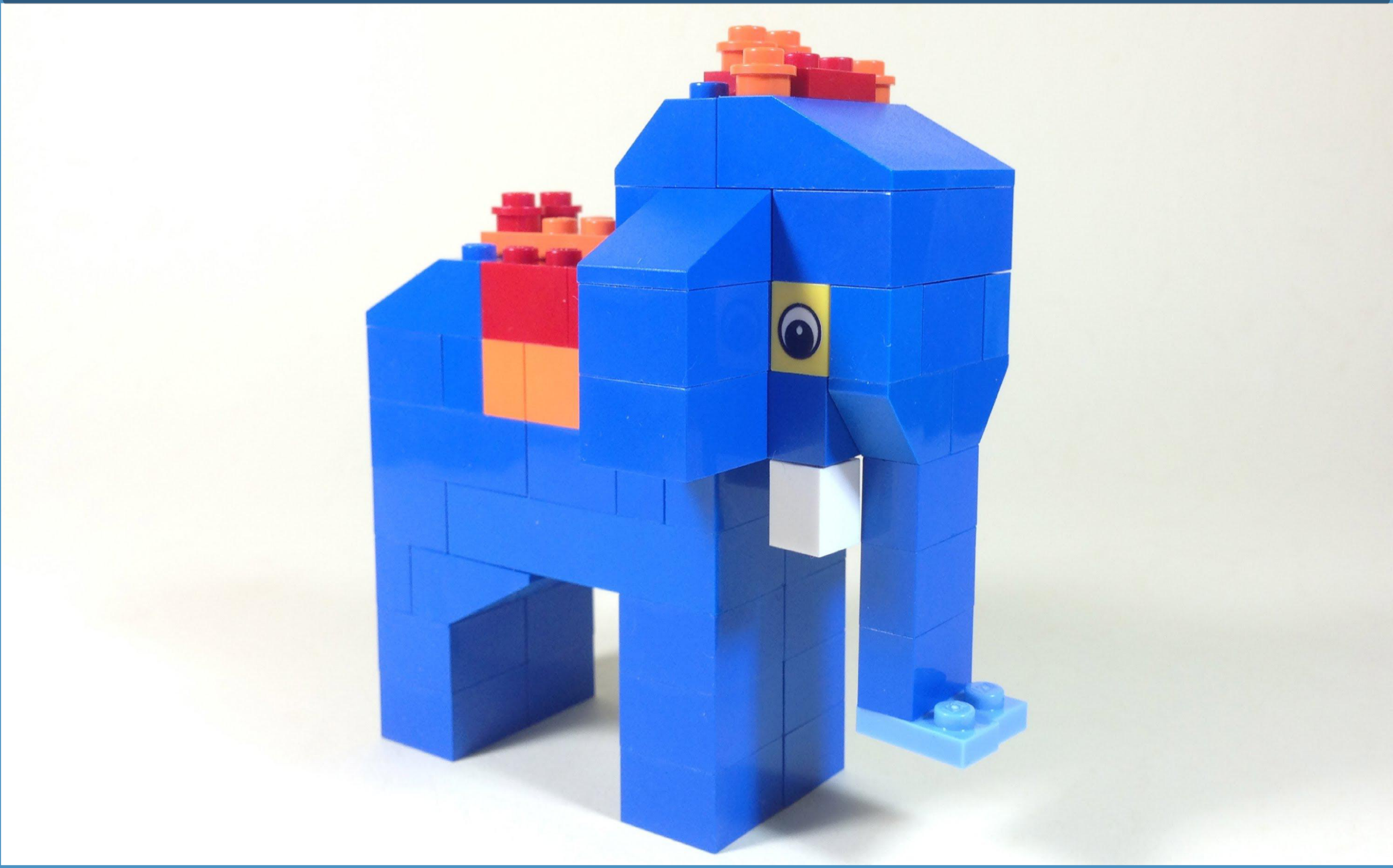
Functional PHP

The Art of Function Composition



Luis Atencio

Functional PHP

The Art of Function Composition

Luis Atencio

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

This version was published on 2016-09-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my wife Ana, the girl of my dreams

Contents

Thinking Functionally in PHP?

Introduction

That's right! If you thought that PHP stood for *PHP: Hypertext Preprocessor*, you're wrong. It stands for **PH**unctional **P**rogramming… OK I'm only kidding.

The PHP community has come a long way since the early starts of PHP mainly as a procedural, imperative language. Now, since PHP 5 you've become a Object-Oriented (OO) developer. You take advantage of abstract classes and interfaces to properly implement the guiding principles of polymorphism, inheritance, and encapsulation. All of this comes into play when building rich domain models utilizing all of the coolest design patterns. Having learned all this, have you been able to reduce the development effort of building large enterprise web applications? Certainly. But is complexity still an issue and bugs frequent? Is your application easy to unit test? How reusable is your code?

The PHP applications of yesterday are no match for the complex, dynamic, and distributed applications of today. It's commonplace now that our users demand that their applications run in cloud environments, integrated with a network of third-party services, and expect SLAs to hold at above 99%. The new buzzword is *microservices* architectures. You'll always be dealing with having to balance low cost with return of investment against our desire to build robust, maintainable architectures.

