',
  ['Some street', '12345', 'Some City'],
  'xxx xx-xx-xx',
  ['admin', 'staff'],
  ['username' => 'gilles', 'password' => '******'],
  [12, 34, 53, 65, 78, 95, 102]
];

print_r(m\extract('[name, _, phone, groups, [username, _],
posts@(first:_)]', $data));
// Array (
//     [name] => Gilles
//     [phone] => xxx xx-xx-xx
//     [groups] => Array ( [0] => admin [1] => staff )
//     [username] => gilles
//     [posts] => Array ( ... )
//     [first] => 12
//)
```

Once you've extracted your data, you can use the `extract` function from PHP to import the variables into the current scope.

If you want to create functions a bit like we saw in the Haskell examples, there is the `func` helper method. Obviously the syntax is not as good, but it can come in handy:

```
<?php
$fact = m\func([
    '0' => 1,
    'n' => function($n) use(&$fact) {
        return $n * $fact($n - 1);
    }
]);
```

Beware that function creation is still in the beta stage at the time of writing. There are some issues, so the API might change in the future. If you encounter any issues, consult the documentation.

Type classes

Another idea you will encounter often when reading papers, posts, or tutorials about functional programming is the type class, especially if the content is about Haskell.

The concept of a type class was first introduced in Haskell as a way of implementing operators that could be easily overloaded for various types. Since then a lot of other uses have been discovered for them. For example, in Haskell, functors, applicatives, and monads are all type classes.

Haskell needed type classes mostly because it is not an object-oriented language. The overloading of operators was, for example, solved differently in Scala. You can write equivalents to type classes in Scala, but it is more of a pattern than a language feature. In other languages, there are various ways to emulate some features of Haskell type classes using *traits* and *interfaces*.

In Haskell, a type class is a set of functions that need to be implemented on a given type. One of the simplest examples is the `Eq` type class:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Any type implementing, the dedicated term is **derives**, the `Eq` class must also have corresponding implementation for the `==` and `/=` operators, otherwise you will have a compilation error. This is exactly like an interface for a class, but for a type instead. This means you can enforce the creation of operators, like in our case, not just methods.

You can create instances for your type class pretty easily; here is one for the `Maybe` instance:

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
```

```
    _ == _ = False
```

It should be fairly easy to understand. The two `Just` values on the left are equal if their inner content is, the `Nothing` value is equal to itself, anything else is different. Defining these `Eq` instances allows you to check for the equality of two monads anywhere in your Haskell code. The instance definition in itself just enforces that the type `m` variable stored inside the monad also implements the `Eq` type class.

As you can see, the functions from the type class are not implemented inside the type. It is done as a separate instance. This allows you to declare this instance anywhere in your code.

You can even have multiple instances for the same type class for any given type and import the one you need. Imagine, for example, having two monoid instances for integers, one for the product and the other for the sum. This is, however, discouraged as it causes conflicts when both instances are imported:

```
Prelude> Just 2 == Just 2
True
Prelude> Just 2 == Just 3
False
Prelude> Just 2 == Nothing
False
```

Note

`Prelude>` is the prompt in the Haskell REPL on which you can simply run Haskell code like you do when running PHP on the CLI with the `-a` parameter.

We might think that type classes are just interfaces, or rather traits as they can also contain implementation. But if we take a closer look at both our type class and its implementation for monads, there are at least three shortcomings that anything we could do in PHP, at least while being reasonable, will have.

To demonstrate this, let's imagine we created a PHP interface called

`Comparable`:

```
<?php
interface Comparable
{
    /**
     * @param Comparable $a the object to compare with
     * @return int
     *    0 if both object are equal
     *    1 is $a is smaller
     *    -1 otherwise
     */
    public function compare(Comparable $a): int;
}
```

Setting aside the fact that PHP does not allow for operator overloading like we demonstrated with the `==` symbol in Haskell, try thinking of the three features that a Haskell type class allows that are near to impossible to emulate in PHP.

Two of them are about enforcing the correct type. Haskell will do checks for us automatically, but in PHP we will have to write code to check if our values are of the right type. The third issue is related to extensibility. Think, for example, of classes declared in an external library that you want to compare.

The first issue is related to the fact that the `compare` function expects a value of the same type as the interface, meaning if you have two unrelated classes, `A` and `B`, both implementing the `Comparable` interface, you could compare an instance of the class `A` with an instance of the `B` class without any error from PHP. Obviously this is wrong and this forces you to first check if both values are of the same type before comparing them.

Things get even trickier when you have a class hierarchy as you don't really know what type to test for. Haskell, however, will automatically get the first common type both values share and use the related comparator.

The second issue is a bit more complicated to find out. If we implement

the `Comparable` interface on any kind of container, we will need to check when the comparison is run that the contained values are also comparable. In Haskell, the type signature `(Eq m) => Eq (Maybe m)` already takes care of that for us and an error will be raised automatically if you try to compare two monads that hold non-comparable values.

The Haskell type system also enforces that the values inside the monad are of the same type, which is an issue related to the first problem we spotted previously.

Finally, the third and probably biggest issue is about the possibility to implement the `Comparable` interface on a class from an external library or the core of PHP. Since Haskell type class instances are outside of the class itself, you can add one for anything at anytime, be it for an existing type class or a new one you just created.

You can create adapters or wrapping classes around those objects, but then you will have to perform some kind of boxing/unboxing dance to pass the right type to the various methods using your objects, which is far from a pleasant experience.

Scala being an object-oriented language without core support for type classes, the extensibility issue is solved cleverly by using a feature of the language called **implicit conversion**. Broadly, the idea is that you define a conversion from type `A` to type `B` and when the compiler finds a method expecting an instance of the `B` class, but you passed an instance of the `A` class, it will look for this conversion and apply it if available.

This way, you can create the adapters or wrappers we proposed using before, but instead of having to perform the transformation manually, the Scala compilers take care of it for you, making the process completely transparent.

As a closing note, since we spoke about comparing objects and redefining operators, there are currently two RFCs proposed for PHP about these two topics:

- https://wiki.php.net/rfc/operator-overloading

- https://wiki.php.net/rfc/comparable

There is, however, no RFC in sight concerning type classes or implicit type casting like we saw exists in Scala.

Algebraic structures and category theory

Until now we have avoided talking too much about mathematics. This section will try to do so in a light way as most functional concepts have roots in mathematics and a lot of abstractions we already discussed are ideas taken from the field of category theory. The content in this section will probably help you to gain a better understanding of the content of this book, but there is no need to understand them perfectly to use functional programming.

Throughout this section, mathematical terms will be defined when encountered. It is not that important that you get all the nuances, it is just to give you a broad idea of what we are talking about.

The root of functional programming is the *lambda calculus*, or *λ-calculus*. It is a *Turing complete formal system* in mathematical logic to express computation. The term lambda to refer to closure and anonymous functions comes from this. Any code you write can be transformed to lambda calculus.

Note

A language or system is said to be Turing-complete when it can be used to simulate a Turing machine. A synonym is computationally universal. All major programming languages are Turing-complete, meaning that anything you can write in C, Java, PHP, Haskell, JavaScript, and so on, can be also written in any of the others. A formal system is composed of the following elements:

- A finite set of symbols or keywords
- A grammar defining a valid syntax
- A set of axioms, or founding rules
- A set of inference rules to derive other rules from the axioms

An algebraic structure is a set with one or more operations defined on it and a list of axioms, or laws. All abstractions we studied in the previous chapters are algebraic structures: monoids, functors, applicatives, and monads. This is important, because, when a new problem is shown to follow the same rules as an existing set, anything that was previously theorized can be reused.

Note

A set is a fundamental concept in mathematics. It is a collection of distinct objects. You can decide to group anything together and call it a set, for example, the numbers 5, 23, and 42 can form a set {5, 23, 42}. Sets can be defined explicitly as we just did, or using a rule, for example, all positive integers.

Category theory is the field that studies the relationship between two or more mathematical structures. It formalizes them in terms of a collection of objects and arrows, or **morphisms**. A morphism transforms one object or one category to another. A category has two properties:

- The ability to **compose morphisms** associatively
- An **identity morphism** going from one object or category to itself

Sets are one of the multiple possible categories that respect these two properties. The idea of category, objects, and morphisms can be highly abstract. For example, you can consider types, functions, functors, or monoids as categories. They can be finite or infinite and hold a variety objects as long as the two properties are respected.

If you are interested in this subject and want to learn more about the mathematical aspect of it, I can recommend the book *Cakes, Custard and Category Theory: Easy Recipes for Understanding Complex Maths* written by Eugenia Cheng. It is really accessible, no prior mathematical knowledge is required, and actually fun to read.

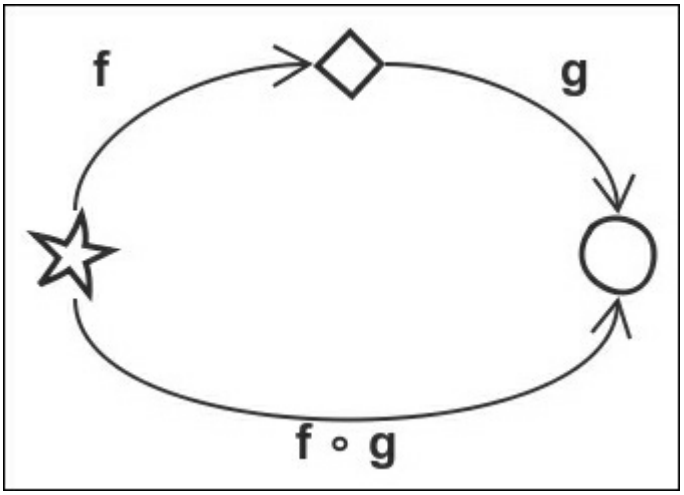
The whole idea of pure functions and referential transparency comes from the lambda calculus. Type systems, and especially Hindley-Milner, are deeply ingrained in formal logic and category theory. The concept of

morphism is so close to the one of function and the fact that composition is at the center of category theory results in the fact that most advances made in the last couple of years on functional languages being in one way or another linked to this mathematical field.

From mathematics to computer science

As we just saw, category theory is an important cornerstone in the functional world. You don't have to know about it to code functionally, but it definitely helps you understand the underlying concepts and reason about the most abstract stuff. Also, a lot of the terms used in papers about functional programming or even libraries are taken directly from category theory. So, if you develop a good instinct about it, it will usually be a lot easier to understand what you are reading.

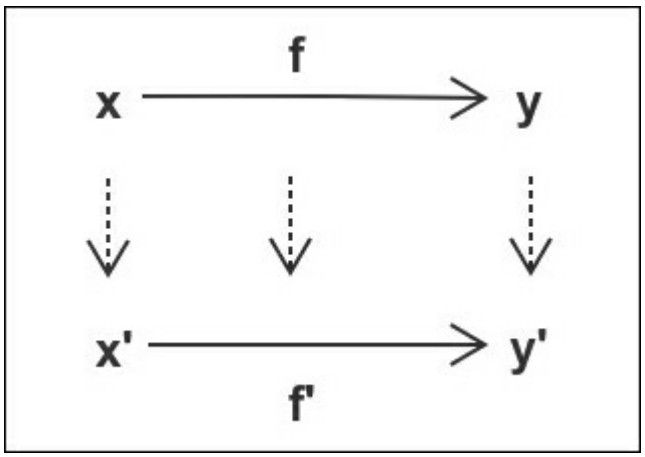
A category is in fact a pretty simple concept. It is only some objects with arrows that go between them. If you are able to represent something graphically using those ideas, it will most probably be correct and it will greatly help you to understand what is happening. For example, if we decide to represent function composition we will end up with something like this:



In the diagram, it doesn't really matter if our three shapes are whole categories or objects in a given category. What is, however, immediately clear is that if we have two morphisms, `f` and `g`, the result is the same if

we apply them sequentially or if we apply the composed version. The function can work on the same types, for example, we transform from one shape to another, or the triangle represents the strings, the diamond represents chars, and the circles represent integers. It makes no difference.

Another example would be the homomorphism law for applicative; `pure(f)->apply($x) == pure(f($x))`. If we consider the `pure` function to be a morphism from one category to the category that represents the possible objects for the applicative, we can visualize the law as follows:



The dotted arrow is the `pure` function, which is used to move `x`, `f`, and `y` over the applicative category. It becomes fairly self-evident that the law is sound when we see it like that. What do you think? By the way, both those diagrams are called **commutative diagrams**. In a commutative diagram, each path with the same start and endpoints are equivalent when using composition between the arrows.

Also, you could consider that each type is a category and that a morphism is a function. You can imagine functions as morphisms going from an object in a category to an object in the same category or a different one. You can represent the type and class hierarchy by smaller categories inside bigger ones. For example, integers and floats would be two categories inside the numeric category and they would overlap on some numbers.

This might not be the most academically sound way to describe both types and functions and category theory, but it's an easy one to grasp. It makes it easier to conceptualize abstract concepts such as functors or monads in parallel with what we are accustomed to.

You can visualize more traditional functions operating on values themselves. For example, the `strtoupper` function is a morphism from an object in the `string` category to another object in the same category, whereas the `count` method is a morphism from an object in the `array` category to an object in the `integer` category. So these are arrows from an object to another object.

If we step back from our basic types as we did in the second diagram, we can imagine functions working on types themselves. For example, the `pure` function of a monad takes a certain category, be it a type or a function, and lifts it to a new category in which all objects are now wrapped inside the context of the monad.

The idea is interesting, because any arrows that you had before can also be lifted and will continue to have the same results inside their new context, as proved by the *homomorphism* law we just visualized.

This means that, if you have trouble understanding what is happening when you use a monad, or any abstraction really, just draw the operations on a sheet of paper using categories, objects, and arrows and you can then boil everything down to its essence either by removing or adding context.

Important mathematical terms

There are some mathematical terms that you might encounter when reading about functional programming. They are used because they allow us to quickly convey what the topic is about and what properties can be expected from the structure currently described.

One such term we already learned about is the monoid. One mathematical definition you can find about it is *A monoid is a set that is closed under an associative binary operation and possess an identity*

element. At this point you should be able to understand this definition; however, here is a quick rundown for the string concatenation monoid:

- The set is all possible string values
- The binary operation is the string concatenation operator, `.`
- The identity element is the empty string, `''`

The idea of a set being closed under an operation indicates that the given operation result is also part of the set. For example, performing an addition of two integers will always result in an integer.

Here is a quick glossary of various mathematical terms that you might encounter in your readings.

- **Associative**: An operation is associative if the order of operation does not matter; examples are addition, and product is *a + (b + c) === (a + b) + c*.
- **Commutative**: An operation is commutative if you can change the order of the operands. Most associative operations are also commutative as *a + b === b + a* `. An example of an associative operation that isn't commutative is function composition as *f (g h) === (f g) h* but *f g != g f*.
- **Distributive**: Two operations are distributive if *a * (b + c) == (a * b) + (a * c)*. In this case, multiplication is "distributive over" addition. In the example, * and + can be replaced by any binary operation.
- **Semigroup**: A set closed under an associative operation.
- **Monoid**: A semigroup with an identity element.
- **Group**: A monoid with inverse elements. The inverse is a value you can add to another element to get the identity. For example, *10 + -10 = 0, -10* is the inverse of 10 in the integer addition group.
- **Abelian group**: A group where the operation is commutative.
- **Ring**: An abelian group with a second monoidal operation, which is distributive over the first one. For example, integers are a ring where addition is the first operation and multiplication is the second.
- **Semiring**: A ring where the abelian group is replaced by a commutative monoid (that is, the inverse element does not exist).

- **Comonad**, **cofunctor**, **coXXX**: The dual category to monad, functor, or anything. Another word for dual would be opposite. If a monad is a way to put something in a context, its comonad would be the way to get it out from it. This is a really vague definition without explaining the use, which would require a chapter in itself.

Fantasy Land

Now that we've got the theory out of the way, I want to present you the JavaScript project that describes interfaces for common algebraic structures named **Fantasy Land** at https://github.com/fantasyland/fantasy-land.

It has been widely adopted by the community and more and more projects each day implement the proposed interface for a better interoperability between various implementations of those algebraic structures. Under the Fantasy Land namespace itself you can find implementations of the various monads we already discovered before and many other more advanced functional constructs. Notably, there is also the **Bilby** library (http://bilby.brianmckenna.org/), which tries to be as close as possible to the Haskell philosophy.

Why am I talking about a JavaScript library? Because the `php-functional` library has kind of ported the Fantasy Land specification to PHP https://github.com/widmogrod/php-functional/tree/master/src/FantasyLand.

I would like nothing better than other projects using those as a base for their own implementation of functional code as it would bolster functional programming in PHP by providing developers with a bigger set of possible features they can use.

At the time of writing, there are discussions to separate the Fantasy Land port from the rest of the library so that it can be used without also depending on everything else. I hope by the time you read this, undertaking has been completed and I urge you to use this set of common interfaces.

Monad transformers

We saw that monads are already a pretty powerful idea if you take each of them separately. What if I told that you can compose monads together to benefit from the features of multiples of them at the same time? For example, a `Maybe` interface and a `Writer` monad that could inform you why the operation returned no result.

This is exactly what monad transformers are about. A monad transformer is in every respect similar to the monad that it exemplifies, except that it is not a standalone entity, instead it modifies the behavior of another monad. In a way, we can imagine this as adding a new monadic layer in top of some other monad. Of course, you can stack multiple layers on top of one another.

In Haskell, most existing monads have a corresponding transformer. Usually, they have the same name with a `T` suffixed: `StateT`, `ReaderT`, `WriterT`. If you were to apply a transformer to the identity monad, the result will have the exact same feature as the equivalent monad as the identity monad is just a simple container.

In order for this to work correctly, the `State`, `Reader`, `Writer`, and other Haskell monads are in fact type classes with two instances; one is the transformer and the other is the traditional monad.

We will end our foray here as there are no implementations of this concept I am aware of in PHP; it would be sufficiently difficult to try doing it ourselves and it would require at least one full chapter.