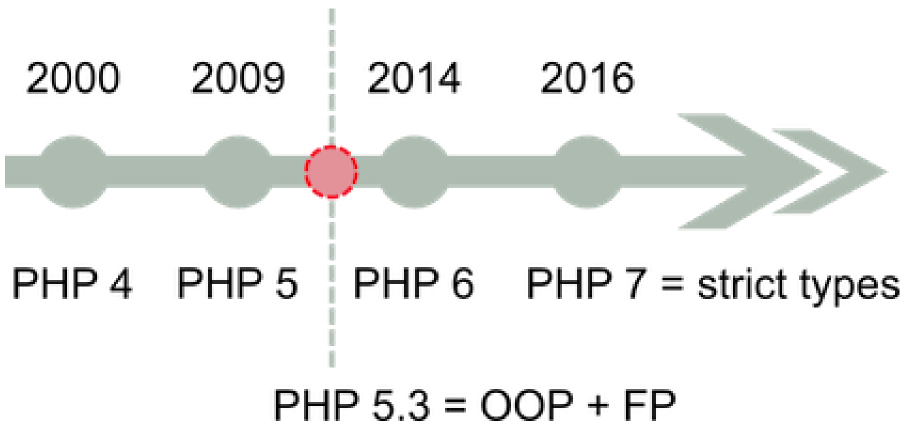
Naturally, as developers, we gravitate towards MVC frameworks that help us create an extensible and clean system design with scaffolding and plumbing for routing, templates, persistence models, services, dependency injection (DI), and built-in integration with database servers–Laravel is a good example of this and there are many others. Despite all of this, our business logic code is still becoming hard to reason about, and this is because we still use things like shared variables, mutable state, monolithic functions, side effects, and others. These seemingly small concerns, which we secretly know to be bad practices but do them anyways, are what functional programming encourages and challenges you to pay close attention to.

Object-oriented design certainly moves the needle in the right direction, but we need more. Perhaps you've been hearing about functional programming (FP) in recent years and how companies like Twitter move to Scala, WhatsApp written in Erlang. Also, language manufacturers are placing functional constructs into their languages. Hence, Java, JavaScript, F#, C#, Scala, Python, Ruby, all have some form of functional features because the industry is realizing that writing code functionally is opening the door to very clean and extensible architectures, as well as making their developers more productive. Companies like Netflix have bet their success on functional-reactive systems, which are built heavily on these ideas, so we'll look at reactive solutions built only using PHP.

If you don't know that PHP also supports writing functional code, then you downloaded the right book. FP is the programming paradigm you need. While based on very simple concepts, FP requires a shift in the way you think about a problem. This isn't a new tool, a library, or an API, but a different approach to problem solving that will become intuitive once you've understood its basic principles, design patterns, and how they can be used against the most complex tasks. Also, it's not an all or nothing solution. In later chapters, I'll show you how FP can be used in conjunction with OO and unpack the meaning of "OO in the large, FP in the small."

Which PHP version to use?

As I mentioned before, you can implement a functional style using PHP 5.3+. This is around the time that the *Closure* class was added to the language (more on this later). However, there are good reasons for upgrading to PHP 7. Aside from it being much faster, matching the runtime speed of Hack[1], and sometimes even better, the latest release adds strict typing and scalar type declarations[2].



Type declarations allow you to qualify any function parameter with its proper class or scalar type (boolean, integer, string, MyClass, etc). These were partially supported in PHP 5 as "type hints" but without scalar support. In PHP 7, you can also declare the type of a function's return value.

Being a dynamic language, PHP will always attempt to coerce values of the wrong type into the expected scalar type, if appropriate. For instance, a function that expects an integer argument, when given a string, will coerce that value to an integer:

```
function increment($counter) {
        return ++$counter;
}
increment("Ft. Lauderdale"); //-> Ft. Lauderdalf ????
```

[1] http://hacklang.org/

[2] http://php.net/manual/en/functions.arguments.php#functions.arguments.type-declaration

Sometimes we want this flexibility, but when we overdo it this can lead to very confusing and hard to read code. What we'll do in this book is use PHP's strict typing mechanism which you can turn on by including this at the top of every file:

```
declare(strict_types=1);
```

Type declarations makes your code instantly self-documented, allows the compiler to perform certain clever optimizations, and lets modern IDEs better guide you using type inference. Types checks also ensure your code is correct by enforcing type constraints on your function calls. In PHP, a *TypeError* occurs when a function contract is violated:

```
increment("Ft. Lauderdale");

PHP Warning:  Uncaught TypeError: Argument 1 passed to increment() must be of th\
e type integer, string given...
```

Armed with the proper tool, let's discuss why learning to think functionally is important and how it can help you combat the complexities of PHP programs.

Hello FP

As I mentioned before, functional programming is not a framework or a tool, but a way of writing code; thinking functionally is radically different from thinking in object-oriented or imperative terms. So, how do you become functional? How do you begin to think this way? Functional programming is actually very intuitive once you've grasped its essence. Unlearning old habits is actually the hardest part and can be a huge paradigm shift for most people that come from a different background.

In simple terms, FP is a software development style that places a major emphasis on the use of functions. In this regard, you might consider it a procedural programming paradigm (based on procedures, subroutines, or functions), and at its core it is, but with very different philosophies. You might say, "well, I already use functions on a day-to-day basis at work; what's the difference?" As I mentioned earlier, functional programming requires you to think a bit differently about how to approach the tasks you are facing. Your goal will be to abstract entire control flows and operations on data with functions in order to avoid *side effects* and *reduce mutation of state* in your application. By practicing FP, you'll become an expert in certain PHP language constructs that are rarely used in other paradigms, like taking advantage of closures and higher-order functions, which were introduced back in PHP 5.3. Both of these concepts are key to building the functional primitives that you'll be using in your code.