At least you've heard about the idea and who knows, maybe someone will create a library adding monad transformers to PHP in the future.

Lenses

Modifying data when everything is immutable can become cumbersome pretty quickly if you have some complicated data structure. Say, for example, you have a list of users, each user has a list of post associated to them and you need to change something on one of the posts. You will need to copy or recreate everything just to modify your value, since you cannot change anything in place.

Like I said, cumbersome. But as with mostly everything, there is a nice and clean solution in the form of lenses. Imagine your lens as if it was a part of a binocular; it lets you focus on one part of your data structure easily. You then have the tools to modify this as easily and get a whole new shiny data structure with your data changed to whatever you want it to be.

A lens is a lookup function that lets you get a particular field at the current depth of your data structure. You can read from a lens, it will return the pointed value. And you can write to a lens, it will return you the whole data structure with your modification.

What is really great about lenses is that since they are functions you can compose them, so if you want to go deeper than the first level, compose your second lookup function with the first one. And you can add a third, fourth, and so on lookups on top of that. At each step you could get or set the value if you want.

And since they are functions, you can use them as any other function, you can map them, put them in applicatives, bind them to a monad. Suddenly, an operation that was very cumbersome can leverage all the power of your language.

Sadly, as immutable data structures are not really common in PHP, no one took the time to write a lens library for it. Also, the details of how this is possible are a bit messy and would need quite some time to explain. This is why we'll leave it here for now.

If you are interested, the Haskell `lens` library has a webpage with lots of information and a great introduction, although a truly challenging video, at http://lens.github.io/.

Summary

In this chapter, we covered a lot of different topics. Some were concepts you will be able to use when developing in PHP such as avoiding stack overflows when doing recursion, pattern matching, the point-free style, and the usage of the `const` keyword to make your code a bit easier to read.

Others were purely theoretical with no current usage in PHP, ideas that were only presented so that you will be able to understand a bit better other writings about functional programming, such as monad transformers, the link between functional programming and category theory, and lenses.

Finally, some topics were somewhat in the middle, useful in your day-to-day coding, but a bit harder to put in to practice as the support in PHP is lacking. Even if it is not perfect, you are now aware of type classes, type signatures, and algebraic structures.

I have faith that you were not too much put off by the never-ending change of topics in this chapter and that you learned some valuable and interesting stuff. I also hope that this content sparked some interest in you to learn more about those topics and maybe try a functional language to see what all that hinted goodness was about.

In the next chapter, we'll get back to a more practical topic as we will discuss testing functional code first, and second we will learn about a methodology called property-based testing, which is not strictly speaking about functional programming, but was first theorized in Haskell.

Chapter 8. Testing

We already asserted multiple times throughout the book that pure functions are easier to test; it is time we prove it. In this chapter, we will first present a small glossary about the topic in order to ensure we speak a common language. We will then continue with how a functional approach helps with traditional testing. Finally, we will learn about a different way to test code, called **property-based testing**.

None of the subjects of this chapter are strictly confined to functional programming; you will be able to use anything in any legacy codebase. Also, this is not a book about testing, so we will not go into every detail. It is also assumed that you have some prior knowledge about testing code in PHP.

In this chapter, we will cover the following topics:

- Small testing glossary
- Testing pure functions
- Test parallelization as a speed-up technique
- Property-based testing

Testing vocabulary

I won't claim to give you a complete glossary of all testing-related terms and also I won't explain the subtle differences and interpretations that could be made for each of them. The idea of this section is simply to lay some common ground.

The glossary won't be in alphabetical order, but rather terms will be grouped by categories. Also, it must by no means be considered a complete glossary. There are a lot more terms and techniques that pertain to testing than what will be presented here, especially if you include all testing methods related to performance, security, and usability:

- **Unit testing**: Tests conducted against each individual component

separately. What is considered a *unit* varies-a function/method, a whole class, a whole module. Usually, dependency to other units is mocked to cleanly isolate each part.

- **Functional testing**: Tests the software as a black box to ensure that it meets the specifications. External dependency is usually mocked.
- **Integration testing**: Tests conducted against the whole application and its dependencies, including external ones, to ensure that everything integrates correctly.
- **Acceptance testing**: Tests conducted by the final customer / end user against a set of agreed-upon criteria.
- **Regression testing**: Repeats a test after some change is made to ensure no issues were introduced in the process.
- **Fuzz testing / Fuzzing**: Tests conducted by inputting massive amounts of (semi) random data in order to make it crash. This helps discover coding errors or security issues.
- **Ad-hoc testing**: Tests performed with no formal framework or plan.
- **Component testing**: See *unit testing*.
- **Blackbox testing**: See *functional testing*.
- **Behavioral testing**: See *functional testing*.
- **User Acceptance testing (UAT)**: See *acceptance testing*.
- **Alpha version**: Usually, the first version that is tested as a black box. It can be unstable and cause data loss.
- **Beta version**: Usually, the first version that is feature-complete and in a state good enough to be released to external people. It can still have serious issues and should not be used in a production environment.
- **Release Candidate (RC)**: A version that is deemed stable enough to be released to the public for a final test. Usually the last RC is "promoted" as the released version.
- **Mocking (mock)**: Creating components that imitate other parts of the software or an external service to test only the matter at hand.
- **Stubbing (stub)**: See *mocking*.
- **Code coverage**: The percentage of the application code or features that is covered by the tests. It can have different granularity: by lines, by functions, by components, and so on.
- **Instrumentation**: The process of adding code to the application in

order to test and monitor behavior or coverage. It can be done manually or by a tool either in the source, in a compiled form, or in-memory.

- **Peer review**: A process where one or multiple colleagues examine the produced work such as code, documentation, or anything pertaining to the release.
- **Static analysis**: Analysis the application without running it, usually done by a tool. It can provide information about coverage, complexity, coding style, or even found issues.
- **Static testing**: All of the testing and reviews performed without executing the application. See *peer review* and *static analysis*.
- **Smoke tests**: Superficially testing the main parts of an application to ensure the core features work.
- **Technical review**: See *peer review*.
- **Decision point**: A statement in the code where a change in the control flow can happen, typically an `if` condition.
- **Path**: The sequence of statements executed from the beginning of the function to the end. A function can have multiple paths depending on its decision point.
- **Cyclomatic complexity**: A measure of the complexity of a piece of code. There are various algorithms to compute it; one is "number of decision points + 1".
- **Defect**, **failure**, **issue,** or **bug**: Anything that does not work as expected in the application.
- **False-positive**: A test result seen as a defect when in fact everything works fine.
- **False-negative**: A test result seen as a success when in fact there is a defect.
- **Test-driven Development (TDD)**: A development methodology where you start by writing a test and then the minimum amount of code to make it pass before repeating the process.
- **Behavior-driven Development (BDD)**: A development methodology based on TDD where you describe behavior using a domain-specific language instead of writing traditional tests.
- **Type-driven Development**: A running joke in the functional world where you replace tests with a strong type system. Depending on

whom you ask, the idea might be taken more or less seriously.

- **X-driven Development**: There is a new best development methodology created every week; the website http://devdriven.by/ tries to reference them all.

Testing pure functions

As we just saw in the glossary, there are a lot of potential ways to test an application. In this section, we will, however, limit ourselves to tests at the function level; or in other words, we will do unit testing.

So, what makes pure functions so much easier to test? There are multiple reasons; let's start by enumerating them and we will then see why with real test cases:

- Mocking is simplified as you only need to provide input arguments. No external state to create, no singletons to stub.
- Repeated calls will yield exactly the same result for a given arguments list, whatever the time of day or previously run tests. There is no need to put the application in a certain state.
- Functional programming encourages smaller functions doing exactly one thing. This usually entails test cases that are easier to write and understand.
- Referential transparency usually means you need fewer tests to gain the same level of trust in your code.
- The absence of side-effects guarantees that your test will have no consequences on any other subsequent tests. This means you can run them in any order you want without worrying about resetting the state between each test or running them in isolation.

Some of these claims may seem a bit bold to you, or maybe you are unsure why I made them. Let's take some time to verify why they are true with examples. We will separate our examples into four different parts to makes things easier to follow.

All inputs are explicit

As we discovered earlier, a pure function needs to have all of its inputs as arguments. You cannot rely on some static method from a singleton, generate random numbers, or get any kind of data that can change from an external source.

The corollary is that you can run your test at any time during the day, on any environment, and for any given list of arguments, and the output will stay the same. This simple fact makes both writing and reading tests a lot easier.

Imagine you have to test the following function:

```
<?php

function greet()
{
  $hour = (int) date('g');

  if ($hour >= 5 && $hour < 12) {
    return "Good morning!";
  } elseif ($hour < 18) {
    return "Good afternoon!";
  } elseif ($hour < 22) {
    return "Good evening!";
  }
  return "Good night!";
}
```

The problem is that, when you call the function, you need to know what time it is so you can check whether the return value is correct. This fact leads to some issues:

- You basically have to re-implement the function logic inside the test, thus possibly having the same bug in both the test and the function.
- There is a slight chance that, between the time you computed the expected value and the function gets the time again to return a result, a minute elapsed, changing the current hour and thus the function result. Those kinds of false positive are a real headache to debug.
- You cannot test all possible outputs without somehow manipulating the system clock.
- The dependency to the current time being hidden, the person reading the test can only infer what the function is doing.

By simply moving the `$hour` variable as a parameter, we solve all the previously mentioned issues.

Also, if you use a test runner that allows you to create a data provider for your tests, such as **PHPUnit** or **atoum**, testing the function becomes as simple as creating a provider that creates a list of hours associated with the expected return and simply feeds the time to the function and checks the result. This test is a lot simpler to write, understand, and expand than anything else you would have needed to write earlier.

Referential transparency and no side-effects

Referential transparency ensures that you can replace a function call (with certain arguments) with the result of the computation anywhere in your code. This is also an interesting property for testing as it mostly means you will need to test less to gain the same amount of trust. Let me explain.

Usually, when you do unit testing, you try to choose the smallest unit possible that satisfies the trust you want to place in your code. Usually, you will test either at the module, class, or method level. Obviously, when doing functional programming, you will test at the function level.

Your functions will obviously call other functions. In a traditional testing setup, you would try to mock as many as those as possible in order to ensure that you test only the functionality of the current unit and you are not impacted by possible bugs in other functions.

Although not impossible, it's cumbersome to mock functions in PHP, so this becomes a bit difficult in our case. This is especially true for composed functions such as `$title = compose('strip_tags', 'trim', 'capitalize');` due to the way composition is implemented in PHP using closures.

So what do we do? Pretty much nothing. The goal of unit testing is to gain confidence in the fact that your code works in the expected way. In a traditional imperative approach, you mock as many dependencies as possible for the following reasons:

- Each dependency can depend on some state you need to provide, making your job tougher. Even worse, dependencies can have

dependencies of their own that also require some state, and so on.
- Imperative code can have side effects, which could lead to your function or some dependencies having issues. This means that without mocks, you are not only testing your function, but all other dependencies and the interaction between them; in other words, you are doing integration testing.
- Control structures introduce decision points, which can make reasoning about a function complex; this means that, if you reduce the number of moving pieces to the strict minimum, your function is easier to test. Mocking other function calls reduces this complexity.

When doing functional programming, the first issue is moot as there is no global state. Everything your dependencies will ever need is either already in the arguments to your tested function or will be computed along the way. So mocking dependencies will make you do more work instead of less.

Since our functions are pure and referentially transparent, there is no risk of side effects having any consequences on the computation result, meaning even if we have dependency, we are not doing integration testing. Sure, a bug in one of the functions that is called will result in an error, but hopefully it will also have been caught earlier by another test, making it clear what is happening.

Concerning the complexity, if we go back to our composed function, `$title = compose('strip_tags', 'trim', 'capitalize');`, I posit it is really easy for anyone to understand what is happening. If all three functions are already tested, there is nothing much that can go wrong, even if we were to rewrite this without the `compose` command:

```
<?php

function title(string $string): string
{
  $stripped = strip_tags($string);
  $trimmed = trim($stripped);
  return capitalize($trimmed);
}
```

There is not much to test here. Obviously, we would have to write some tests to ensure that we pass the right temporary value to each function and that the plumbing works as expected, but if we have confidence in all three called functions, we can have a lot of confidence that this function will work also.

This line of reasoning is only possible because we know due to the properties of referential transparency that none of the three functions will have any impact on any of the others in some subtle way, meaning that their own unit tests give us trust enough in the fact that they will not break.

The result of all this is that usually you will write fewer tests for functional code because you will gain trust quicker. However, it does not mean that the `title` function does not need to be tested, because you could have made a small mistake somewhere. Each component should still be tested, but probably with a bit less care in correctly isolating everything.

Obviously, we are not talking about database access, or third-party APIs, or services here; those should always be mocked for the same reasons as in any test suite.

Simplified mocking

This might already be clear, but I really want to stress the point that any mocking you will have to do will be greatly simplified.

First of all, you will only need to create the input arguments of the function under test. In some cases, this represents creating some pretty big data structures or instantiating complex classes, but at least you don't have to mock external states or a whole lot of services that are injected in your dependencies.

Also, this might not be true in all cases, but usually your functions operate on a smaller scale because they are a small part of something bigger, meaning that any one function will only take some really precise

and concise parameters.

Obviously, there will be exceptions, but not that many, and as we discussed earlier, since all of the parts making the big picture will already be tested. Your degree of confidence should then already be higher than is usually the case in a more imperative application.

Building blocks

Functional programming encourages the creation of small building blocks that get reused as part of bigger functions. Those small functions do usually only one thing. This makes them easier to understand, but also easier to test.

The more decision points a function has, the more difficult it is to come up with a way to test each possible execution path. A small specialized function has usually at most two of those decision points, making it fairly easy to test.

Bigger function usually don't perform any kind of control flow, they are just composed of our smaller blocks in a straightforward way. Since this means there is only one possible execution path, it also means that they are easy to test.

Closing words

Of course, I am not saying that you won't encounter some pure functions that are difficult to test. It's just than in general you will have less trouble writing your tests and you will also gain trust in your code quicker.

With the industry moving ever closer to methodologies such as TDD, this means that functional programming is really a good fit for a modern application. This is especially true once you realize that most advice you'll find in order to write "testable code" is already enforced by using only functional programming techniques.

Speeding up using parallelization

If you have ever searched for a solution to speed up your test suites, chances are that you found something about test parallelization. Usually, users of PHPUnit will find the **ParaTest** utility, for example.

The main idea is to run multiple PHP processes simultaneously in order to leverage all the processing power of the computer. This approach works for mostly two reasons:

- A single test run has bottlenecks such as disk speed for source file parsing or database access.
- PHP being single-threaded, a multi-core CPU, like nearly all computers have nowadays, is not used to its full potential by a single test run.

By running multiples tests in parallel, both those issues can be solved. The ability to do this is, however, limited by the fact that each test suite is independent from the others, a property that is already enforced by referential transparency in a functional codebase.

This means that, if the functions under test follow the functional principles, you can run all your tests in parallel without having to make any adaptation. In some cases, this could divide by ten the time taken for your whole test suite, greatly improving the feedback loop when you develop in the process.

If you are using PHPUnit utility, the aforementioned ParaTest utility is one of the easiest ways to get started. You can find it on GitHub at https://github.com/brianium/paratest. I advise you to use the `--functional` command-line parameter so that each function can be tested simultaneously instead of just the test cases.

There is also a brand-new utility for PHPUnit users called **PHPChunkIt**. I haven't had the opportunity to test it, but I hear it is interesting. You can find it on GitHub at https://github.com/jwage/phpchunkit.

Another more flexible option is using Fastest, available at https://github.com/liuggio/fastest. The examples shown in the tool documentation are for PHPUnit, but in theory it is able to run anything in parallel.

If you are using the atoum utility instead, by default your tests are already in what they call *concurrent* mode, which means they run in parallel. You can modify this behavior for each test using annotations as stated in the execution engine documentation at https://atoum-en.rtfd.org/en/latest/engine.html.

The **behat** framework users can use the **Parallel Runner** extension, also available on GitHub at https://github.com/shvetsgroup/ParallelRunner. If you are using **CodeCeption** framework, it is sadly a bit difficult to achieve; the documentation (http://codeception.com/docs/12-ParallelExecution) has, however, multiple possible solutions for you.

I strongly suggest you look into parallelizing your tests as it will be time well spent. Even if you are only able to save a few seconds on each run, this gain quickly accumulates. Faster tests means you will run them more often and this is usually a good way to improve code quality.

Property-based testing

Tired of spending time tediously writing test cases, John Hughes and Koen Claessen decided it was time for a change. A little more than 15 years ago, they wrote and published a paper about a new tool they called *QuickCheck*.

The main idea is that, instead of defining a list of possible input values and then asserting that the result is what we expect, you define a list of properties that characterize your function. The tool then generates as many test cases as wanted automatically and verifies that the property holds.

The default operating mode is for *QuickCheck* to generate random values and feed them to your functions. The result is then checked against the properties. If a failure is detected, the tool will then try to reduce the inputs to the minimal set of inputs generating the issue.

Having a tool generating as many testing values as you want is invaluable to find edge cases it would have taken you hours to think about. The fact that the test case is then reduced to its minimal form is also great to easily determine what is going wrong and how to fix it. It so happens that random values are not always the best way to test something. This is why you can also provide generators that will be used instead.

Also, thinking of your tests as a set of properties that need to hold true is a great way to focus more clearly on what the system is supposed to do instead of focusing on finding test values. This is especially helpful when doing TDD as your tests will be more akin to a specification.

If you want to learn more about this approach, the original paper is available online at http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf. The author uses Haskell in his paper but the content is however fairly easy to read and understand.

What exactly is a property?

A property is a rule that your function must respect in order to be determined correct. It can be something really simple, such as the result of a function adding to integers requiring also to be an integer, or anything more complex, such as verifying the monad laws.

You usually want to create properties that are not already enforced otherwise, be it by another property or the language. For example, if we use the scalar type systems introduced by PHP 7, our preceding integer example is not needed.

As an example, we will take something from the paper. Say we just wrote a function that reverses the order of elements in an array. The authors propose that this function should have the following properties:

- The `reverse([x]) == [x]` property reverses an array with a single element and should yield the exact same array
- The `reverse(reverse(x)) == x` property reverses an array twice and should yield the exact same array
- The `reverse(array_merge(x, y)) == array_merge(reverse(y), reverse(x))` property, reversing two merged arrays should yield the same result as merging the second array reversed to the first one reversed

The first two properties will guarantee that our function does not mess with the values. If we were to have only those two properties, a function doing absolutely nothing besides returning its parameter will pass the test with flying colors. This is where the third property comes into play. The way it is written ensures that our function does what we expect of it because there is no other way the property will hold.

What is interesting about those properties is that at no time do they perform any kind of computation. They are simple to implement and understand, meaning it is nearly impossible to introduce bugs in them. If you were to test your functions by somehow re-implementing the computation they are doing, it would kind of defeat the whole point.

Although pretty simple, this example shows perfectly that it is not easy to find valuable properties that are both meaningful and simple enough to ensure they will have no bugs. If you have trouble finding good properties, I encourage you to take an overview and think of your function in terms of the business logic you are trying to implement. Do not think in terms of inputs and outputs but try to see the broader picture.

Implementing the add function

A great explanation on why property based testing is a valuable tool can be found in a slide deck available online at http://www.slideshare.net/ScottWlaschin/an-introduction-to-property-based-testing. There is also a companion blog post with some more information at http://fsharpforfunandprofit.com/posts/property-based-testing-2/. I will try to summarize them quickly here.

A developer is asked to write a function adding two values with some tests. He writes two tests where the expected result is 4; everything is fine. The person asking for the function asks for more tests; they fail for the reason that the function was always returning the value 4 instead of doing anything meaningful.

The developer rewrites the function so that the tests pass again, but a new round of tests continues to fail. What was really done was to incorporate the results to the new tests as special cases in the original function. The excuse advanced by the developer is that they were following the TDD best practices saying that you need to *write the minimal code that will make the test pass*.

What is happening might seem stupid for such a simple function, but if you replace it with some kind of complicated business logic that needs to be implemented, such a story is probably more common that you would think and is also one of the pitfalls of TDD as stated by its opponent. If you follow TDD to the letter, your code will never be better than your tests.

The slide deck continues by introducing tests where each value is a random integer and the function is tested by comparing the result to `x + y`. In this case, there is no way the developer could cheat using special cases in its function. There is obviously another issue, however, you re-implemented the function inside the test to verify the result.

Enter property based testing. The first property implemented is `add(x, y) == add(y, x)`. The developer implements the `add` property as being `x * y`, which correctly passes the test.

This means we need a second property, for example, the `add(add(x, 1), 1) == add(x, 2)` property. This can also be beaten with the implementation of `x - y`, but in this case the first test will fail. This is why the developer's newest implementation is simply to return `0`.

At this point, a final property, `add(x, 0) == x` is added. The developer is finally forced to write a correct implementation for our function as he isn't able to find a way to cheat it this time.

If we go back to our final three properties and compare them to what we know about the addition properties in mathematics, we can draw the following comparison:

- In the `add(x, 0) == x` property, 0 is the *identity* of the addition
- In the `add(x, y) == add(y, x)` property, the addition is *commutative*
- In the `add(add(x, 1), 1) == add(x, 2)` property, the addition is *associative*

All three properties are in fact well-known properties of the operation we were trying to implement. As we said earlier, taking a step back and reflecting about the what instead of the who is a great help when coming up with properties.

The remainder of the slides are a great and interesting read, but as I don't want to plagiarize the entire content, I'd rather encourage you to go read it online. I will just take three more pieces of advice from them as I find them really great and easy to remember:

- **Different paths, same destination**: Come up with two different ways to get the same results using the function under tests, like we did for the third property of `reverse`.
- **There and back again**: If your function has an inverse, try applying both to see if you get the initial value back, like we did for the second property of `reverse`.
- **Some things never change**: If some properties of your input are not changed by the function, test for them, for example, array length or type of the data.

With all this, you should now have a good idea about how to find good properties for your functions. It is still a difficult task, but in the end you'll probably save a lot of time as you won't have to add edge cases as you find them.

If you'd like to have a good example of a real-life bug that was discovered thanks to property-based testing, John Hughes himself gave a great talk with some nice examples at https://vimeo.com/68383317.

The PhpQuickCheck testing library

Having seen the theoretical aspects of property-based testing in general, we can now shift our attention to a PHP-specific implementation-the `PhpQuickCheck` library. The source code is available on GitHub at https://github.com/steos/php-quickcheck and the library can be installed using **`composer`** command:

```
composer require steos/php-quickcheck -stability dev
```

You might need to change your `minimum-stability` setting to dev in your `composer.json` file, or add the dependency manually as explained on the GitHub page, because there is currently no stable release of the library.

The project was started in September 2014 and most of its development took place until November of the same year. Since then, not many new features have been added, mostly improvement of the coding styles and

some minor improvements.

Although we can't say the project is really alive today, it is one of the first serious attempts to have a `QuickCheck` library in PHP and it has some functionalities that are not yet available in its main contender and will be discussed later.

But let's not get ahead of ourselves; let's get back to our first example, the reverse function. Imagine we wrote the `array_reverse` function available in PHP and we needed to test it. This is how it would look with the `PhpQuickCheck` library:

```
<?php

use QCheck\Generator;
use QCheck\Quick;

$singleElement = Quick::check(1000, Generator::forAll(
    [Generator::ints()],
    function($i) {
        return array_reverse([$i]) == [$i];
    }
), ['echo' => true]);

$inverse = Quick::check(1000, Generator::forAll(
    [Generator::ints()->intoArrays()],
    function($array) {
        return array_reverse(array_reverse($array)) == $array;
    }
), ['echo' => true]);

$merge = Quick::check(1000, Generator::forAll(
    [Generator::ints()->intoArrays(), Generator::ints()-
>intoArrays()],
    function($x, $y) {
        return
            array_reverse(array_merge($x, $y)) ==
            array_merge(array_reverse($y), array_reverse($x));
    }
), ['echo' => true]);
```

The `check` static method accepts the amount of test data it needs to generate as the first argument. The second argument is an instance of

`Generator` function; usually, you will use `Generator::forAll` to create it in the example. The last part is an array of options you can pass in the random generator `seed` variable, the `max_size` function for the generated data (the meaning of this value depends on the generator used), or finally the `echo` options which will display a dot (`.`) for each passed test.

The `forAll` instance accepts an array representing the arguments to your test and the test itself. In our case, for the first test, we generate random integers and for the other two, random integer arrays. The test must return a Boolean value: `true` for passed, `false` otherwise.

If you were to run our little example, it would display a dot for each random data generated, because we passed the `echo` option. The resulting variable contains information about the test results themselves. In our case, if you displayed `$merge`, it would show:

```
array(3) {
  ["result"]=> bool(true)
  ["num_tests"]=> int(1000)
  ["seed"]=> int(1478161013564)
}
```

The `seed` instance will be different on each run except if you pass one as parameter. Reusing the `seed` instance allows you to create the exact same test data. This can be useful to check whether a particular edge case is correctly fixed after being discovered.

An interesting feature is automatically determining which generator to use based on type annotations. You can do so using methods on the `Annotation` class:

```
<?php

/**
 * @param string $s
 * @return bool
 */
function my_function($s) {
    return is_string($s);
}
```

```
Annotation::check('my_function');
```

This feature can, however, only work with annotation right now and type hints will be ignored.

As you can see with those small examples, the `PhpQuickCheck` library relies heavily on static functions. The codebase in itself is also sometimes a bit hard to understand and the library lacks good documentation and an active community.

All in all, I don't think I would recommend using this over the option we'll see next. I just wanted to present the library to you as a possible alternative and, who knows, its status might change in the future.

Eris

Eris development started out in November 2014, roughly at the time the `PhpQuickCheck` library got its last big feature introduced. As we will see, the coding style is definitively more modern. Everything is cleanly organized in namespace and helpers take the form of functions instead of static methods.

As usual, you can get Eris using the **`composer`** command:

```
composer require giorgiosironi/eris
```

The documentation is available online at http://eris.rtfd.org/ and it is quite complete. The only gripe I have with it is that the sole examples are for people using PHPUnit to run their test suites. It should be doable to use it with other tests runners, but this is something that isn't documented for now.

If we wanted to use Eris to test the properties we defined for `array_reduce`, our test case would look something like this:

```
<?php

use Eris\Generator;
```

```
class ArrayReverseTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testSingleElement()
    {
        $this->forAll(Generator\vector(1, Generator\nat()))
             ->then(function ($x) {
                 $this->assertEquals($x, array_reverse($x));
             });
    }

    public function testInverse()
    {
      $this->forAll(Generator\seq(Generator\nat()))
           ->then(function ($x) {
               $this->assertEquals($x,
array_reverse(array_reverse($x)));
           });
    }

    public function testMerge()
    {
      $this->forAll(
               Generator\seq(Generator\nat()),
               Generator\seq(Generator\nat())
           )
           ->then(function ($x, $y) {
               $this->assertEquals(
                   array_reverse(array_merge($x, $y)),
                   array_merge(array_reverse($y),
array_reverse($x))
               );
           });
    }
}
```

The code is somewhat similar to what we wrote for the `PhpQuickCheck` library but leverages methods that are added by the provided trait to our test case and generator functions instead of static methods. The `forAll` method accepts a list of generators representing the arguments to our test function. You can subsequently use the `then` keyword to define the function. You have access to all asserters provided by PHPUnit.

The documentation explains in detail how you can configure various aspects of the library, such as the amount of generated test data, limiting the execution time, and so on. Each generator is also detailed at length with various examples and use cases.

Let's see what happens when we have a failing test case. Imagine we want to prove that no strings are also a numerical value; we could write the following test:

```
<?php

class StringAreNotNumbersTest extends
\PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testStrings()
    {
        $this->limitTo(1000)
             ->forAll(Generator\string())
             ->then(function ($s) {
        $this->assertFalse(is_numeric($s),"'$s' is a numeric
value.");});
    }
}
```

You can see how we raised the number of iterations using the `limitTo` function to 1,000 from the default of 100. This is because a lot of strings are in fact not numerical values and without this raise, I was only able to get a failure one test out of three. Even with this higher limit, it is still possible that sometimes all test data will pass the test without failures.

This is the kind of output you would get:

```
PHPUnit 5.6.2 by Sebastian Bergmann and contributors.
F 1 / 1 (100%)
Reproduce with:
ERIS_SEED=1478176692904359 vendor/bin/phpunit --filter
StringAreNotNumbersTest::testStrings

Time: 42 ms, Memory: 4.00MB

There was 1 failure:
```

```
1) StringAreNotNumbersTest::testStrings
'9' is a numeric value.
Failed asserting that true is false.

./src/test.php:55
./src/Quantifier/Evaluation.php:51
./src/Quantifier/ForAll.php:154
./src/Quantifier/ForAll.php:180
./src/test.php:57

FAILURES!
Tests: 1, Assertions: 160, Failures: 1.
```

The test failed after 160 iterations with the string `"9"`. Eris also gives you the command to run if you want to reproduce exactly this failing test by seeding the random generator manually:

```
ERIS_SEED=1478176692904359 vendor/bin/phpunit -filter
StringAreNotNumbersTest::testStrings".
```

As you can see, the library is fairly easy to use when your tests are written for PHPUnit. Otherwise, you might need to do some adaptation but I think it is worth your time.

Closing words

The `QuickCheck` library is easier to use in strictly typed functional programming language because it is sufficient to declare generators for certain types and some properties for your functions, and nearly everything else can be done automatically. The `PhpQuickCheck` library tries to emulate this behavior but the result is a bit tedious to use.

However, this doesn't mean you can't use property-based testing effectively in PHP! Once you have created your generators, the framework will use it to generate as much test data as you let it, possibly uncovering edge cases you would never have thought of. For example, there is a bug in the `DateTime` method's implementation in PHP that arises on leap years and could easily be overlooked when creating test data manually. See the *Testing the language* part at

http://www.giorgiosironi.com/2015/06/property-based-testing-primer.html (by the creator of Eris) for more details on the issue.

Writing properties can be challenging, especially in the beginning. But more often than not, it helps you reason about the feature you are implementing and will probably lead to better code because you took the time to think about it from a different angle.

Summary

In this chapter, we had a quick look at what can be done on the testing front when you use a more functional approach to programming. As we saw, functional code is often easier to test because it enforces what is considered best practice for testing when doing imperative coding.

By having no side-effects and explicit dependencies, you can avoid most of the issues you usually encounter when writing tests. This results in less time spent testing and more time to concentrate on your application.

We also discovered property-based testing, which is a great way to discover issues related to edge cases. It also allows us you to take a step back and think about the properties you want to enforce for your functions, which is akin to creating a specification for them. This approach is particularly effective when doing TDD as it forces you to think about what you want instead of how to do it.

Now that we have discussed testing to ensure our functions do what they should, we will learn about code optimization in order to allow for application performance in the next chapter. A well-tested codebase will help you do the necessary refactoring to achieve better speed.

Chapter 9. Performance Efficiency

Now that we have covered various techniques related to functional programming, it is time to analyze how it impacts performance in a language such as PHP, which is still imperative at its core even if there are more and more functional features introduced with each version.

We will also discuss why performance does not matter so much in the end and how we can leverage memoization and other techniques to alleviate this issue in some cases.

We will also explore two optimization techniques enabled by referential transparency. The first one will be memoization, which is a type of caching. We will also speak about running long computations in parallel and how you can take advantage of this in PHP.

In this chapter, we will cover the following topics:

- Performance impact of functional programming
- Memoization
- Parallelization of computation

Performance impact

Since there is no core support for features such as currying and function composition, they need to be emulated using anonymous wrapper functions. Obviously, this comes with a performance cost. Also, as we have already discussed in the part about tail-call recursion in Chapter 7, *Functional Techniques and Topics*, using a trampoline is also slower. But how much execution time do you lose compared to a more traditional approach?

Let's create a few functions that will serve as a benchmark and test the various speeds we can achieve. The function will execute a really simple

task, adding two numbers, in order to ensure we measure the overhead as effectively as possible:

```
<?php

use Functional as f;

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

function manualCurryAdd($a, $b = null) {
    $func = function($b) use($a) {
        return $a + $b;
    };

    return func_num_args() > 1 ? $func($b) : $func;
}

$curryiedAdd = f\curry('add');

function add2($b)
{
    return $b + 2;
}

function add4($b)
{
    return $b + 4;
}

$composedAdd4 = f\compose('add2', 'add2');

$composerCurryedAdd = f\compose($curryiedAdd(2),
$curryiedAdd(2));
```

We created the first function `add` and curryied it; this will be our first benchmark. We will then compare a specialized function adding `4` to a value to two different compositions. The first will be the composition of two specialized functions and the second the composition of two curryied versions of the `add` function.

We will use the following code to benchmark our functions. It is pretty

basic but it should suffice to demonstrate any meaningful differences:

```
<?php

use Oefenweb\Statistics\Statistics;

function benchmark($function, $params, $expected)
{
    $iteration   = 10;
    $computation = 2000000;

    $times = array_map(function() use($computation, $function,
$params, $expected) {
        $start = microtime(true);

        array_reduce(range(0, $computation),
function($expected)  use ($function, $params) {
            if(($res = call_user_func_array($function,
$params))  !== $expected) {
                throw new RuntimeException("Faulty
computation");
            }

            return $expected;
        }, $expected);

        return microtime(true) - $start;
    }, range(0, $iteration));

    echo sprintf("mean: %02.3f seconds\n",
Statistics::mean($times));
    echo sprintf("std:  %02.3f seconds\n",
Statistics::standardDeviation($times)); }
```

The statistics methods are from the `oefenweb/statistics` package available via composer. We also check that the returned value is the one we expect as an extra precaution. We will run each function 2 million times 10 times in a row and display the mean time for the 2 million runs.

Let's run the benchmark for currying first. The displayed results are for PHP 7.0.12. When trying this with PHP 5.6, all benchmarks are slower but they exhibit the same differences between the various functions:

```
<?php
```

```
benchmark('add', [21, 33], 54);
// mean: 0.447 seconds
// std:  0.015 seconds

benchmark('manualCurryAdd', [21, 33], 54);
// mean: 1.210 seconds
// std:  0.016 seconds

benchmark($curryiedAdd, [21, 33], 54);
// mean: 1.476 seconds
// std:  0.007 seconds
```

Obviously, the results will vary depending on the system the test is run on, but the relative difference should stay roughly the same.

First, if we look at the standard deviation, we can see that each of the 10 runs took mostly the same time, which indicates that we can trust our numbers to be a good indicator of the performances.

We can see the curryied version is definitively slower. Manual currying is a bit more efficient, but both curryied versions were mostly three times slower than the simple function version. Before drawing conclusions, let's see the results for the composed functions:

```
<?php

benchmark('add4', [10], 14);
// mean: 0.434 seconds
// std:  0.001 seconds

benchmark($composedAdd4, [10], 14);
// mean: 1.362 seconds
// std:  0.005 seconds

benchmark($composerCurryedAdd, [10], 14);
// mean: 3.555 seconds
// std:  0.018 seconds
```

Again, the standard deviation is small enough so that we can consider the numbers valid.

Concerning the values themselves, we can see that the composition is

also about three times slower and the composition of the curryied function is, without much surprise, nine times slower.

Now if we take our worst case at `3.55` seconds against our best case at `0.434` seconds, this means we have an overhead of `3` seconds when using composition and currying. Does it matter? Does it seem like a lot of lost time? Let's try to imagine these numbers in the context of a web application.

Does the overhead matter?