Without further ado, let's start with a simple 'Hello FP' example. Creating a simple script is probably the easiest way to get PHP up and running, and that's all you'll need for this chapter. Fire up your PHP REPL [`shell> php -a`]. Because I want to focus more on building the theoretical foundations in this chapter, I'll use very simple examples and simple functions. As you move through the book, we'll tackle on more real-world examples that involve file systems, HTTP requests, databases, etc.

```
$file = fopen('ch01.txt', 'w');
fwrite($file, 'Hello FP!'); //-> writes 'Hello FP'
```

This program is very simple, but because everything is hard-coded you can't use it to display messages dynamically. Say you wanted to change the message contents or where it will be written to; you will need to rewrite this entire expression. Consider wrapping this code with a function and making these change-points parameters, so that you write it once and use it with any configuration.

```
function toFile($filename, $message) {
    $file = fopen($filename, 'w');
    return fwrite($file, $message);
}

toFile('ch01.txt', 'Hello FP'); //-> writes 'Hello FP'
```

An improvement, indeed, but still not a completely reusable piece of code. Suppose your requirements change and now you need to repeat this message twice. Obviously, your reaction will be to change the business logic of `toFile` to support this:

```
function toFile($filename, $message) {
    $file = fopen($filename, 'w');
    return fwrite($file, $message. ' ' . $message);
}

toFile('ch01.txt', 'Hello FP'); //-> writes 'Hello FP Hello FP'
```

This simple thought process of creating parameterized functions to carry out simple tasks is a step in the right direction; however, it would be nice to minimize reaching into your core logic to support slight changes in requirements. We need to make our code more *extensible*. Thinking functionally involves treating parameters as not just simple scalar values but also as functions themselves that provide additional functionality; it also involves using functions or (*callables*) as just pieces of data that can be passed around anywhere. The end result is that we end up evaluating and combining lots of functions together that individually don't add much value, but together solve entire programs. I'll make a slight transformation to the function `toFile`:

```
function toFile($filename): callable {
        return function ($message) use ($filename): int {
                $file = fopen($filename, 'w');
            return fwrite($file, $message);
        };
}
```

At first, it won't be intuitive why I made this change. Returning functions from other functions? Let me fast-forward a bit more and show you how I can use this function in its current form. Here's a sneak peek at this same program using a functional approach.

```
$run = compose(toFile('ch01.txt'), $repeat(2), 'htmlentities');
$run('Functional PHP <i>Rocks!</i>');

//-> writes 'Functional PHP &lt;i&gt;Rocks!&lt;/i&gt;
//           Functional PHP &lt;i&gt;Rocks!&lt;/i&gt;'
```

And just as I directed this input text to a file, I could have just as easily sent it to the console, to the database, or over the network. Without a doubt, this looks radically different than the original. I'll highlight just a couple of things now. For starters, the file is not a scalar string anymore; it's a function or *callable* called `toFile`. Also, notice how I was able to split the logic of IO from manipulating its contents. Visually, it feels as though we're creating a bigger function from smaller ones. In traditional PHP applications, it's rare to see functions used this way. We typically declare functions and invoke them directly. In FP, it's common to pass around function references.

Above all, the important aspect about this code sample above is that it captures the process of decomposing a program into smaller pieces that are more reusable, reliable, and easier to understand; then they are combined to form an entire program that is easier to reason about as a whole. Thinking about each of these simple functions individually is very easy, and separating the concerns of business logic and file IO makes your programs easier to test. Every functional program follows this fundamental principle.

Now I just introduced a new concept `compose`, itself a function, to invoke a series of other functions together. I'll explain what this means later on and how to use it to its fullest potential. Behind the scenes, it basically links each function in a chain-like manner by passing the return value of one as input to the next. In this case, the string "Functional PHP <i>Rocks!</i>" was passed into `htmlentities` which returns an HTML-escaped string, passed into `$repeat(2)`, and finally its result passed into `toFile`. All you need to is make sure every function individually executes correctly. Because PHP 7 ensures that all of the types match, you can be confident this program will yield the right results. This is analogous to stacking legos together and will be central to the theme in this book: "The Art of Function Composition."

So, why does functional code look this way? I like to think of it as basically parameterizing your code so that you can easily change it in a non-invasive manner—like adjusting an algorithm's initial conditions. This visual quality is not accidental. When comparing the functional to the non-functional solution, you may have noticed that there is a radical difference in style. Both achieve the same purpose, yet look very different. This is due to functional programming's inherent declarative style of development.

Declarative coding

Functional programming is foremost a declarative programming paradigm. This means they express a logical connection of operations without revealing how they're implemented or how data actually flows through them. Chapter 3 shows you how to build these data flows using point-free coding.

As you know, the more popular models used today in PHP are procedural and object-oriented, both imperative paradigms. For instance, sites built using older versions of Wordpress or Moodle are heavily procedural; whereas, sites built using Laravel are completely OOP.

Imperative programming treats a computer program as merely a sequence of top-to-bottom statements that change the state of the system in order to compute a result. Let's take a look at a simple imperative example. Suppose you need to square all of the numbers in an array. An imperative program follows these steps:

```
$array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for($i = 0; $i < count($array); $i++) {
   $array[$i] = pow($array[$i], 2);
}

$array; //-> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Imperative programming tells the computer, in great detail, how to perform a certain task (looping through and applying the square formula to each number, in this case). This is the most common way of writing this code and most likely will be your first approach to tackling this problem.

Declarative programming, on the other hand, *separates program description from evaluation*. It focuses on the use of expressions to describe what the logic of a program is or what output would look like without necessarily specifying its control flow or state changes. Familiar examples of declarative code are seen in SQL statements:

```
SELECT * FROM Person
WHERE age > 60
ORDER BY age DESC;
```

Also, Regular Expressions. The following code extracts the host name from a URL:

```
@^(?:http://)?([^/]+)@i
```

Also, CSS (and alt-CSS like LESS or SASS) files are also declarative:

```
body {
    background: #f8f6f6;
    color: #404040;
    font-family: 'Lucida Grande', Verdana, sans-serif;
    font-size: 13px;
    font-weight: normal;
    line-height: 20px;
}
```

In PHP, declarative code is achieved using higher-order functions that establish a certain vocabulary based on a few set of primitives like filter, map, reduce, zip, compose, curry, lift, etc. These are just some of the common terms used a lot with FP that you'll learn about in this book. Once you employ this vocabulary to concoct the instructions (subroutines, functions, procedures, etc) that make up your program, the PHP runtime translates this higher level of abstraction into regular PHP code:

Shifting to a functional approach to tackle this same task, you only need to be concerned with applying the correct behavior at each element and cede control of looping to other parts of the system. I can let PHP's `array_map()` do the work:

```
$square = function (int $num): int {
   return pow($num, 2);
};
array_map($square, $array);  //-> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The function `array_map()` also works with *callables* or functions just like the `compose` function shown earlier. Notice a trend? No? Consider this example. Suppose we also needed to add all of the values within the array. So, we need `add` function:

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

Now, without fiddling with the internals of the function, I use the adder to reduce the array into a single number:

```
array_reduce(array_map($square, $array), 'add'); //-> 285
```

Comparing this code to the imperative version, you see that you've freed yourself from the responsibility of properly managing a loop counter and array index access; put simply, the more code you have, the more places there are for bugs to occur. Also, standard loops are not reusable artifacts, only when they are abstracted within functions. Abstracting loops with functions will allow you to take advantage of PHP's *anonymous function syntax* and *closures*.

Designing for immutability and statelessness

Functions like `array_map()` have other benefits: they are immutable, which means it doesn't change the contents of the original array that's passed to it.

Coding with immutable variables has many benefits such as:

1. One of the main causes of bugs in software is when the state of an object inadvertently changes, or its reference becomes null. Immutable objects can be passed around to any function and their states will always remain the same. You can count on having the peace mind that state is only permitted to grow but never change. It eases the "cognitive load" (the amount of state to keep track of in your head) of any single component of your system.
2. Immutable data structures are important in shared memory multithreaded applications. We won't talk much about concurrent processing in this book because PHP processes run in isolation for the most part. Now, whether designing for parallelism or not, stateless objects is a widely used pattern seen in many common PHP deployments. For example, as a best practice, Symfony services (or service objects) should always be stateless. A service[3] shouldn't persist any state and provide a set of transient functions that take in the domain its working on, perform some kind of computation of business logic, and return the result.

Unfortunately, unlike Scala, F#, and others, PHP provides very poor support for immutable variables. You're only options really are using the `define` function or `const` keyword. Here's a quick comparison of the two:

- `const` are defined at compile time, which means compilers can be clever about how to store them, but you can't conditionally declare them (which is a good thing). The following fails in PHP 7:

```
if (<some condition>) {
   const C1 = 'FOO';
}
else {
   const C2 = 'BAR';
}
```

- Constants declared with 'define' are a bit more versatile and dynamic. Because the compiler won't attempt to allocate space for them until it actually sees them. So you can do this:

[3] https://igor.io/2013/03/31/stateless-services.html

```
if (<some condition>) {
   define('C1', 'FOO');
}
else {
   define('C2', 'BAR');
}
```

But you should always check if a constant has been defined with `defined($name)` before accessing its value using `constant($name)`.

- `const` properly scopes constants into the namespace you're class resides in. `define` will scope constants globally by default, unless the namespace is explicitly added.
- `const` behaves like any other variable declaration, which means it's case sensitive and requires a valid variable name. Because `define` is a function, you can use it to create constants with arbitrary expressions and also declare them to be case insensitive if you wish to do so.

All things considered, `const` is much better to use because it's an actual keyword and not just a function call, which is what constants should be. Perhaps in the future, PHP moves in the direction of allowing this class-level only feature to be used anywhere to define an entity that can only be assigned once, whether it be a variable or function:

```
const $square = function (int $num): int {
   return pow($num, 2);
};
```

Of course, as it is now, it's unrealistic to expect that we declare all of our variables using `define`. But you can get close to achieving stateless programs, just like those Symfony services we spoke about earlier, using pure functions.

Pure functions fix side effects

Functional programming is based on the premise that you will build immutable programs based on pure functions as the building blocks of your business logic. A pure function has the following qualities:

- It depends only on the input provided and not on any hidden or external state that may change during its evaluation or between calls.
- It doesn't inflict changes beyond its scope, like modifying a global object or a parameter passed by reference, after its run.

Intuitively, any function that does not meet these requirements would be qualified as "impure." For example, while services are pure objects, repository or Data Access Objects (DAO) aren't. This includes your Laravel Active Record objects, for example. Operations in DAOs always cause side effects because their sole purpose is to interact with an external resource, the database.