We performed two million executions of our methods and it accounted for three seconds. A recent project I worked on, an e-commerce application for a luxury brand available in 26 countries and more than 10 languages and written entirely from scratch without the help of any framework, had more or less 25,000 function calls to render one page.

Even if we admit that all of those calls are made to composed functions that were curryied beforehand, this means that the overhead is now around 40 milliseconds in the worst case scenario. The application in question took roughly 180 milliseconds to display a page, so we are speaking of a 20-25% decrease in performance.

It is still a lot, but far from the three times slower figure we were seeing before. The overhead linked to the functional techniques will grow linearly with each function call. In the benchmark it seems great because the performed computation is trivial. In a real-life application, you have external bottlenecks such as databases, third-party APIs, or filesystems. You also have functions performing complex computations taking more time than a simple addition. In such a case the introduced overhead is a smaller part of the total execution time of your application.

This is also a worst-case scenario where we assume everything is composed and curryied. In a real-world application, you might use a traditional framework containing functions and methods without overhead. You will also be able to identify hot paths in your code and manually optimize them using explicit currying and composition instead

of the helper functions. It is also not necessary to curry everything; you will have functions with only one argument that won't need it and some functions for which it makes no sense to use currying.

Also, those are numbers to be considered for an application with a cold cache. Any mechanism you already have in place to reduce the rendering time of you pages will continue to work the same. For example, if you have a Varnish instance running, your pages will probably continue to be served at the same speed.

Let's not forget

We compared a really small function to composition and currying. A modern PHP codebase would use classes both for holding business logic and values. Let's simulate this using the following implementation of our `add` function:

```
<?php

class Integer {
    private $value;
    public function __construct($v) { $this->value = $v; }
    public function get() { return $this->value; }
}

class Adder {
    public function add(Integer $a, Integer $b) {
        return $a->get() + $b->get();
    }
}
```

The time taken by the traditional approach would increase:

```
<?php

benchmark([new Adder, 'add'], [new Integer(21), new
Integer(33)], 54);
// mean: 0.767 seconds
// std:  0.019 seconds
```

Just by wrapping everything in a class and using a getter, the execution time nearly doubled, meaning suddenly the functional approach is only

1.5 times slower in the benchmark, and the overhead in our sample application is now 10-15%, which is already a lot better.

Can we do something?

Sadly, there is not really something that we could do ourselves. We could shave off a little bit of time with more efficient implementation of the `curry` and `compose` methods as we demonstrated using the manually curryied version of the `add` method, but this won't amount to much.

An implementation of both those techniques as a core part of PHP would, however, bring a lot of benefits, probably getting them on a par with traditional functions and methods, or really close. But, as far as I know, there is no plan to do so in the near future.

It could also be possible to create a C language extension for PHP to implement those two functions in a more efficient way. This will, however, be impractical as most PHP-hosting companies do not let people install custom extensions.

Closing words

As we just saw, using techniques such as currying and function composition has an impact on performance that is rather hard to mitigate on your own. In my opinion, the benefits outweigh this cost but it is important to switch to functional programming knowingly.

Also, most web applications nowadays have some kind of caching mechanism in front of the PHP application. So the only cost would be when populating this cache. If you are in such a situation, I see no reason to avoid using the techniques we learned.

Memoization

Memoization is an optimization technique where the result of an expensive function is stored so that it can be returned directly in any subsequent call with the same parameters. It is a specific case of data caching.

Although it can be used on non-pure functions with the same invalidation issues as any other cache mechanism, it is mostly used in functional languages where all functions are pure, thus simplifying greatly its usage.

The idea is to trade computation time for storage space. The first time you call a function for a given input, the result is stored and the next time the same function is called with the same arguments, the already computed result can be returned immediately. This can be achieved fairly easily in PHP using the `static` keyword inside your function:

```
<?php

function long_computation($n)
{
    static $cache = [];
    $key = md5(serialize($n));

    if(! isset($cache[$key])) {
        // your computation comes here, the rest is boilerplate
        sleep(2);
        $cache[$key] = $n;
    }

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

There are obviously a dozen different ways to do something similar, but this is simple enough to get an idea of how it works. One can also imagine implementing an expiration mechanism, or, since we use memory space instead of computation time, some kind of data structure where values get erased when not used to make room for newer results.

Another option would be to store the information to disk, to keep the values saved between multiple runs of the same script, for example. There exists at least one library in PHP (https://github.com/koktut/php-memoize) doing just that.

The library, however, does not work well with recursive calls out-of-the-box, as the function itself is not modified and thus the value will only be saved for the first call, not the recursive ones. The article (http://eddmann.com/posts/implementing-and-using-memoization-in-php/) linked in the library readme discusses this issue in more detail and proposes a solution.

It is interesting to note that **Hack** has an attribute that will automatically memoize the results of a function with arguments of a certain type (https://docs.hhvm.com/hack/attributes/special#__memoize). If you are using Hack and want to use the annotation, I recommend you read the *Gotchas* section first as it might not always do what you want.

Note

Hack is a language adding new features on top of PHP and running on the PHP Virtual Machine written by Facebook-the **HipHop Virtual Machine** (**HHVM**). Any PHP code is compatible with Hack, but Hack adds some new syntax, making the code incompatible with the vanilla PHP interpreter. For more information, you can visit http://hacklang.org/.

Haskell, Scala, and memoization

Neither Haskell nor Scala performs memoization automatically. And none of the two has something in their core to do so, although you can find multiple libraries offering this feature.

There is a misconception that Haskell memoizes all functions by default, which is due to the fact that the language is lazy. What really happens is that Haskell tries to delay the computation of a function call as much as possible and, once it does so, it uses the referential transparency

property to replace other similar calls with the computed values.

There are, however, multiple cases where this replacement cannot happen automatically, leaving no other choice than to compute the value again. If you are interested in the topic, this *Stack Overflow* question is a good start with all the right keywords at http://stackoverflow.com/questions/3951012/when-is-memoization-automatic-in-ghc-haskell.

We'll leave the discussion here, as this book is about PHP.

Closing words

This was a really quick presentation of memoization as the technique is fairly simple to implement and there is nothing much to really say about it. I just wanted to present it so you are aware of the term.

If you have some long-running computation that gets called multiple times with the same arguments, I recommend you use the technique as it can really speed up things and requires nothing from the caller. It is really transparent to use.

Beware that it is, however, not a silver bullet. Depending on the data structure of your return values, it can eat up memory pretty quickly. If you encounter this issue, you can use some mechanism to clean up older, or less used, values from the cache.

Parallelization of computation

Another nice benefit of having pure functions is that you can divide a computation into multiple small parts, distribute the workload, and assemble the result. It is possible to do so for any mapping, filtering, and folding operations. The function used for folding needs to be monoidal as we will see. Functions for mapping and filtering have no particular constraint besides purity.

Mapping does not have any particular constraint beside a pure function. Say you have four cores, or computers; you will only need to follow these steps:

1. Split the array into four parts.
2. Send a part to each core to do the mapping.
3. Merge the results.

In this particular case, it might be slower than doing it on a single core as the merging operation adds an overhead. However, as soon as the computation takes longer, you are able to use more of the computing power at your disposal, and thus gain time.

Filtering operates in exactly the same way as mapping, except you send a predicate instead of a function.

Folding can only happen if you have a monoidal operation, because each split needs to start with the empty value otherwise it might skew the results:

1. Split the array into four parts.
2. Send a part to each core to do the folding with the empty value as initial value.
3. Put all results in a new array.
4. Perform the same fold operation on the new array.

If your collection is really big, you can again divide the final fold into

multiple parts.

Parallel tasks in PHP

PHP was created when computers had only one core and since then the traditional way of using it is to have a single thread to serve each request. You can have multiple workers declared in your web server to serve different requests using different processes, but one request will usually use only one thread, and thus only one core.

Although a thread-safe version of the PHP binary exists, Linux distributions usually ships the non-thread-safe one because of the aforementioned reason. This does not mean it is impossible to parallelize tasks in PHP, but it sure makes it a lot harder.

The pthreads extension

PHP 7 has seen the release of a new version of the **pthreads** extension, which allows you to run multiple tasks in parallel using a newly designed object-oriented API. It is really great and there is even a *polyfill* to perform the tasks sequentially if the extension is not available.

Note

The term *polyfill* originated in JavaScript development. It is a small piece of code that replaces a feature that is not implemented in the user's browser. Another term that is sometimes used is *shim*. In our case, the *pthreads-polyfill* provides an API in all points similar to the one of the extension but that runs the tasks sequentially.

Sadly, using the extension is kind of a challenge. First of all, you need to have a thread-safe PHP binary, also called a **ZTS** binary for **Zend Thread-safe**. As we just saw, distributions usually don't ship this version. As far as I know, there are currently no distributions with official PHP packages having ZTS enabled. Google is usually helpful when trying to find instructions to create your own ZTS binary for your Linux distribution.

Windows and Mac OS users are in a better place as you can download ZTS binaries on http://www.php.net and you can enable the option when installing PHP with the `homebrew` package manager.

The other limitation is that the extension will refuse to load in a CGI context. This means you will only be able to use it on the command line. If you are interested in the reason the maintainer of the pthreads extension chose to put this constraint in place, I suggest you read this blog post he wrote, at http://blog.krakjoe.ninja/2015/09/the-worth-of-advice.html.

Now, if we assume you are able to have a ZTS version of PHP and you are only writing a CLI application, let's see how we could perform a parallel fold using the **pthreads** extension. The extension is hosted on GitHub at https://github.com/krakjoe/pthreads, and installation instructions can be found in the official PHP documentation at http://docs.php.net/manual/en/book.pthreads.php.

There are obviously multiple ways we could go about implementing folding using threads. We will try to go for a generic method. In some cases, a more specialized version might be quicker but this should cover a whole range of use cases already:

```
<?php

class Folder extends Thread {
    private $collection;
    private $callable;
    private $initial;

    private $results;

    private function __construct($callable, $collection,
$initial)
    {
        $this->callable = $callable;
        $this->collection = $collection;
        $this->initial = $initial;
    }

    public function run()
```

```
    {
        $this->results = array_reduce($this->collection, $this-
>callable, $this->initial);
    }

    public static function fold($callable, array $collection,
$initial, $threads=4)
    {
        $chunks = array_chunk($collection,
ceil(count($collection) / $threads));

        $threads = array_map(function($i) use ($chunks,
$callable,  $initial) {
            $t = new static($callable, $chunks[$i], $initial);
            $t->start();
            return $t;
        }, range(0, $threads - 1));

        $results = array_map(function(Thread $t) {
            $t->join();
            return $t->results;
        }, $threads);

        return array_reduce($results, $callable, $initial);
    }
}
```

The implementation is pretty simple; we have a simple `Thread` performing the reducing of each chunk and we combine them at the end using a simple `array_reduce` function. We could have opted to use a `Pool` instance to manage the various threads but, in such a simple case, it would have complicated the implementation.

Another possibility would have been to recurse until the resulting arrays contain at most `$threads` elements; this way, we would have used the full computational power at our disposal until the end. But again, this would have complicated the implementation.

How do you use it? Just call the static method:

```
<?php

$add = function($a, $b) {
```

```
    return $a + $b;
};

$collection = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

echo Folder::fold($add, $collection, 0);
// 55
```

If you want to play with this idea a bit, a little library implements all three higher-order functions in a parallel way (https://github.com/functional-php/parallel). You can install it using composer:

```
composer require functional-php/parallel
```

Messaging queues

Another option in PHP to parallelize tasks is to use a messaging queue. Message queues provide an asynchronous communication protocol. You will have a server that will hold the messages until one or multiple clients retrieve them.

We can implement parallel computation by having our application send X messages to the server, one for each distributed task. A certain number of workers will then retrieve the messages and perform the computation, sending back the result to the application as a new message.

There are a lot of different message queue implementations that you can use. Usually, the queue itself is not implemented in PHP, but most of them have a client implementation that you can use. We will use **RabbitMQ** and the **php-amqplib** client.

Explaining how to install the server is out of scope for this book, but you have a lot of tutorials available on the Internet. We will also not explain all the details about the implementation, only what is related to our topic. You can install the PHP library using composer:

```
composer require php-amqplib/php-amqplib
```

We will need both an implementation for our workers and the application. Let's first create a file containing the common parts, which we will call `09-rabbitmq.php`:

```
<?php

require_once './vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;

$connection = new AMQPStreamConnection('localhost', 5672,
'guest',  'guest');
$channel = $connection->channel();
list($queue, ,) = $channel->queue_declare($queue_name, false,
false, false, false);

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

Now we create the worker:

```
<?php
use PhpAmqpLib\Message\AMQPMessage;

$queue_name = 'fold_queue';
require_once('09-rabbitmq.php');

function callback($r) {
    global $fold_function;

    $data = unserialize($r->body);

    $result = array_reduce($data['collection'], $fold_function,
$data['initial']);

    $msg = new AMQPMessage(serialize($result));

    $r->delivery_info['channel']->basic_publish($msg, '', $r-
>get('reply_to'));
    $r->delivery_info['channel']->basic_ack($r-
>delivery_info['delivery_tag']);
};

$channel->basic_qos(null, 1, null);
$channel->basic_consume('fold_queue', '', false, false, false,
```

```
false, 'callback');

while(count($channel->callbacks)) {
    $channel->wait();
}

$channel->close();
$connection->close();
```

And now we create the application itself:

```
<?php
use PhpAmqpLib\Message\AMQPMessage;

$queue_name = '';
require_once('09-rabbitmq.php');

function send($channel, $queue, $chunk, $initial)
{
    $data = [
        'collection' => $chunk,
        'initial' => $initial
    ];
    $msg = new AMQPMessage(serialize($data), array('reply_to'
=>  $queue));
    $channel->basic_publish($msg, '', 'fold_queue');
}

class Results {
    private $results = [];
    private $channel;

    public function register($channel, $queue)
    {
        $this->channel = $channel;
        $channel->basic_consume($queue, '', false, false,
false,  false, [$this, 'process']);
    }

    public function process($rep)
    {
        $this->results[] = unserialize($rep->body);
    }

    public function get($expected)
    {
```

```
        while(count($this->results) < $expected) {
            $this->channel->wait();
        }

        return $this->results;
    }
}

$results = new Results();
$results->register($channel, $queue);

$initial = 0;

send($channel, $queue, [1, 2, 3], 0);
send($channel, $queue, [4, 5, 6], 0);
send($channel, $queue, [7, 8, 9], 0);
send($channel, $queue, [10], 0);

echo array_reduce($results->get(4), $fold_function, $initial);
// 55
```

Obviously, this is a really naive implementation. Requiring files like that is bad practice since PHP 5 at least and the code is really brittle, but it serves the purpose of demonstrating the possibilities offered by a message queue.

When you launch the worker, it registers itself as a consumer of the `fold_queue` queue. When a message is received, it uses the folding function declared in the common part on the data and sends the result back on the queue defined as the reply to. The loop ensures we wait for incoming messages; given the code, the worker should never exit by itself.

The application has a `send` function that sends messages on the `fold_queue` queue. The `Results` class instance registers itself as a consumer of the default queue in order to receive the results of each worker. Then four messages are sent, and we ask the `Results` instance to wait for them. Finally, we reduce the received data to get the final result.

If you launch only one worker, the results will be sent sequentially; however, if you launch multiple workers, each one of them will retrieve

a message from the RabbitMQ server and process it, thus enabling parallelization.

Compared to using threads, a message queue has multiple benefits:

- The workers can be on multiple computers
- The worker can be implemented in any other language having a client for the chosen queue
- The queue server provides redundancy and failover mechanisms
- The queue server can perform load-balancing between the workers

Using the pthreads library when available is probably a bit easier if you plan on only distributing your workload across the cores of a unique computer, but if you want to have more flexibility, message queues are the way to go.

Other options

There are other ways to start parallelizing computations in PHP, but usually they make retrieving the values more difficult than what we just saw.

One option is to use the `curl_multi_exec` function to execute multiple HTTP requests asynchronously. The general structure would be similar to what we used in the message queue example. However, the possibilities are also limited compared to the full power of a complete messaging system.

You can also create other PHP processes using one of the multiple related functions. In this case, the difficulty is often to pass and retrieve the data without loss as the way to do so will depend on a number of factors related to the environment. If you want to go this way, the `popen`, `exec`, or `passthru` functions are probably your best bet.

If you don't want to do all the grunt work, you can also use the `Parallel.php` library, which abstracts most of the complexity away. You can install it using composer:

```
composer require kzykhys/parallel
```

The documentation is available on GitHub at https://github.com/kzykhys/Parallel.php. As the library uses Unix sockets, most of the issues related to data loss are gone. However, you won't be able to use it on Windows.

Closing words

As we saw, it might not be the easiest thing to work with multiple threads or processes in PHP, especially when in the context of a web page. This can, however, be achieved and it can greatly speed up long computations.

With the rewrite of pthreads for PHP 7, we can hope that more Linux distributions and hosting companies will start providing a ZTS version.

If this is the case, and parallel computation starts becoming a real thing in PHP, it might be possible to do some light big data processing without having to resort to external libraries in other languages such as the **Hadoop framework**.

I want to finish with a few words about message queues. Even if you don't use them in a functional way to process data and get results back, they are a great way to perform lengthy operations in the context of a web request. For example, if you give your users a way to upload a bunch of images and you need to process them, you can enqueue the operation and return immediately to the user. The queued message will be processed in due time and your user won't have to wait.

Summary

In this chapter, we discovered that sadly there is a cost to pay when doing functional programming. Since PHP has no core support for features such as currying and function composition, there is an overhead linked to the wrapper functions when using them. This can obviously be an issue in some cases, but caching can often alleviate this cost.

We talked about memoization, a caching technique that, coupled with pure functions, allows you to speed up subsequent calls to a given function without having to invalidate the stored results.

Finally, we discussed parallelizing computations in PHP by leveraging the fact that any pure operation performed on a collection can be distributed across multiple nodes without having any concerns about shared state.