Programming with immutability can feel rather strange at first. After all, the whole point of imperative design, which is what we're accustomed to, is to declare that variables are to mutate from one statement to the next (they are "variable" after all). PHP doesn't make any distinctions between *values* (immutable variables) and standard variables–they're all declared with the same "$" dollar sign. This is a very natural thing for us to do. Consider the following function that reads and modifies a global variable:

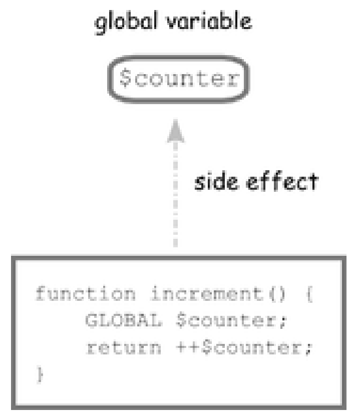
```
// resides somewhere in the global
// space (possibly in a different script)

$counter = 0;

...

function increment(): int {
   GLOBAL $counter;
   return ++$counter;
}
```

You can also encounter side effects by accessing instance state through `$this`:

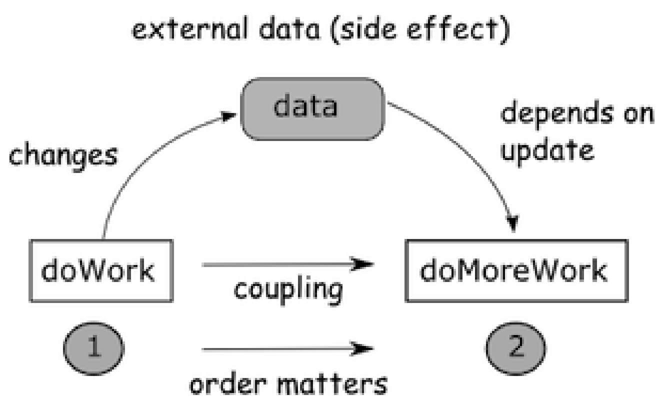
```
class Counter {
   private $_counter;

   public function __construct(int $init) {
      $this->_counter = $init;
   }

   ...

   public function increment(): int {
      return ++$this->_counter;
   }
}
```

Because we don't have support for immutable variable modifiers, with a bit of discipline, we can still achieve the same goal. One thing we can do is stop using global variables and the `GLOBAL` mechanism in PHP. This is not only considered bad practice from a functional point of view, but also a bit frowned upon in modern PHP applications. A function like `increment` is impure as it reads/modifies an external variable `$counter`, which is not local to the function's scope (it could actually live in a complete different file).

In addition, in the case of object methods, using `$this` is automatically accessing the methods's external instance scope. While this isn't nearly as bad as global scope, from a pure FP perspective, it's also a side effect.



Generally speaking, functions have side effects when reading from or writing to external resources. Matters get worse, when this state is shared:



In this case, `doWork` and `doMoreWork` are very tightly coupled. This coupling means that you necessarily need to invoke `doWork` before calling `doMoreWork`, always in that order. Hence, you lose the autonomy of these functions and make them harder to unit test in isolation. Side effects create a *temporal coupling* or dependency, which means the execution of one can determine the outcome of the next. The result of `doMoreWork` is reliant on `doWork`. In functional programming, functions should behave like reusable artifacts that can be evaluated in any order and continue to yield correct results.

Not all side effects are this obvious. Some of them are embedded into language level functions. Frankly, you should be wary of functions that use parameter references `&`, such as:

```
bool sort ( array &$array [, int $sort_flags = SORT_REGULAR ] )
```

Instead of returning a new sorted array, it sorts the array in place and returns a (arguably useless) boolean result:

```
$original = ['z', 'a', 'g'];
sort($original);
$original; //-> ['a', 'g', 'z'];
```

Here are some other forms:

- Changing a variable, property or data structure globally
- Changing the original value of a function's argument
- Processing user input
- Throwing an exception, unless it's caught within the same function
- Printing to the screen or logging
- Querying the DOM of an HTML page and browser cookies
- Writing to/reading from files and databases

So, now you need to ask yourself: What practical value would you get from a program that couldn't do any of these things? Indeed, pure functions can be very hard to use in a world full of dynamic behavior and mutation– the real world. But, to benefit from functional programming you don't need to avoid all of these; FP just provides a framework to help you manage/reduce side effects by separating the pure code from the impure. Impure code produces externally visible side effects like the ones listed above, and in this book you'll learn ways to deal with this.

For instance, I can easily refactor `increment` to accept the current counter:

```
function increment(int $counter): int {
   return ++$counter;
}
```

This pure function is now not only immutable but also has a clear contract that describes clearly the information it needs to carry out its task, making it simpler to understand and use. This is a simple example, of course, but this level of reasoning can be taken to functions of any complexity. Generally, the goal is to create functions that do one thing and combine them together instead of creating large monolithic functions.

Referential Transparency

We can take the notion of purity a step further. In functional programming, we redefine what it means to create a function. In a sense we go back to basics, to the maths, and treat functions as nothing more than a mapping of types. In this case, its input types (arguments) to its output type (return value). We'll use a pseudo Haskell notation throughout the book when document a function's signature. For example, `f :: A -> B` is a function `f` that takes an object of `A` and returns an object of type `B`. So, increment becomes:

```
increment :: int -> int
```

Essentially the arrow notation is used to define any callable. A function like `toFile`, which returns a function from another function, is defined as:

```
toFile :: string -> string -> int
```

Functions in mathematics are predictable, constant, and work exactly like this. You can imagine that `A` and `B` are sets that represent the domain and the codomain of a function, respectively. For instance, the type `int` is analogous to the set of all integers Z. Functions that return the same values given the same input always, resembling this mathematical property, are known as *referentially transparent (RT)*. A RT function can always be directly substituted or replaced for its computed value into any expression without altering its meaning. In other words, these are all equivalent:

```
add(square(increment(4)), increment(16)) =~ add(square(5), 17) =~ add(25, 17) =~\
 25 + 17 = 42
```

This means that you can use the value `42` in place of any of these expressions. This conclusion can only be reached when there are no side effects in your code. Using RT functions we derive the following corollary. Because there are no side effects, *a function's return value is directly derived from its input*. Consequently, void functions `A -> ()` as well as zero-arg functions `() -> A` will typically perform side effects.

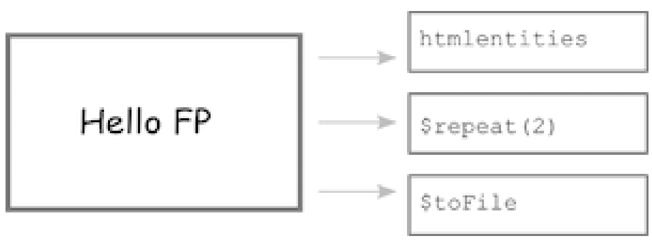
This makes your code not only easier to test, but also allows you to reason about entire programs much easier. Referential transparency (also known as equational correctness) is inherited from math, but functions in programming languages behave nothing like mathematical functions; so achieving referential transparency is strictly on us, especially in a non-pure functional language such as PHP. The benefit of doing is is that when your individual functions are predictable, the sum of the parts is predictable and, hence, much easier to reason about, maintain, and debug, and especially test.

Later on, you'll learn that functional programs are inherently testable, because referential transparency surfaces another principle of good tests, idempotency. An idempotent unit test is a fancy term to describe a unit test that's repeatable and consistent, so that for a given set of input you're guaranteed to compute the same output, always. This "contract" will then be documented as part of your test code; in essence, you get self-documenting code.

By now you realize why pure functions are the pillar of FP, but we have one more concept to introduce. If pure functions are the pillars, then composition is the base or the glue that makes up the entire system.

A look into functional decomposition

At the core, functional programming is effectively the interplay between decomposition (breaking programs into small pieces) and composition (glueing the pieces back together). It is this duality that makes functional programs modular and so effective. As I mentioned previously, the unit of modularity, or "unit of work" is the function itself. Thinking functionally typically begins with decomposition or learning to break up a particular task into logical subtasks (functions). Because the modularity boundary is the function, we can identify three components in the "Hello FP" program earlier:



Modularization in functional programming is closely related to the *Singularity* principle (one of the SOLID design principles), which states that functions should have a single purpose. The idea is that you learn to decompose programs into the simplest reusable units of composition. In FP, your functions should abide by this motto:

Loosely coupled + highly cohesive = No side effects + single purpose

Purity and referential transparency will encourage you to think this way because in order to glue simple functions together they must agree on their types of inputs and outputs as well as arity or number of arguments. You can see how using PHP 7's strict typing helps in this regard, because it helps you paint the flow of your program's execution as it hops from one function to the next. In short, they must agree on their exposed contracts, which is very similar to the *Coding to Interfaces* OO design principle. From referential transparency, we learn that a function's complexity is sometimes directly related to the number of arguments it receives. A practical observation and not a formal concept indicates that the lower the number of function parameters, the simpler the function tends to be.

This process is essentially what `compose` does. Also called a function combinator (for obvious reasons), `compose` can glue together functions in a loosely decoupled way by binding one function's output to the next one's input. This is essentially the same *coding to interfaces* but at a much lower level of granularity.

To sum up, what is functional programming?

With all of this so far, we can pack the fundamental FP principles into the following statement:

"Functional programming refers to the declarative evaluation of pure functions to create immutable programs by avoiding externally observable side effects."

Hope you've enjoyed this brief introduction of some of the topics covered in this book. In the next chapters we'll learn what enables PHP to be used functionally. In particular, you'll learn about higher-order, first-class functions, and closures, and some practical examples of using each technique. This will set the stage for developing data flows in chapter 3. So stay tuned!

Higher-order PHP

I mentioned in chapter 1 that functional programming is not a new framework, library, or design pattern. Rather, it's a way of thinking that offers an alternative to the way in which you design your code. However, paradigms by themselves are just abstract concepts that need the right host language to become a reality. And, yes! PHP is that language. In this chapter, we'll take a look at two important features of PHP that make functional possible: higher-order functions and closures. Both of these are instrumental in building all of the techniques you'll learn about in this book. The goal of this chapter is to teach you how to use functions in a very different way–the *functional* way.

PHP's first-class, higher-order functions

A higher-order function is defined as one that can accept other functions as arguments or return another function. This is in direct relationship with another term you might've heard before, that is *first-class functions*. Both are intimately related, as the ability of a language artifact to be passed in as an argument or returned from a functions hinges on it being considered just another object. This also means, of course, that functions can be assigned to variables. Let's take a look at a few examples:

Functions in PHP can be manipulated just like objects. In fact, if you were to check the type of a function, you'll find out that they are instances of the class `Closure`[4]:

```
var_dump(function () { });

//-> class Closure#1 (0) {

    }
```

If functions behave like objects, it's logical to think we can assign them to variables.