The next chapter will be dedicated to developers using a framework as we will discover how we can leverage the techniques we've learned so far in the context of the most common frameworks currently in use in the PHP world.

Chapter 10. PHP Frameworks and FP

Now that we've seen how functional programming can be used to solve common programming issues, it is time we apply these techniques when developing with a framework. This chapter will present various ways to do so with a few of the most common PHP frameworks.

Before we start, a little disclaimer. I am by no means an expert in each of the frameworks we will discuss here. I have worked with all of them at different levels, but this doesn't mean I know everything there is to know about them. So, it is possible I will not be presenting the latest best practices here, despite the research conducted while writing this chapter.

This being said, we won't write a lot of code in this chapter. We will mainly look at how you can interface existing functional code with the framework structure and how you can leverage the various framework features to help you write in a functional way. We will also discuss the pros and cons of each framework regarding functional programming.

In this chapter, we will have a look at the following frameworks:

- Symfony
- Laravel
- Drupal
- WordPress

I hear some of you in the background whispering that Drupal and WordPress are not frameworks but content management systems. I agree with you, but keep in mind that people are using both of them to create full-blown applications with e-commerce and other features, so they have their place here.

Also, the **CodeIgniter** framework is missing from the list as I have never worked with it. However, you can probably use most of the advice that will be presented here with any framework, including CodeIgniter.

As a matter of fact, most of the advice in each part is useful in a variety of contexts. This is why I strongly suggest you read the sections about each framework. This will allow me to avoid repeating myself too much.

Symfony

With the focus firmly on Dependency Injection, the Symfony framework is well suited to write functional code. Symfony developers are accustomed to declaring their controllers and services in a way that explicitly define their dependencies.

We could argue that injecting the whole container is a bit problematic. In a strict sense, the controller and service can still be pure, but obviously the cognitive burden is a bit heavier since you need to read the code to know exactly which dependency is used.

In this part, we will discuss what parts of Symfony are well suited to functional programming and where you need to be cautious. We won't be able to cover everything as Symfony is a really complete framework with a lot of components but is should suffice to get you started.

Handling the request

The original `Request` class is not compliant with the `PSR-7 HTTP` message interfaces we already spoke about. This means it is not immutable, as the specification proposes. It is, however, really easy to obtain a PSR version of the request if you are using at least version 3.0.7 of the `SensioFrameworkExtraBundle` framework. You only need to install the required dependencies using Composer and change your controller actions signature a bit:

```
composer require sensio/framework-extra-bundle
composer require symfony/psr-http-message-bridge
composer require zendframework/zend-diactoros
```

Your controller needs to use the new `ServerRequestInterface` classes instead of the more traditional `Request` class in its method signature:

```
<?php

namespace AppBundle\Controller;

use Psr\Http\Message\ServerRequestInterface;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Zend\Diactoros\Response;

class DefaultController extends Controller
{
    public function indexAction(ServerRequestInterface
$request)
    {
        return new Response();
    }
}
```

If, for any reason, you need to get hold of a `Request` or `Response` instance that is compatible with Symfony's interfaces, you can use the **Symfony PSR-7** Bridge.

If you correctly inject only the dependencies you need in your controllers and services, with the help of the new PSR-7-compliant HTTP messages, you are now well prepared to write functional code for your Symfony application.

Database entities

One challenge you might encounter when writing completely pure function code is database access. Usually, developers use **Doctrine** with Symfony and, as far as I know, Doctrine does not yet have any facilities to help with writing referentially transparent code.

Using something like an IO monad is also cumbersome in the context of a framework because, at each entry and exit point of your functions, you will have to either encapsulate the parameters or transform the results to the format expected by the framework.

We will try to see how we can mitigate this issue using various techniques. While we are at it, we will also learn how to leverage the

`Maybe` type when using Doctrine.

Embeddables

Although not strictly related to functional programming, the idea of value objects can be used to attain some sort of immutability on your entities. It is also an idea worth exploring for its own benefits.

It is an idea we have already discussed in Chapter 2, *Pure Functions, Referential Transparency, and Immutability*. I will, however, take this opportunity to give a somewhat different definition taken from domain-driven design:

- **Entity**: something that has an identity independent from its properties.
- **Value object**: something that has no identity separate from its properties.

A common example is a person having an address. The person is an entity and the address is a value object. If you change the name, address, or any other property of a person, it is still the same person. However, if you change anything on the address, it becomes a totally different address.

Doctrine implements this idea under the name **Embeddables**. The term comes from the fact that a value object is always related to an entity, or embedded, as it has no point of existing on its own. You can find documentation on the official website at http://docs.doctrine-project.org/en/latest/tutorials/embeddables.html.

Although I doesn't recommend hijacking this feature, to implement immutability for every relations you have, I strongly urge you to think of embeddables when designing your entities and using them whenever you can. It will help you with both coding functionally and improving the quality of your data model.

Avoiding setters

If you start looking for best practices regarding the use of Doctrine and

most ORMs, there is a good chance you will one day find someone proposing we avoid creating setter methods. There are usually a multitude of good arguments to do so. In our case, we will just concentrate on one of them-we want immutable entities to help us write pure functional code.

In most cases, the proposed solution to get rid of setters will be to think in terms of tasks. For example, instead of having a `setState` and a `setPublicationDate` method setter on a `BlogPost` class, you will have a `publish` method, which will in turn change those two fields.

This is great advice as it allows you to have most of the business logic inside the entity where it belongs and it avoids having the object in some weird state because not all necessary steps were taken by the developer. A traditional class with setters would be something like the following:

```
<?php

class BlogPost
{
    private $status;
    private $publicationDate;

    public function setStatus(string $s)
    {
        $this->status = $s;
    }

    public function setPublicationDate(DateTime $d)
    {
        $this->publicationDate = $d;
    }
}
```

It can be transformed to the following implementation:

```
<?php

class BlogPost2
{
    private $status;
    private $publicationDate;
```

```
    public function publish(DateTime $d)
    {
        $this->status = 'published';
        $this->publicationDate = $d;
    }
}
```

As you can see, we modify the values in place, leaving us with a side-effect. We might naively think that it's enough to clone the current object in the `publish` method and return the new version with the modified properties to obtain an immutable version of our method; sadly, this solution does not work.

Doctrine stores which entities are managed by one of its units of work and a cloned entity is not in a managed state. We could attach the entity using some trick but then we would be in one of two situations:

- Both entities are managed, leading to possible issues with the internal coherence of metadata inside Doctrine itself
- Only the latest entity is managed, meaning our call to the `publish` method had the side effect of detaching the previous entity from Doctrine

The nail in the coffin is that there is currently no API available to do this from inside an entity. This is why I don't recommend pursuing immutable entities with the current Doctrine version at the time of writing (that is, version 2.5.5).

Anyway, avoiding creating setters on your entities will already be a huge step in the direction of a referentially transparent codebase. It will also help you a lot with keeping your business logic all in one place with no possibility of entities being left in an invalid state.

Why immutable entities?

Instead of a long speech, let's demonstrate this using a simple example. Doctrine uses instances of the `DateTime` class for anything related to dates and times. Since the `DateTime` class is mutable, this can lead to issues not at all easy to pinpoint:

```php
<?php

$date = $post->getPublicationDate();

// for any reason you modify the date
$date->modify('+14 days');

var_dump($post->getPublicationDate() == $date);
// bool(true)

$entityManager->persist($post);
$entityManager->flush();
// nothing changes in the database :(
```

The first issue is that you have a reference to the same object stored inside the entity. This means that if, for any reason, you change it, the date will also change inside the post. This might be what you want, but there is absolutely no doubt that this is a side-effect. Especially if you return the `$date` variable to a potential caller. How is it supposed to know that modifying the date will lead to modifying an entity?

The second issue is more problematic. As Doctrine uses the object identity and not its value to determine whether something has changed, it will not know the date is now different and saving the post back into the database will amount to nothing.

There is a package available on GitHub (https://github.com/VasekPurchart/Doctrine-Date-Time-Immutable-Types) for this particular issue, but, any time you use mutable instances instead of embeddables or any other kind of value objects, you can run into similar problems. Do yourself a favor and use immutability whenever possible.

Symfony ParamConverter

We discussed modifying already instantiated entities and persisting them back to the database. But what about getting them in the first place? The `SensioFrameworkExtraBundle` framework contains a nice little annotation called `@ParamConverter`, which allows us to let the

framework do the job and keep the side-effect of getting entities from the database outside our codebase.

Here is a small example so that you understand how to use this annotation (if you want to know more, you can read the official documentation on the Symfony website at http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/a

```
<?php

use
Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter
;

class PostController extends Controller
{
    /**
     * @Route("/blog/{id}")
     * @ParamConverter("post", class="SensioBlogBundle:Post")
     */
    public function showAction(Post $post)
    {
        // do something here
    }
}
```

Using the information from the route alongside the defined parameters conversion, the framework is able to give you the `Post` instance directly or generate a `404 error` if it isn't able to find it.

Using the annotation, your method does not need to perform database access anymore as it will receive the data directly. We might argue that impure code exists somewhere else and it would be true, but Symfony is not supposed to be a pure codebase anyway. We were able to push the impurity out of our own code and that is what matters to us.

In this particular case, the type-hint would have been sufficient in relation to the route. The `ParamConverter` annotation will automatically enter an action when a function signature references an entity class. There is no harm in keeping the annotation if you find it clearer, or you can decide to only use it in more complex cases.

There will obviously be circumstances where this mechanism is not powerful enough. I know some bundles provide similar features with more flexibility; you might be able to find one that suits your needs. And if nothing else works, you can still perform the query yourself or use an IO monad to do it for you.

Maybe there is an entity

Doctrine can be quite easily adapted to return instances of the Collection and Maybe monads. The first step is to create a new repository:

```
<?php

use Widmogrod\Monad\Maybe as m;
use Widmogrod\Monad\Collection;

class FunctionalEntityRepository extends EntityRepository
{
    public function find($id, $lockMode = null, $lockVersion =
null)
    {
        return m\maybeNull(parent::find($id, $lockMode,
$lockVersion));
    }

    public function findOneBy(array $criteria, array $orderBy =
null)
    {
        return m\maybeNull(parent::findOneBy($criteria,
$orderBy));
    }

    public function findBy(array $criteria, array $orderBy =
null,  $limit = null, $offset = null)
    {
        return Collection::of(parent::findBy($criteria,
$orderBy,  $limit, $offset));
    }

    public function findAll()
    {
        return Collection::of(parent::findAll());
    }
```

```
}
```

Then, you need to configure Symfony to use this new class as the default repository; this can be easily done by adding the following key to your YAML configuration:

```
doctrine:
  orm:
    entity_managers:
      default_em:
        default_repository_class:
MyBundly\MyNamespace\FunctionalEntityRepository
```

If you aren't using Symfony, you can use the `setDefaultRepositoryClassName` method on a Doctrine `Configuration` class instance to achieve the same effect.

With everything we have discussed regarding Doctrine, you won't be able to have a purely functional code when the database is involved but you are prepared enough to reap most of the benefits.

Organizing your business logic

The official Symfony best practices contain some advice on how and where to write your business logic. We will expand upon them a bit to facilitate writing functional code.

The first advice is to avoid writing any logic in parts related to the framework itself: routing and the controllers. Those should be as straightforward as possible. It is a good idea to follow this advice because this way it doesn't matter as much if you are forced to do some database access in the controller.

I recommend you to do everything database-related inside the controller itself, so anything with side-effects is segregated there and your business logic can follow proper functional techniques.

Also, instead of using the service container, you should inject only the dependencies you need both in your controller and your services. This

will greatly reduce the cognitive burden as the signature of your methods and constructors will be enough to determine the dependencies of your business logic.

I would also recommend you avoid using setter injection as calling the setter will modify the state of your service, thus breaking immutability and also leading to possible issues if the setter is called multiple times.

By taking the decision to limit your side effects to the controller, you can concentrate on writing functional code in your entities and services. This will make the entities and services easy to reason about and to test. Then, since the controller should contain no logic of its own, you can test the final parts using integration and functional tests and quickly gain confidence in your application.

Flash messages, sessions, and other APIs with side-effects

Flash messages are often used to communicate information to your users in a non-obtrusive way. The API proposed by Symfony to manage them is sadly not referentially transparent as you need to call a method on the controller that will add the message to a queue. The same is true for session data management.

This issue could be solved by integrating them somehow inside the `Response` object. This would, however, need to be done at the framework level. Such changes would either need to be incorporated upstream or would require a lot of maintenance.

One solution is to leverage the Writer or State monads in your services to hold that information and then persist them in the controller as we already decided to use it for side effects related to the database.

However, I don't recommend using the IO monad as it will prove complicated without some kind of support at the framework level, especially since PHP lacks a good alternative to the *do notation* annotation available in Haskell. It will only complicate your code

without real benefits.

The `Form` API is another example of instances with a lot of inner state and impure methods. It is, however, declarative enough for this fact to not pose a lot of issues. The fact that you can abstract the form creation into its own class also helps a lot with being able to consider it side-effect free.

I strongly suggest you create `Form` types whenever you can and treat the resulting `Form` instances as immutable objects as much as possible.

Closing words

Although designed with a strong emphasis on object-oriented design, Symfony offers us a good foundation to write functional code. Some concessions are needed, especially when it comes to database access and non-functional APIs provided by the framework, but the code you write yourself can be pretty much functional from start to end except in the controller itself.

One downside of using a functional approach might be that you find yourself creating more services and classes to segregate all side-effects in a single part of your request life cycle.

This has, however, the benefit of having a clearly decoupled codebase that might even be reused outside of Symfony if given the proper care. You will also be able to test each part more easily in separation.

There is also nothing preventing you from applying functional techniques gradually in some parts and services of your application. By being able to slowly migrate your code to something more functional, you are able to apply the techniques immediately and it also helps with the learning curve of other people. You can use the following resources to better understanding the application of functional techniques:

- https://vimeo.com/177154259
- http://www.slideshare.net/boerdedavid/being-functional-in-php

Laravel

As we have already discussed, the `collection` API from Laravel is a good example of an immutable data structure with nice functional methods on top of it. The database layer returning instances of the `Collection` class really helps with streamlining its use.

However, the **Facade** pattern proposed by the framework is a no-go if you want to keep your functions pure. As soon as you use a façade, you end up using an external dependency that is not declared in your function signature.

Whatever your take on this pattern is, if you want to write referentially transparent code, you have to get rid of them. Luckily, Laravel provides helper functions for most common tasks and a way to access the container that is baking the facade. As we will see, it is thus not that difficult to use something different.

Since Laravel is also a framework based on object-oriented principles and the MVC pattern, all general advice from the Symfony sections also applies, especially the ones about decoupling the various parts and trying to segregate side effects in a unique place for each request, often the controller.

The way of achieving that may differ a bit but not too much; this is why I encourage you to read the previous section if you haven't done so already, as that advice will not be repeated here.

Database results

As already said, Laravel has a really great immutable collection implementation. All database queries returning multiple entities will return an instance of it. There's a really good book presenting in detail all the ways you can leverage its features. You can find it alongside screencasts and other tutorials on the author's website at https://adamwathan.me/refactoring-to-collections/. The collection might

be immutable, but the objects in it are not. Since the Laravel ORM, Eloquent, is really different from Doctrine, it is, however, possible for us to make them so. Instead of using a `Repository` pattern and the `UnitOfWork` pattern, you simply have methods on a `Model` class that you have to extend, the `ActiveRecord` pattern. This means it is possible to implement your methods in a way that makes your entities immutable without having the issues we encountered about the internal state of Doctrine itself:

```
<?php

use Illuminate\Database\Eloquent\Model;

class BlogPost extends Model
{
    private $status;
    private $publicationDate;

    public function publish(DateTime $d)
    {
        $new = clone $this;

        $new->status = 'published';
        $new->publicationDate = $d;
        return $new;
    }
}
```

As long as you don't modify the field used for the primary key, any call to the `save` method of your object will update the current row in the database. You might, however, want to add a `__clone` method on your object if you have non-scalar properties. PHP will only perform a shallow copy by default, meaning all references will stay the same. It might be what you want, but you need to make sure of it.

If you want to enforce the immutability of certain properties, there is a package available on GitHub (https://github.com/davidmpeace/immutability) to do exactly that. It should not be necessary if you are rigorous, but it might be a nice feature to have in a legacy codebase with both traditional and functional parts.

Using Maybe

As with Doctrine, it is also possible to return an instance of `Maybe` instead of `null`. As the structure is a bit different, we first need to create a new `Builder` class:

```
<?php

use Illuminate\Database\Eloquent\Builder as BaseBuilder;
use Widmogrod\Monad\Maybe as m;

class FunctionalBuilder extends BaseBuilder
{
    public function first($columns = array('*'))
    {
        return m\maybeNull(parent::first($columns));
    }

    public function firstOrFail($columns = array('*'))
    {
        return $this->first($columns)->orElse(function() {
            throw (new ModelNotFoundException)-
>setModel(get_class($this->model));
        });
    }

    public function findOrFail($id, $columns = array('*'))
    {
        return $this->find($id, $columns)->orElse(function() {
            throw (new ModelNotFoundException)-
>setModel(get_class($this->model));         });
    }

    public function pluck($column)
    {
        return $this->first([$column])->map(function($result) {
            return $result->{$column};
        });
    }
}
```

Multiple methods were redefined because of the use of the `first` function which now returns an instance of the `Maybe` type instead of null. There is, however, no need to return instances of the `Collection` as the

Laravel version is already a nice implementation even if it is not a monad.

Now we need our own `Model` class, which will use our `FunctionalBuilder` method:

```
{
        return $this->find($id, $columns)->orElse(function() {
            throw (new ModelNotFoundException)-
>setModel(get_class($this->model));
        });
    }

    public function pluck($column)
    {
        return $this->first([$column])->map(function($result) {
            return $result->{$column};
        });
    }
}
```

Those two new classes should work in most cases, but as we are redefining methods used by the framework itself, it is possible you might run into trouble. If this is the case, I would love to hear from you so the implementation can be refined to avoid the issue.