Assigning to a variable

This means that you should be able to treat a function just like any other type of object. Which means that they can be assigned to variables. Consider a simple string concatenation function:

[4]http://php.net/manual/en/class.closure.php

```
$concat2 = function (string $s1, string $s2): string {
        return $s1. ' '. $s2;
};

$concat2('Hello', 'World');  //-> 'Hello World'
```

Behind the scenes this code takes the anonymous function (RHS) and assigns it to the variable `$concat2` (LHS). Alternatively, you can check for the presence of a function variable using `is_callable()`[5]:

```
is_callable($concat2) // 1
```

Returned from a function

Functions can also be returned from other functions. This is an extremely useful technique for creating families of functions. It's also the main part of implementing argument currying, which you'll learn about in later chapters. Consider a simple `concatWith` function:

```
function concatWith(string $a): callable {
   return function (string $b) use ($a): string {
      return $a . $b;
   };
}

$helloWith = concatWith('Hello');
$helloWith('World'); //-> 'Hello World'
```

As a parameter

Supplying functions as parameters allows you to administer specialized behavior on top of another function. Suppose I create a simple function that takes a *callable* and applies it over its other parameters:

```
function apply(callable $operator, $a, $b) {
   return $operator($a, $b);
}
```

Through the callable, I can inject any behavior I want:

[5] http://php.net/manual/en/function.is-callable.php

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

$divide = function (float $a, float $b): float {
   return $a / $b;
};

apply($add, 5, 5); //-> 10

apply($divide, 5, 5); //-> 10
```

Consider a version of apply that's a bit more useful and expressive:

```
function apply(callable $operator): callable {
   return function($a, $b) use ($operator) {
      return $operator($a, $b);
   };
}
```

This function is very explicit in what it's purpose is, and how I can use it to derive other types of functions from it. Let's go over some simple examples:

```
apply($add)(5, 5); //-> 10

apply($divide)(5, 5); //-> 1

// New function adder
$adder = apply($add);
$divider = apply($divide);

$adder(5,5); //-> 10
$divider(5,5); //-> 1
```

I mentioned earlier that higher-order functions allow you to supply specialized behavior via function arguments. Let's see this in action. What would happen if I call `apply($divide)(5, 0)`? Correct, a division by zero error:

```
Warning: Division by zero in .../code/src/ch02/ch02.php ...
```

To fix this, I'll create a function called `safeDivide` that supplies extra null-check logic. This function is a lot more resilient, returning PHP's `NAN` constant back to the caller instead of an exception.

```
function safeDivide(float $a, float $b): float {
   return empty($b) ? NAN : $a / $b;
}

apply($safeDivide)(5, 0); //-> NAN
```

The other reason why I prefer this approach is that checking for `NAN` requires a lot less effort and it's much cleaner than having to try and catch exceptions:

```
try {
   $result = apply($safeDivide)(5, 0);
   ...
   return $result;
}
catch(Exception $e) {
    Log::error($e->getMessage());
}
```

I think this is a much cleaner API design:

```
$result = apply($safeDivide)(5, 0);
if(!is_nan($result)) {
   ...
   return $result;
}
else {
   Log::warning('Math error occurred! Division by zero!');
}
```

This approach avoids throwing an exception altogether. Recall from chapter 1 that throwing exceptions is not only a side effect, as it causes the program stack to unwind and logs written, but also doesn't respect the *Locality Principle* of code. In particular, it fails to obey *spatial locality*, which states that related statements that should be executed sequentially shall be placed near each other. This has more application on CPU architecture, but can also be applied to code design.

Before I leave this topic of passing functions as arguments, it's important to mention that you may pass any user-defined function variable as an argument, as well as most native to PHP, but not the ones that are part of the language such as: `echo`, `print`, `unset()`, `isset()`, `empty()`, `include`, `require`, `require_once`, and others. In these cases, your best bet is to wrap them using your own.

To recap, higher-order functions are possible in PHP because, as of PHP 5.3, they are actually `Closure` instances behind the scenes. Before this version, this was just considered an internal design decision, but now you can reliably take advantage of it to approach problems very differently. In this book, you'll learn to master higher-order functions.

Furthermore, because functions are true instances, as of PHP 5.4 you can actually invoke methods on them which gives you more control of an anonymous function after it's been created (as you might expect, `Closure` instances all implement the magic method `__invoke()`, which is important for consistency reasons with other classes).

Heck, even plain objects are invokable

Aside from having true first-class, higher-order functions, PHP takes it to the next level with invocable objects. Now, this isn't really a functional concept whatsoever, but used correctly it could be a pretty powerful technique. In fact, PHP's anonymous function syntax under the hood gets compiled to a class with an `__invoke()` method on it.

Now, the reason why this isn't really a functional technique per se, is that functional programming tends to impose a clear separation of behavior and state. To put it another way, doing away with the use of `this`. One reason for doing this is that `$this` keyword is a gateway for side effects. Consider this simple `Counter` class:

```
class Counter {
   private $_value;

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

   public function increment(): int {
      return $this->_value++;
   }
}

$c = new Counter(0);
$c->increment(); //-> 1
$c->increment(); //-> 2
$c->increment(); //-> 3
```

The `increment` function is theoretically considered not pure (or impure) because it's reaching for data in its outer scope (the instance scope). Fortunately, this class encapsulates this state pretty well and doesn't expose any mutation methods as in the form of a setter. So, from a practical standpoint, this object is predictable and constant. We can go ahead and make this object invocable by adding the magic `__invoke()` method to it:

```
public function __invoke() {
   return $this->increment()
}

$increment = new Counter(100);
increment(); //-> 101
increment(); //-> 102
increment(); //-> 103
```

In practical functional programming, there are many design patterns that revolve around wrapping values into objects and using functions to manipulate them. But for the most part, we'll prefer to separate the behavior from the state. One way of doing this to keep things semantically meaningful is to prefer static functions that declare arguments for any data they need to carry out their job:

```
class Counter {
   ...
   public static function increment(int $val): int {
      return $val + 1;
   }
}