You might also want to modify Collection monad so that an instance of the Maybe monad is returned instead of `null` in the various methods where it is appropriate. This will, however, require many more modifications than what we have done so far. As far as I know, there is currently no package providing this feature.

Getting rid of facades

The concept of Facades might be useful to reduce the learning curve for newcomers and it might facilitate the use of the services proposed by Laravel. But whatever your opinion of them is, they are not functional for one bit because they introduce side-causes as soon as you use them.

It is fairly easy to get rid of them by injecting the dependencies inside

your controllers and services, as is common practice in the Symfony world. Besides allowing you to write functional code, there is a hidden benefit to stopping using facades-your code will be less tied to Laravel and thus you might be able to reuse it more.

Laravel offers a feature called **automatic injection** which will allow you to very easily get hold of the various components available through a Facade. It uses the type-hints to automatically inject the wanted dependency upon the class instantiation. It works in a multitude of contexts-controllers, event listeners, and middleware, for example.

Getting an instance of the `UserRepository` class is as simple as the following:

```
<?php

namespace App\Http\Controllers;

use App\Users\Repository as UserRepository;

class UserController extends Controller
{
    protected $users;

    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }
}
```

The type-hint to use can be easily found by referring to the table available in the documentation at https://laravel.com/docs/5.3/facades#facade-class-reference.

This nifty mechanism doesn't really decouple you from the framework, however, as you need to use the correct type-hint. Another way to manually inject your dependencies via the `bootstrap/start.php` of your project is described in the article at http://programmingarehard.com/2014/01/11/stop-using-facades.html/.

HTTP request

As with Symfony, it is really easy to use the interfaces defined in PSR-7 instead of the one from the framework. Laravel uses the same bridge as Symfony to perform the transformation. You only need to install two packages using Composer:

```
composer require symfony/psr-http-message-bridge
composer require zendframework/zend-diactoros
```

And then it is enough to use the `ServerRequestInterface` class as the type-hint instead of `Request` method when you want to obtain an instance of the PSR-7 version. Laravel will take care of converting the `Response` method to its own format itself if your controller action returns the PSR-7 version.

Closing words

The implementation decisions taken by the Laravel core developers go both ways. Some of them, such as the immutable collection implementation, are great when it comes to functional programming. Others, such as the use of Facades, make our life a bit more difficult.

It is, however, fairly simple to transform our code in order to use a more functional approach. The only difficulty you might encounter is when reading the documentation or tutorials, which will often describe patterns and practices we are trying to avoid.

All in all, Laravel is as good as Symfony when it comes to writing functional code. The book about its collection implementation mentioned earlier is also a great way to learn how to use some functional techniques in relation to the Collection monad implementation. As far as I know, this kind of resource does not exist for Symfony.

Drupal

Drupal modules until version 7 relied on hooks to perform operations. Drupal hooks are functions following a certain naming pattern that get called by Drupal when various events occur when responding to a request to modify various aspects of the generated web page.

In an ideal world, all hooks would receive all needed information to perform their work and the way to modify something would be using the return value. This is mostly true for some parts of the module API. Sadly, there are some functions that receive parameters passed by reference, such as the `hook_block_list_alter` function. Also, you sometimes need to access global variables, for example, to get hold of the current language.

Drupal 8 moved to a class-based approach. The content should now be created inside a controller in order to be closer to Symfony terminology. The reason is that this new version now uses some of Symfony's core components. This does not mean it is impossible to use functional programing anymore, just that things are a bit different.

It is not the role of this book to explain in detail what changed from version 7 to version 8. There are plenty of tutorials out there doing a great job of that. Most of what will be presented here is general enough to be useful and true for both Drupal versions.

Database access

In Drupal 7, there are multiple functions you can use to perform database queries and access the results. Usually, you start with a `db_query` function, which returns a result object with various methods to inspect and process the data. The preferred Drupal 8 way is not to get a database connection injected inside your module or service and use it in a more object-oriented way.

This change does not really affect us as, in both cases, it is not possible

to query the database in a referentially transparent way. Also, people usually don't use an ORM with Drupal; most requests to the database are done using SQL directly.

This is why we won't linger on this subject besides repeating that it is important to segregate database access as much as possible so that the rest of your code can be functional.

Dealing with hooks requiring side effects

The minimal Drupal module is composed of two files, the `info` file and the `module` file. The format of the `info` file changed from an ad hoc text file in Drupal 7 to a YAML file in Drupal 8, but the file still contains information about the module. The `module` file is the main PHP file of the module.

As we saw, some hooks require side-effects to perform their job and there is hardly a way around that fact. What I can recommend is to use the module file as the one that will hold all non-strictly functional code and putting all computations somewhere else.

In the case of Drupal 8, some controller methods might also need to have side-effects. In this case, I'll give you the same advice as for Laravel and Symfony: keep those in the controller and use external services/helpers to perform referentially transparent computation.

How will we do that for the `hook_block_list_alter` function we spoke about earlier? First of all, this applies only for Drupal 7 as, in the next version, blocks are managed through a class, which kind of solves the issue of referential transparency for this particular case.

My advice is simply to create a second PHP file in your module, containing only pure functions. For example, this file could contain a `new_blocks` function taking the current blocks and language as its only parameters.

Then, in the module file, you can do the following:

```
function my_module_block_list_alter(&$blocks) {
    global $language;

    $blocks = new_blocks($blocks, $language);
}
```

This function clearly has both side causes and side-effects; there is not much we can do about that. However, the `new_blocks` function can be pure, which means you can easily reason about it and test it like we saw in the previous chapters.

This method can be applied to nearly anything. As soon as you get side causes or side-effects, perform those in the module file and then use a different file to hold your pure functions, which will do the necessary processing and computations. If you are using Drupal 8, instead of using the `module` file, you can use the controller as we already discussed for Symfony and Laravel.

Hook orders

The beauty of Drupal comes from all the various modules available. This is so true that some people came up with a variation of one of the Apple marketing slogans: *There's a module for that!*. This comes with some difficulties, however. Not all modules are equal when it comes to quality, and you usually end up with a bunch of them for any given application.

The corollary is that the information your own hooks receive may already have been altered by previous modules. Say you are writing a module to reorder some blocks on the page; it is perfectly possible that some of the blocks you expect to be present were already removed. Or maybe a key you are using in an associative array is already registered or will be overwritten.

This can lead to some issues that are a bit hard to pinpoint exactly. Since your functions will be pure, it is, however, relatively easy to detect the fact that it comes from something else by explicitly adding a test to ensure it works as expected for a given dataset.

A good piece of advice concerning this problem is to not make assumptions about what may or may not be present in anything you receive from Drupal. Always apply some kind of check to ensure that the data you receive is correctly structured and present.

Closing words

Probably for historical reasons, some Drupal hooks need to have side-effects in order to perform their duty. Also, not all information is passed as parameters to them, requiring us to access the global scope to get them. This fact requires that we find workarounds to keep as much code as possible pure.

By introducing a more object-oriented approach coupled with service injection, Drupal 8 makes things a bit easier. Quite on a par with the experience we can have with Symfony or Laravel, but things are still not perfect.

If you are rigorous in separating your impure functions from your pure ones in at least two files, your experience writing functional code can be really good. It might seem cumbersome to always create two functions to implement one hook, but it is the price you have to pay if you want pure functions and, in my opinion, it is worth it.

As we discussed, you can still experience issues on the final page rendering even if your pure code is thoroughly tested due to the order some hooks are called in, but those are usually easier to spot if you can have confidence in your functions.

The functional developer experience with Drupal is not perfect, but it gets close. You will have to make some concessions, but you can bind those to a few files to limit their impact on the rest of your code.

WordPress

WordPress also has a hook system in place, although different from Drupal's one. Instead of creating functions with a certain name, you register functions to a certain hook. Sadly, most of those hooks need to have side-effects by definition. For example, we can do this with the help of the `wp_footer` hook:

> *This hook provides no parameters. You use this hook by having your function echo output to the browser, or by having it perform background tasks. Your functions shouldn't return, and shouldn't take any parameters.*

No return value, no parameters; we are forced to create a function with side-effects. This means that you will have to create wrapper functions around your code even more than what we just demonstrated with Drupal.

Luckily, WordPress also allows you to have multiple files per plugin. Thus, the recommendation is the same-put all your impure code in the main file. Get all the information you need from the global context and perform any kind of operation with side-effects there. Once you have everything you need, call your pure functions for processing and computation.

Some WordPress tutorials present object-oriented programming as the next evolution for developers when they have mastered a more procedural way of writing plugins. If you plan on using functional techniques, it does not matter. You can organize your code using only functions or you can group them in classes. I would advise you to stick to the method you are more at ease with.

Database access

There are basically two ways to access databases in a WordPress plugin. You can use the `WP_Query` method directly and its object-oriented

interface. Or you can use helper functions such as `get_posts` and `get_pages`.

You might have heard or read somewhere that it is best to use the `WP_Query` function when writing your plugin using classes, and helpers when using functions. From a functional standpoint, it doesn't matter. None of them are in any way referentially transparent. You can use whichever you like best or suits your needs better.

There is nothing much to be said about database access in a WordPress codebase. The issue is the same as with the other frameworks-there is currently no way to perform them in a pure way.

This being said, the advice stays the same-try to segregate any code with side-causes and side-effects to a single file of your plugin and then delegate the computation and processing to pure functions living somewhere else.

Benefits of a functional approach

I said in the introduction of this part that it does not matter whether you use functions or objects to organize your code. This is only partially true. WordPress lacks all the injection features of frameworks such as Symfony or Laravel. This means that, if you are using objects, you will encounter difficulty in sharing instances around.

It isn't really an issue if your object is only used to hold pure methods that don't use any kind of internal state, but as we saw, it is sometimes necessary to make concessions. If you need to share an instance with such a state, your only solution is to make it available globally. The problem with such a variable is that it can be reassigned to something else, causing issues later on.

On the contrary, a function is available from anywhere and you cannot redefine it. This leads to more robust code as you limit the possibilities of side-effects.

Closing words

Drupal's first release dates back to 2000, making it the oldest tool presented here. WordPress was born in 2003. Drupal, however, has been rewritten since then, whereas the WordPress codebase was mostly extended without a full rewrite.

Why am I telling you this? Because most of the issues you encounter when trying to write functional code in WordPress will be related to its legacy codebase. The way software was written in 2000 is a bit different than the best practices we expect now.

A lot of work was done to modernize WordPress, but there is only so much you can do. Especially when the focus is not to make the framework functional-developer friendly. It is nonetheless possible to write functional code if you are willing to jump through some hoops to isolate the parts with side-effects.

WordPress being mostly based on hooks, most of the API is composed of functions. Some of them referentially transparent, others not at all. It will take you some rigor to cleanly isolate those from the rest of your code. The benefits are always the same:

- Reduced cognitive burden
- Facilitated code reuse
- Easier testing

The downside will be that your main plugin file will mostly be composed of really small functions that only serves as wrapper around the impure functions of the WordPress API and then call your referentially transparent functions.

If the name of your wrappers are close enough to their Wordpress counterpart, reading your code and navigating through it should be pretty easy for anyone having Wordpress knowledge. In the end, it is still a good idea to write as much functional code you are capable of.

Summary

As we already discussed earlier, there is no mainstream framework having a functional approach at its core. In this chapter, we tried to see how the techniques we've learned can be applied more or less successfully to some available frameworks and CMSes.

As we saw, it is always possible to use functional programming at least at some level. Sadly, depending on the framework, you will have to create non referentially transparent code at some point.

As I said in the introduction, I am not an expert in all the libraries we discussed in this chapter, so take everything with a grain of salt. A more seasoned developer might do things differently. The examples provide, however, a good starting point for anyone wanting to try functional programming.

Also, when doing so, remember that it is first and foremost a way of thinking. It will be the way you approach the issue at hand that is the most important. If, at some point, you need to create non-pure code to accommodate an external dependencies or the framework you are using, so be it. This won't change the benefits you can get for the functional code you've written.

Now that we've seen how to use functional programming in an existing framework or a legacy codebase, the next chapter will cover designing a whole application using a paradigm known as Functional Reactive Programming or FRP.

Chapter 11. Designing a Functional Application

Creating a whole application respecting the precepts of functional programming might seem like an impossible task. How can you write any meaningful software if you cannot have any side-effects? In order to perform any kind of computation, you will need at least some inputs and display results.

Functional languages have various mechanisms to circumvent those limitations. We will quickly present some of them so that you can have a better idea about how an application can be written in a purely functional way.

We will then learn more in depth about a paradigm called **Functional Reactive Programming** (**FRP**), as a way to design an application having a user interface. We will lay the foundation for using this technique in PHP to see if it is possible to use it to write a complete application.

In this chapter, you will learn about the following topics:

- Writing a complete application in a purely functional language
- Functional Reactive Programming
- Designing a PHP application using FRP

Architecture of a purely functional application

Applications are like functions. If you have an application without any input, its outcome will always be the same. You have the possibility of modifying some values in the source code and recompiling your software to change its result, but this is contrary to the main reason we write applications in the first place.

This is why you need a way to feed data to an application in order for it to perform any kind of meaningful computation. Those inputs can be of multiple types:

- Command-line parameters
- File content
- Database content
- Fields in a graphical interface
- Third-party services
- Network request

Out of all those, only the first one could be considered as not breaking the referential transparency of our whole application. If you consider your application as one big function, feeding data on the command line could be considered as its parameters, thus keeping everything pure. All other kinds of input are de facto impure, since two subsequent retrievals of the data could lead to different values.

The canonical Haskell way to solve this issue is to use the **IO monad**. Instead of performing its operations immediately, the IO monad stores all steps in a queue. If you name this IO operation `main`, Haskell will know it has to run it when the compiled program is executed.

Obviously, the application itself is not pure anymore if you perform any kind of IO operation inside the monad. However, the code itself can be written in a referentially transparent way. It's the Haskell runtime that will perform all impure operations when the IO monad is run and will then pass the various obtained values around. Using this trick, you can write pure functional code with all the benefits it brings and still perform IO operations.

This approach is usable in Haskell because you can use monad transformers to combine multiple monads. The do notation also helps a lot by writing code encapsulated in the IO monad without all the overhead associated with it. For example, here is a small program reading lines in the Terminal and printing them with the words in reverse order:

```
main = do
  line <- getLine
  if null line
    then return ()
  else do
    putStrLn $ reverseWords line
    main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

It reads mostly like any imperative source code performing the same task. PHP lacks the syntactic sugar and there exists no implementation of monad transformers, so it is quite hard to do this. This is why we make compromises as discussed in the previous chapter, or we need some other approach, as we will see in the following section.

The idea at play can be generalized. Any impure function can be split into two functions, one pure and one encapsulating the side causes and side effects. This is exactly what we were referring to in the previous chapter when we stated that most impure functions should be contained in the MVC application's controller.

If you have an impure function `f` taking `A` as a parameter and returning `B`, you can create the following two functions:

- A pure function `g`, which takes `A` and returns `D` parameter. The parameter `D` is being a description of the IO operations that need to be performed.
- An impure function `h` taking `D` and performing the described operations like an interpreter would do.

If we take the example of a Haskell application, the Haskell runtime itself would be our impure `h` function. If our source were to return an instance of the IO monad, as our example on just above is doing, it would be used as the `D` parameter and the side-effects would be interpreted.

If you are writing a web application using the **Symfony** framework, we could consider the framework as the the impure `h` function and the `D`

parameter would be the result of executing your controller for example. Another possibility would be to add our custom impure wrapper around our functional code.

The main idea is to reduce the number of functions like `h` to the minimum. Haskell forces you to have only one such function and it's even hidden inside the runtime. If you are using PHP, it's up to you to enforce this rule as effectively as possible.

This concept of having a description of the computations and an interpreter to perform them is central to a lot of the more advanced techniques in the functional world. It is also quite important in computer programming as a whole. If we take a bit of distance, we can see the following:

- The description is like an Abstract Syntax Tree (**AST**)
- The interpreter takes the AST and runs it

This is how most modern compilers work, first they parse the source code to transform it in an AST and then interpret it to create the binary file. You will also find the same pattern again and again in most complex applications.

An advanced construct using this structure is the *free monad*. This monad is currently a hot topic in the functional world and its usage is growing fast. We are missing quite a bit of theory to approach the topic here, but if you are interested you will surely find a lot of information on the Internet, for example, http://underscore.io/blog/posts/2015/04/14/free-monads-are-simple.html.

However, this pattern is problematic when you accept user interaction during the lifecycle of the application. Since the main idea is to delay the execution of effective computations by describing them instead, you cannot perform part of the computation to display a user interface and then react to user input. This is one of the issues that FRP tries to solve.

From Functional Reactive Animation to Functional Reactive Programming

As is often the case when it comes to functional programming, the foundations behind the subject at hand date back a bit. In 1997, Conal Elliott and Paul Hudak published a paper called *Functional Reactive Animation, or Fran*.

The main goal of Fran is to allow the modeling of animations with two concepts called **behaviors** and **events**. Behaviors are values based on the current time, and events are conditions based on external or internal stimuli. Those two notions allow us to represent any kind of animation at any point in time although the animation itself is continuous.

Instead of directly creating the representation of your animation as it is usually the case, you describe it using behaviors and events. The interpretation, and thus representation, is then left to the underlying implementation. This is similar to what we just described. As events such as keyboard inputs or mouse clicks can be encoded inside Fran, the model you are creating allows for a pure functional application to respond to external inputs like those.

Reactive programming

Before we go any further, let's speak a bit about what *reactive* means in the programming world. It's an idea that has gotten quite a lot of traction in the last few years.

First, there is the *Reactive Manifesto* (http://www.reactivemanifesto.org/), which presents a list of properties that are really interesting to have for any software. Those properties are: responsiveness, resilience, elasticity, and being message-driven.

The Wikipedia (https://en.wikipedia.org/wiki/Reactive_programming) definition states something quite different:

> *In computing, reactive programming is a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.*

An example is then given of the expression $a = b + c$, where the value of `a` is automatically updated when any of `b` or `c` changes.

The JavaScript world is in effervescence about the idea, with libraries such as `Bacon.js` or `RxJS`. The core idea shared by all of those libraries revolves around events or event-streams.

As we can see, there are multiple definitions of what reactive programming is. Sadly, none of them really quite match what we just learned about Fran. As it has been floating around at least since the seventies, the definition that we will keep for the remainder of this chapter is the academic one, which can be found on Wikipedia.

I am not saying the other ones are invalid, just that we need to have a common ground here. Also, next time you speak of reactive programming with someone, first make sure you are on the same page concerning what the topic is.

As a final example of reactive programming, let's consider the following piece of code:

```
<?php

$a = 10;
$b = 5;
$c = $a + $b;

echo $c;
// 15
```

```
$a = 23;
echo $c;
```

In a traditional imperative language, the last line will still display 15. However, if our application were to follow the rules set by reactive programming, the new value of `$a` would also affect the value of `$c` and the program would display 28.

Functional Reactive Programming

As you can probably guess, values changing over time when other changes are made are far from being referentially transparent. Also, the concept of variables is completely missing from some functional languages. How can we reconcile reactive and functional programming?

The core idea is to make the time component and the previous events parameters of your functions when they need them. This is exactly what Fran proposed with behaviors and events. Both time and events are usually proposed for consumption as a stream. Using functional mapping and filtering, you are able to decide which events on the stream interest you.

Your functions take one or multiple inputs from this stream alongside the current state of the application. They must then return the new state of the application. The runtime will take care of calling the various registered functions when the events happen.

You might have the impression that it is similar to event-driven programming. In a way it is, but there is a big difference. In a traditional event-driven application, events are triggered, but the return value of the handlers is often of no importance; they need to have side-effects to perform something.

When doing FRP, the runtime takes care of orchestrating all registered handlers. Keeping the current state of the application, passing it to each handler, and updating it with their results. This allows for the functions to be pure.

Another programming paradigm that might be a bit closer than event-driven programming is the actor model. I won't describe it here as it will be out of scope for this book, but for people aware of it, I will just say that there are two main differences:

- As you have pure functions instead of actors, you cannot have a private state influencing the way you respond to a given message, or event
- The runtime manages the event stream; there is no way for the handlers to send new messages to other parts of the application

Time traveling

FRP also has another benefit. If you record the sequence of events leading to a particular application state, you can replay them. Where it gets better is that you can implement what is called a **time traveling debugger**. Since your application is using pure functions, you can go back to any point in time and get the exact same state as you've had before.

This kind of debugger also allows you to replay any number of steps back and forth until you can pinpoint exactly what is happening. Also, you can make changes to your code and play the same events to see how your modification affected your software.

If you want to see such a debugger in action, you can head over to the one proposed by the **Elm** language, specifically their online version with a naive implementation of the Mario platform game (http://debug.elm-lang.org/edit/Mario.elm).

The Elm debugger is probably one of the first of its kind. Although similar ideas have been implemented in traditional languages, the very nature of imperative programming requires us to record a lot more than just the stream of events. This is why it is a really costly operation, slowing down the execution of the program a lot.

You also need to restart the program from the beginning in order to be sure to attain the same state. However, in a pure application, you can do

this in a more straightforward way. Implementations more akin to the one found in Elm are now being created, for example, for the **React** JavaScript library.

Disclaimer

There are FRP and FRP, but instead of paraphrasing the creator of the idea, let me instead quote him:

> *Over the last few years, something about FRP has generated a lot of interest among programmers, inspiring several so-called "FRP" systems implemented in various programming languages. Most of these systems, however, lack both of FRP's fundamental properties.*

You can see the full text alongside related slides and video on GitHub (https://github.com/conal/talk-2015-essence-and-origins-of-frp).

As often, there is some kind of divergence between the academical world and the usage people make of the research results. I won't dwell on the details as this is supposed to be only an introductory chapter. However, it is important that you are aware of this fact.

The main point of contention is the fact that FRP is about *continuous time*, whereas most implementations consider only *discrete events or values*. If you want to know more about those differences, I strongly suggest you watch the previously linked video, available on the GitHub repository of Elliot Conal, the creator of Fran and FRP.

Going further

There are a lot of other things to say about Functional Reactive Programming. In fact, whole books are dedicated to the subject. This is, however, just an introduction so we will stop there. If you want a general approach to the topic not tied to a specific language, I can recommend the newly published *Functional Reactive Programming* by Stephen Blackheath and Anthony Jones.

On the implementation side, the **ReactiveX** project tries to federate libraries available on multiple projects. You can find more information on the official website at http://reactivex.io/. At the time of writing, the following languages are covered; Java, Swift, Python, PHP, Scala, JavaScript, Ruby, Clojure, Rust, Go, C#, C++, and Lua.

As stated in the previous disclaimer, and the introduction on the ReactiveX website, there is currently a conflation of the academic concept of FRP as extended from the original Fran paper and what today's programmer means by the term. Both the aforementioned book and the ReactiveX libraries speak about the latter rather than the original meaning. It does not mean those are bad ideas, quite the contrary; it is just that it is not real FRP.

ReactiveX primer

The `Rx*` libraries made the choice of implementing the functional reactive paradigm by extending the classical Observer pattern into the `Observable` model. For a given stream of values, represented by an instance of the `Observable` model, you can define up to three different handlers:

- The `onNext` handler will be called each time there is a new value available
- The `onError` handler will be called when an exception arises
- The `onCompleted` handler will be called when the stream is closed

This approach makes it easy to work with multiple asynchronous events without having to write complex boilerplate code to manage dependencies between them. Contrary to the traditional Observer pattern, the ability to signal the end of the stream and errors is added to reconcile the interface with iterables.

ReactiveX also defines a bunch of operators to both manipulate observables and their values. There are helper methods to create various kinds of streams, from ranges to arrays, passing by infinitively repeating values and timed release events.

You can also manipulate the stream itself by mapping functions to each emitted value, grouping them into new observables or into arrays of values. You can also filter the values, skip or take a certain number of them, limit the number of emissions for a certain amount of time, and suppress duplicates.

The documentation (http://reactivex.io/documentation/operators.html) has a complete list of what manipulations are available, along with a nice decision tree to decide which one to use based on the context.

RxPHP

Before we start having a look at some examples of RxPHP, I would like to point out that Packt Publishing also published a complete book, *PHP Reactive Programming*, about the topic. You can find more information on their website at https://www.packtpub.com/web-development/php-reactive-programming. This is why we will only explore some basic examples to give you a feel for how using the library might look. If the subject is of interest to you, I strongly suggest you read the dedicated book.

After this very brief introduction to ReactiveX, let's see how it can be used. First we will need to install the required library. We will use a small wrapper around ReachPHP's stream library to make it usable with RxPHP so we can demonstrate accessing files on disk. The following `composer` invocation should install all needed dependencies:

```
composer require rx/stream
```

Now that the library is installed, you can parse data from any PHP stream. For example, a CSV file:

```
<?php

use \Rx\React\FromFileObservable;
use \Rx\Observer\CallbackObserver;

$data = new FromFileObservable("11-example.csv");

$data = $data
    ->cut()
    ->map('str_getcsv')
    ->map(function (array $row) { return $row; });

$data->subscribe(new CallbackObserver(
    function ($data) { echo $data[0]."\n"; },
    function ($e) { echo "error\n"; },
    function () { echo "done\n"; }
));
```

We first create a stream Observable for the file we want to read, then we apply some transformation: separating the input by line, parsing the CSV string in an array, and applying any other data processing you might want. As you can infer from the fact that we reassign the result to `$data` variable, the operation is not made in place, but a new instance is returned each time.

Then, we can subscribe handlers to our stream. In our case, we simply print the first row of each element. Not really functional, but effective enough for a small example.

If you are using **PostgreSQL**, a package allowing you to use Rx to access your database exists. You can use it to retrieve data using a stream. You can install it using the `composer` invocation:

```
composer require voryx/pgasync
```

Creating queries is fairly easy. It is a matter of creating a client with the connection credentials and then calling one of the methods on it to create an Observable instance on which you can subscribe:

```
<?php

$client = new PgAsync\Client([ "user" => "user", "database" =>
"db" ]);

$client->query('SELECT * FROM my_table')->subscribe(new
CallbackObserver(
    function ($row) { },
    function ($e) { },
    function () { }
));
```

Here is a final example demonstrating some of the more advanced filtering and transformation possibilities offered by Rx on the streams themselves. Try to guess what the output will be before running it:

```
<?php

use \React\EventLoop\StreamSelectLoop;
use \Rx\Observable;
```

```
use \Rx\Scheduler\EventLoopScheduler;

// Those are needed in order to create a timed interval
$loop = new StreamSelectLoop();
$scheduler  = new EventLoopScheduler($loop);

// This will emit an infinite sequence of growing integer every
50ms.
$source = Observable::interval(50, $scheduler);

$first = $source
    ->throttle(150, $scheduler) // do not emit more than one
item  per 150ms
    ->filter(function($i) { return $i % 2 == 0; }) // keep only
odd numbers
    ->bufferWithCount(3) // buffer 3 items together before
emitting them
    ->take(3); // take the 10 first items only

$second = $source
    ->throttle(150, $scheduler)
    ->take(10);

$first->merge($second) // merge both observable
    ->subscribe(new CallbackObserver(
        function ($i) { var_dump($i); },
        function ($e) { },
        function () { }
    ));

$loop->run();
```

If you try to run this last piece of code, you need to have the development version of RxPHP, as `throttle` was only recently implemented. If your minimum stability parameter is set to the `dev` edition, you can install it using:

```
composer require reactivex/rxphp:dev-master
```

Achieving referential transparency

As the examples demonstrated, creating streams and subscribing to them is fairly trivial. It is also quite easy to imagine how we can factorize handlers in a way that will allow reuse between multiple observable

instances.

The issue that Rx does not solve for us, however, is the application architecture needed to achieve referential transparency as much as possible. It does not suffice creating a new database query as an Observable to be pure.

The advice I can give you is the same you already heard in the last chapter that is to try to segregate all impure code in one place. In our case, this can be achieved by creating all streams in a unique file, like your `index.php` file, for example, and declaring the handlers somewhere else.

The various handlers can be tested in isolation and you can quickly build up confidence about them as they will be referentially transparent. The integration and functional tests will then take care of testing the streams themselves and the application as a whole.

If you try to use Rx in an existing framework, you can declare streams in your controllers and keep the handlers separated the same way as described previously.

Summary

Functional reactive programming allows us to reconcile pure functions with event management. This means it is possible to create applications requiring inputs from the user or access to third-party services and external data sources. This is especially important as more and more websites make use of web sockets and other such technologies to continually push data to the users.

Besides access to data sources, FRP is great when doing user interface work. A task is usually done with JavaScript on the Web, as PHP is mostly used to treat the request itself and serves an HTML response. PHP might, however, be used more on the desktop with initiative, such as the wrapper around **libui** available in beta for PHP 7 (https://github.com/krakjoe/ui).

Desktop applications in PHP, being a fairly new topic in the community, now might be a great time to create some best practices around it based on state-of-the-art that is functional reactive programming.

We just brushed the surface of this new way of designing applications as it would require a lot more than a chapter to do so fully. Both books mentioned previously are a great starting point if you want to learn more about the topic.

In this chapter, we've learned a bit about the history of FRP. We also tried to discover the differences between traditional reactive programming and its functional counterpart. We quickly spoke about time-traveling debugging and then showed a few examples in PHP.

You have just finished the last chapter of this book. I hope you've had as much fun reading it as it was for me writing it. I also hope I was able to interest you in the topic of functional programming and that you will try to implement the various techniques we've seen in this book in your future projects. There would be no better reward for me than knowing I was able to get a fellow developer interested in this wonderful topic.

Before we part, may I suggest you read the `Appendix`, *What are we Talking About When we Talk About Functional Programming*. It contains a more thorough definition of what functional programming is, its benefits, and its history. You will also find a glossary at the end explaining various terms, some of them seen in this book and others new.

So long, and thanks for all the fish.

Chapter 12. What Are We Talking about When We Talk about Functional Programming

Functional programming has gained a lot of traction in the last few years. Various big tech companies have started using functional languages:

- Twitter on Scala: http://www.artima.com/scalazine/articles/twitter_on_scala.html
- WhatsApp being written in Erlang: http://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success
- Facebook using Haskell: https://code.facebook.com/posts/302060973291128/open-sourcing-haxl-a-library-for-haskell/1

There has been some really wonderful and successful work done on functional languages that compile to JavaScript: the **Elm** and **PureScript** languages, to name a couple. There are efforts to create new languages that either extend or compile to some more traditional languages; we can cite the **Hy** and **Coconut** languages for Python.

Even Apple's new language for iOS development, **Swift**, has multiple concepts from functional programming integrated into its core.

However, this book is not about using a new language, it is about benefiting from functional techniques without having to change our whole stack or learn a whole new technology. By just applying some principles to our everyday PHP, we can greatly improve the quality of our life and our code.

But before going further, let's start with a gentle introduction to what the functional paradigm really is and explain where it comes from.

What is functional programming all about?

If you try searching the Internet for a definition of functional programming, chances are you will at some point find the Wikipedia article (https://en.wikipedia.org/wiki/Functional_programming). Among other things, functional programming is described as follows:

> *In computer science, functional programming is a programming paradigm-a style of building the structure and elements of computer programs-that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.*

The Haskell wiki (https://wiki.haskell.org/Functional_programming) describes it like this:

> *In functional programming, programs are executed by evaluating expressions, in contrast with imperative programming where programs are composed of statements which change global state when executed. Functional programming typically avoids using mutable state.*

Although our take might be a bit different, we can outline some key definitions of functional programming from them:

- Evaluation of mathematical functions or expressions
- Avoiding mutable states

From those two core ideas, we can derive a lot of interesting properties and benefits, which you will discover in this book.

Functions

You're probably aware of what a function is in a programming language, but how it is different from a mathematical function, or as Haskell calls

it, an expression?

A mathematical function does not care about the outside world, or the state of the program. For a given set of inputs, the outputs will always be exactly the same. To avoid confusion, developers often use the terms **pure functions** in this case. We discussed this in Chapter 2, *Pure Functions, Referential Transparency and Immutability*.

Declarative programming

Another difference is that functional programming is also sometimes called **declarative programming**, in contrast to imperative programming. These are called **programming paradigms**. Object-oriented programming is also a paradigm, but one that is strongly tied to the imperative one.

Instead of explaining the difference at length, let's demonstrate it with an example. First an imperative one using PHP:

```
<?php
function getPrices(array $products) {
  // let's assume the $products parameter is an array of
products.
  $prices = [];

  foreach($products as $p) {
    if($p->stock > 0) {
      $prices[] = $p->price;
    }
  }
  return $prices;
}
```

Now let's see how you can do the same with SQL, which is, among other things, a declarative language:

```
SELECT price FROM products WHERE stock > 0;
```

Notice the difference? In the first example, you tell the computer what to do step by step, taking care of storing intermediary results yourselves. The second example only describes what you want; it will then be the

role of the database engine to return the results.

In a way, functional programming looks a lot more like SQL than it does the PHP code we just saw.

Without any explanation, here is how you could do it with PHP in a more functional approach:

```
<?php
function getPrices2(array $products) {
  return array_map(function($p) {
    return $p->price;
  }, array_filter(function($p) {
    return $p->stock > 0;
  }));
}
```

I'll readily admit that this code might not be really clearer than the first one. This can be improved by using dedicated libraries. We will also see in detail the advantages of such an approach.

Avoiding mutable state

As the name itself implies, functions are the most important building block of functional programming. The purest of functional languages will only allow you to use functions, no variables at all, thus avoiding any problems with state and mutating it, and at the same time making any kind of imperative programming impossible.

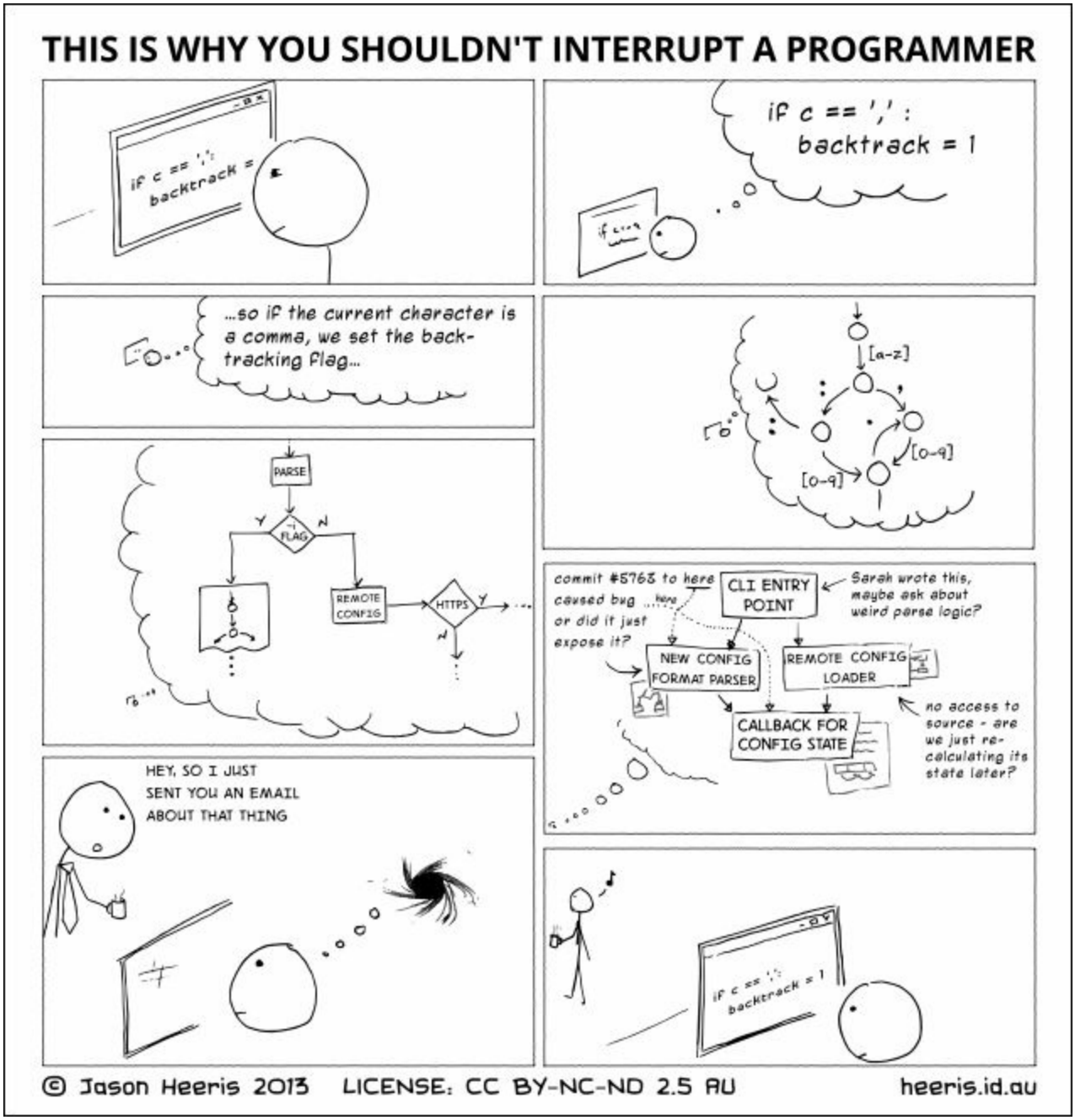
Although nice, the idea is not practical; this is why most functional languages allow you to have some kind of variable. However, those are often immutable, meaning that, once assigned, their value can't change.

Why is functional programming the future of software development?

As we just saw, the functional world is moving, its adoption by the enterprise world is growing, and even new imperative languages take inspiration from functional languages. But why it is so?

Reducing the cognitive burden on developers

You've probably often read or heard that a programmer should not be interrupted because even a small interruption can lead to literally tens of minutes being lost. One of my favorite illustrations of this is the following comic:



This is partly due to the cognitive burden, or in other words, the amount of information you have to keep in memory in order to understand the problem or function at hand.

If we were able to reduce this issue, the benefits would be huge:

- Code will take less time to understand and will be easier to reason about
- Interruption will lead to less disruption in the mental process
- Fewer errors will be introduced due to forgetting a piece of

information
- Small learning curve for newcomers on the project

I posit that functional programming can greatly help.

Keeping the state away

One of the main contenders when it comes to the cognitive burden, as is depicted very well in the comic shown previously, is keeping all these little bits of state information in mind when trying to understand what a piece of code does.

Each time you access a variable or call a method on an object, you have to ask yourself what its value would be and keep that in mind until you reach the end of the piece of code you're currently reading.

By using pure functions, nearly all of this goes away. All your parameters are right there, in the function signature. Moreover, you have the absolute certainty that any subsequent call with the same parameters will have exactly the same outcome, because your function doesn't rely on external data or any object state.

To drive the nail further, let's cite *Out of the Tar Pit* by Ben Moseley and Peter Marks:

> *[...] it is our belief that the single biggest remaining cause of complexity in most contemporary large systems is state, and the more we can do to limit and manage state, the better.*

You can read the whole paper at http://shaffner.us/cs/papers/tarpit.pdf.

Small building blocks

When you do functional programming, you usually create a lot of small functions. You can then compose them like Lego blocks. Each of those small pieces of code is often easier to understand than this big messy method that tries to do a lot of things.

I am not saying that all imperative code is a big mess, just that having a functional mindset really encourages writing small and concise functions that are easier to work with.

Locality of concerns

Let's have a look at the two following examples:

```
$errors = [];
$error_count = 0;

$file = fopen("./some_file.txt", "r");
$line = fgets($file);

while($error_count < 40 && $line !== false) {
    if(strpos($line, 'ERROR') === 0) {
        $errors[] = $line;
        ++$error_count;
    }

    $line = fgets($file);
}



$errors = collect(file_get_contents('./some_files.txt'))
          ->filter(function($l) { return strpos($l, 'ERROR') === 0; })
          ->take(40)
          ->toArray();
```

Imperative versus functional-separation of concerns

As illustrated previously in both fictional code snippets, functional techniques help you organize your code in a way that encourages locality of concerns. In the snippets, we can separate the concerns as follows:

- Creating a list
- Getting data from a file
- Filtering all lines starting with **ERROR** text

- Taking the first 40 errors

The second snippet clearly has a better locality for each of those concerns; they are not spread out in the code.

One can argue that the first code is not optimal and could be rewritten to achieve the same results. Yes, it's probably true. But as for the previous point, a functional mindset encourages this kind of architecture from the get-go.

Declarative programming

We saw that declarative programming is about the *what* instead of the *how*. This helps understanding new code a lot, because our minds have a much easier time thinking about what we want instead of how to do it.

When you order something online or at a restaurant, you don't imagine how, what you want will be created or delivered, you just think of what you want. Functional programming is the same-you start with some data and you tell the language what you want done with it.

This kind of code is also often easier to understand for non-programmers or people with less experience in the language, because we can visualize what will happen to the data. Here is another citation from *Out of the Tar Pit* illustrating this:

> *When a programmer is forced (through use of a language with implicit control flow) to specify the control, he or she is being forced to specify an aspect of how the system should work rather than simply what is desired. Effectively they are being forced to over-specify the problem*

Software with fewer bugs

We already saw that functional programming reduces the cognitive burden and makes your code easier to reason about. This is already a huge win when it comes to bugs, because it will allow you to spot issues

quickly as you will spend less time understanding how the code works to focus on what it should do.

But all the benefits we've just seen have another advantage. They make testing a lot easier too! If you have a pure function and you test it with a given set of values, you have the absolute certainty that it will always return exactly the same thing in production.

How many times have you thought your test was fine, only to discover you had some kind of hidden dependency to an obscure state deep in your application that triggered an issue in some particular circumstances? This ought to happen a lot less with pure functions.

We also learn about property-based testing later in the book. Although the technique can be used on any imperative codebase, the idea behind it came from the functional world.

Easier refactoring

Refactoring is never easy. But since the only inputs of a pure function are its parameters and its sole output is the returned value, things are simpler.

If your refactored function continues to return the same output for a given input, you can have the guarantee that your software will continue to work. You cannot forget to set some state somewhere in an object, because your functions are side-effect free.

Parallel execution

Our computers have more and more cores and the cloud has made it a lot easier to share work across a bunch of nodes. The challenge, however, is ensuring that a computation can be distributed.

Techniques such as mapping and folding, coupled with immutability and the absence of state, make this pretty easy.

Sure, you will still have issues related to distributed computing itself,

such as partitions and failure detection, but splitting the computation into multiple workloads will be made a lot easier! If you want to learn more about distributed systems, I can recommend this article (http://videlalvaro.github.io/2015/12/learning-about-distributed-systems.html) by a former colleague.

Enforcing good practices

This book is the proof that functional programming is more about the way we do things instead of a particular language. You can use functional techniques in nearly any language that has functions. Your language still needs to have certain properties, but not that many. I like to talk about having a functional mindset.

If this is so, why do companies move to functional languages? Because those languages enforce the best practice we will learn in this book. In PHP, you will have to always remember to use functional techniques. In Haskell, you cannot do anything else; the language forces you to write pure functions.

Sure, you can still write bad code in any language, even the purest ones. But usually, people, and developers especially, like to take the path of least resistance. And if this path is the one that leads to quality code, they will take it.

A quick history of the functional world

Historically, functional programming has its roots in the academic world. It's only in recent years that more mainstream companies started using it to develop consumer-facing applications. Some new research into the field is now even done by people outside of universities. But let's begin at the beginning.

The first years

Our story starts in the 1930s when Alonzo Church formalized the Lambda Calculus, a way to solve mathematical problems using functions accepting other functions as parameters. Although this is the foundation of functional programming, it took 20 years for the concept to be first used to implement a programming language when **Lisp** was released in 1958 by John McCarthy. To be fair, **Fortran**, considered the first programming language, was released in 1957.

Although LISP is considered a multi-paradigm language, it is often cited as the first functional language. Quickly, others took the hint and started working around the idea of functional programming, leading to the creation of **APL** (1964), **Scheme** (1970), **ML** (1973), **FP** (1977), and many others.

FP in itself is more or less dead right now, but the lecture in which it was presented by John Backus was pivotal to the research into the functional paradigm. It might not be the easiest read, but it's really interesting nonetheless. I can only suggest you give the whole paper a try at http://worrydream.com/refs/Backus-CanProgrammingBeLiberated.pdf.

The Lisp family

Scheme, first released in 1970, is an attempt to fix some of the shortcomings of Lisp. In the meantime, Lisp gave birth to a

programming language family or dialects:

Common Lisp (1984): an attempt to write a language specification to reunite all the Lisp dialects that were being written at the time.

Emacs Lisp (1985): the scripting language used to customize and extend the Emacs editor.

Racket (1994): first created to be a platform around language design and creation, it's now used in multiple areas such as game scripting, education, and research.

Clojure (2007): created by Rich Hickey after a lengthy reflection to create the perfect language. Clojure targets the **Java Virtual Machine** (**JVM**). It is interesting to note that Clojure can also now have other targets, for example, JavaScript (ClojureScript) and the .NET virtual machine.

Hy (2013): a dialect that targets the Python runtime, allowing the use of all Python libraries.

ML

ML also spawned some children, most notably **Standard ML** and **OCaml** (1996) which are still in use today. It is also often cited as influence in the design of a lot of modern languages. To name a few: Go, Rust, Erlang, Haskell, and Scala.

The rise of Erlang

I said earlier that the mainstream use of functional language is something that started happening in the last few years. This is not entirely true. Ericsson started working on Erlang as soon as 1986, interested in the stability and robustness promised by a functional language.

At first, Erlang was implemented on top of **Prolog** and it proved too slow, but a rewrite to use a virtual machine compiling Erlang to C in

1992, allowed Ericsson to use Erlang on production telephony system as early as 1995. Since then, it has been used worldwide by telecom companies and is considered one of the best languages when it comes to high availability.

Haskell

The year 1990 marked the first release of Haskell, the result of specification work done by academics around the world to create the first open standard around lazy purely functional languages. The idea was to consolidate existing functional languages into a common one so that it could be the basis for further research in functional language design.

Since then, Haskell has grown from a purely academic language in to one of the leading functional languages.

Scala

Scala development was started in 2001 by former Java core developer Martin Odersky. The main idea was to make functional programming more approachable by mixing it with more traditional imperative concepts. The first public release in 2004 targeted both the JVM and the **Common Runtime Language** (**CRM**) used by .NET (this second target was later dropped in 2012).

Scala source code can use its language construct alongside those from the target virtual machine. The ability to use existing Java libraries directly and the ability to fall back to an imperative style is one of the reasons Scala quickly gained ground in the enterprise world.

Since Android uses a Java-compatible virtual machine, Scala is well suited for mobile development and there's also an initiative to compile it to JavaScript, meaning you can use it on both the server and the client for web development.

The newcomers

Nowadays, functional programming languages are starting to gain more mainstream acceptance and new languages are created outside of the academic world. Here is a quick overview of what is being actively worked on by people around the world.

Elm is a serious attempt to create a functional language compiling to JavaScript besides ClojureScript. It is the result of a thesis by Evan Czaplicki trying to create a functional reactive language, a concept we will look into in the last chapter of the book. It gained some coverage when a time-traveling debugger (http://debug.elm-lang.org/) was first presented some years ago, an idea that has since been implemented with much more pain in JavaScript frameworks such as **React**. The barrier of entry is greatly eased by an online editor, really great tutorials, and the fact that you can use **npm** to install it.

PureScript is another functional language compiling to JavaScript. It is closer to Haskell than Elm is and follows a more mathematical approach. The community is smaller, but a lot of work is going on to make the language user-friendly. The PureScript compiler was written in Haskell, it's a bit harder to get started but it's worth it if you want to have robust client-side code.

Idris is, in my opinion, not really ready to shine in a production environment. It has its place in this list, however, as it is one of the more advanced functional languages implementing dependent types. A dependent type is an advanced typing concept that is mostly seen in purely academic languages. It's beyond scope of this book to explain it in detail, but let's do a quick example: *a pair of integers* is a **type**; *a pair of integers where the second one is greater than the first* is a **dependent type** because the type depends on the value of the variable. The advantages of such a typing system is that you can prove more thoroughly that your data is correct and thus the result of your software is also correct. This is, however, a really advanced technique and such languages are rare and hard to learn.

Functional jargon

Like every other field, functional programming comes with its own jargon. This small glossary has the goal to make reading the book easier and also provide you with more understanding of the resources you will find online.

Arity

The number of parameters a function takes. The terms nullary, unary, binary, and ternary are also used to denote functions that take 0, 1, 2, and 3 parameters respectively. See also variadic as follows.

Higher-order functions

A function that returns another function. Chapter 1, *Functions as First Class Citizens*, further explain the concepts of higher-order functions as this is one of the foundations of functional programming.

Side effects

Anything that affects the world outside the current function: changing a global state, a variable passed by reference, a value in an object, writing to the screen or a file, taking user inputs. This concept is an important one and will be explored further in multiple chapters of the book.

Purity

A function is said to be pure if it only uses the explicit parameters and has no side effects. A pure function is a function that will always yield exactly the same result when called with the same parameters. A pure language is a language allowing only pure functions. This concept is an angular stone of functional programing as discussed in Chapter 2, *Pure Functions, Referential Transparency, and Immutability*.

Function composition

Composing functions is a useful technique to reuse various functions as building blocks to achieve more complex operations. Instead of always calling the function `g` on the result of the function `f`, you can compose both functions to create a new function `h`. Chapter 4, *Composing Functions*, demonstrates how this idea can be used.

Immutability

An immutable variable is a variable that cannot be changed once it has been assigned a value.

Partial application

The process of assigning a given value to some parameters of a function to create a new function of a smaller arity. This is sometimes called fixing or binding a value to a parameter. This is a bit difficult to achieve in PHP, but Chapter 4, *Composing Functions*, gives some idea of how to do it.

Currying

Akin to partial application, currying is the process of transforming a function with multiple parameters into multiple unary functions composed to achieve the same result. The reason and idea behind currying were presented in Chapter 4, *Composing Functions*.

Fold/reduce

The process of reducing a collection to a *single* value. This is an often-used concept in functional programming and was demonstrated at length in Chapter 3, *Functional Basis in PHP*.

Map

The process of applying a function on all values of a collection. This is an often-used concept in functional programming and was demonstrated at length in Chapter 3, *Functional Basis in PHP*.

Functor

Any type of value or collection to which you can apply a mapping operation. The functor, given a function, is responsible for applying it to its inner value. It is said that the functor *wraps* the value. This concept was presented in Chapter 5, *Functors, Applicatives, and Monads*.

Applicative

A data structure holding a function inside a context. The applicative, given a value, is responsible for applying the "inner" function to it. It is said that the functor *wraps* the function. This concept was presented in Chapter 5, *Functors, Applicatives, and Monads*.

Semigroup

Any type for which you can associate values two by two. For example, strings are a semigroup because you can concatenate them.

Integers have multiple semigroups:

- The Addition semigroup, where you add integers together
- The Multiplication semigroup, where you multiply the integers together

Monoid

A monoid is a semigroup that also has an identity value. The identity value is a value that when associated with an object of the same type does not change its value. The Addition identity for integers is 0 and the identity for strings is the empty string.

A monoid also requires that the order of association to multiple values does not change the result, for example, *(1 + 2) + 3 == 1 + (2 + 3)*.

Monad

A monad can act both as a functor or as an applicative; refer to the

dedicated Chapter 5, *Functors, Applicatives, and Monads*, for more information.

Lift/LiftA/LiftM

The process of taking something and putting it inside a functor, applicative, or monad respectively.

Morphism

A transformation function. We can distinguish multiple kinds of morphisms:

- **Endomorphism**: The type of the input and output stays the same, for example, making a string uppercase.
- **Isomorphism**: The type changes, but the data stays the same, for example, transforming an array containing coordinates to a Coordinate object.

Algebraic type / union type

The combination of two types into a new one. Scala calls those either types.

Option type / maybe type

A union type that contains a valid value and the equivalent of null. This kind of type is used when a function is not sure to return a valid value. Chapter 3, *Functional Basis in PHP*, explains how to use these to simplify error management.

Idempotence

A function is said to be idempotent if reapplying it to its result does not produce a different result. If you compose an idempotent function with itself, it will still yield the same result.

Lambda

A synonym for an anonymous function, that is, a function assigned to a variable.

Predicate

A function that returns either true or false for a given set of parameters. Predicates are often used to filter collections.

Referential transparency

An expression is said to be referentially transparent if it can be replaced by its value without changing the outcome of the program. The concept is tightly linked to purity. Chapter 2, *Pure functions, Referential Transparency, and Immutability*, explores the slight differences between the two.

Lazy evaluations

A language is said to be lazily evaluated if the result of an expression is only computed when it's needed. This allows you to create an infinite list and is only possible if an expression is referentially transparent.

Non-strict language

A non-strict language is a language where all constructs are lazily evaluated. Only a handful of languages are non-strict, mostly due to the fact that the language has to be pure in order to be non-strict and it poses non-trivial implementation issues. The most well-known non-strict language is probably Haskell.

Nearly all commonly seen languages are strict: C, Java, PHP, Ruby, Python, and so on.

Variadic

A function with dynamic arity is called **variadic**. This means the function accepts a variable number of parameters.

Table of Contents