Counter::increment(100); //-> 101
```

Using containers improve your APIs

Earlier you learned that returning `NAN` in the event that `divide` was called with a zero denominator led to a better API design because it freed your users from having to wrap their code in try/catch blocks. This is always a good thing because exceptions should be thrown only when there's no recovery path. However, we can do better. Working with numbers and `NAN` doesn't really get you anywhere; for example, adding anything to `NAN` (ex. `1 + NAN`) returns `NAN`, and rightfully so. So, instead of burdening your users to place `is_nan` checks after each function call, why not consolidate this logic in one place? We can use containers to do this.

We cans use wrappers to control access to certain variables and provide additional behavior. Looking at it from an OOP design patterns point of view, you can compare this to a Fluent Object pattern. To start with, consider this simple `Container` class:

```
class Container {
   private $_value;

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

   // Unit function
   public static function of($val) {
      return new static($val);
   }

   // Map function
   public function map(callable $f) {
      return static::of(call_user_func($f, $this->_value));
   }

   // Print out the container
   public function __toString(): string {
      return "Container[ {$this->_value} ]";
   }

   // Deference container
   public function __invoke() {
      return $this->_value;
   }
}
```

Containers wrap data and provide a mechanism to transform it in a controlled manner via a mapping function. This is in many ways analogous to the way you can map functions over an array using `array_map()`:

```
array_map('htmlspecialchars', ['</ HELLO >']); //-> [&lt;/ HELLO &gt;]
```

This container behaves exactly the same:

```
function container_map(callable $f, Container $c): Container {
        return $c->map($f);
}
```

I can call it with `container_map`, or I can use it directly. For instance, I can apply a series of transformations on a string fluently like this:

```
$c = Container::of('</ Hello FP >')->map('htmlspecialchars')->map('strtolower');

$c; //-> Container[ &lt;/ hello fp &gt; ]
```

Notice how this looks much cleaner and easier to parse against nesting these function calls one inside the other. Personally, I rather see code in a flattened out and linear model than something like:

```
strtolower(htmlspecialchars('</ Hello FP >')); //-> &lt;/ hello fp &gt;
```

I also added some PHP magic with `__invoke()` that can be used to dereference the container upon invocation as such:

```
$c = Container::of('Hello FP')->map($repeat(2))->map(strlen);
$c(); //-> 16
```

So, what's the use for this pattern? Earlier I mentioned that throwing exceptions, or for that matter, the imperative `try/catch` mechanism has side effects. Let's circle back to that for a moment. Arguably, `try/catch` is also not very declarative and belongs in the same category as for loops and conditional statements. Containerizing values is an important design pattern in FP because it allows you to consolidate the logic of applying a function chain (sequence) of transformations to some value immutably and side effect free; it's immutable because it's always returning new instances of the container.

This can be used to implement error handling. Consider this scenario:

```
$c = Container::of([1,2,3])->map(array_reverse);
print_r($c()) //-> [3, 2, 1]
```

But if instead of a valid array, a `null` value was passed in, you will see:

```
Warning: array_reverse() expects parameter 1 to be array, null given
```

One way to get around this is to make sure all functions you use (PHP or your own) are "safe," or do some level of null-checking. Consider the implementation for a `SafeContainer`:

```
class SafeContainer extends Container {
   // Performs null checks
   public function map(callable $f): SafeContainer {
      if(!empty($this->_value)) {
         return static::of(call_user_func($f, $this->_value));
      }
      return static::of();
   }
}
```

With this I don't have to worry about the error checking to clutter all of my business logic; it's all consolidated in one place. Let's see it in action:

```
$c = SafeContainer::of(null)->map(array_reverse);
print_r($c()); //-> Container[null]
```

The best part about this is that your function chains look exactly the same, there's nothing extra on your part, so you can continue to map as many functions as you need and any errors will be propagated through the entire chain behind the scenes.

Now let's mix it up and use containers with higher-order functions. Here's an example:

```
$c = Container::of('</ Hello FP >')->map(htmlspecialchars)->map(strtolower);

//-> Container[ &lt;/ hello fp &gt; ]
```

Just for completeness sake, here's a container called `SafeNumber` to tackle our division-by-zero problem earlier:

```
class SafeNumber extends Container {
   public function map(callable $f): SafeNumber {
      if(!isset($this->_value) || is_nan($this->_value)) {
         return static::of(); // empty container                }
        else {
           return static::of(call_user_func($f, $this->_value));
        }
    }
}
```

It looks very simple, but the effect of this wrapper type is incredible; most things in functional programming are very basic but with tremendous impact and reach. I'll refactor `safeDivide` earlier to return a `SafeNumber`:

```
function safeDivide($a, $b): SafeNumber {
  return SafeNumber::of(empty($b) ? NAN : $a / $b);
}
```

One thing I like about this function is how honest it is about its return type. It's essentially notifying the caller that something might go wrong, and that it's safe to protect the result. Also, it removes the burden of having to NAN-check any functions that are invoked with this value. I'll show the cases of calling `safeDivide` with a valid as well as an invalid denominator:

```
        function square(int $a): int {
                return $a * $a;
        }

        function increment(int $a): int {
                return $a + 1;
        }

    // valid case
        apply(safeDivide2)(5, 1)->map(square)->map(increment); //-> Container [26]

    // error case
        apply(safeDivide2)(5, 0)->map(square)->map(increment)); //-> Container[ null ]
```

`SafeNumber` abstracts out the details of dealing with an invalid number, so that we're left alone to worry about bigger, more important problems.

In chapter 1, I briefly mentioned that behind the scenes, all functions are instances of the `Closure` class. This is what enables me to map them to containers. This is important to understand, so let's take another look at this class.

Closures

After PHP 5.4+, all functions in PHP are objects created from the `Closure` class. But why do we call it this way and not the more intuitive `Function` type (like JavaScript)? The goal behind this class is to represent anonymous functions[6] where a closure is formed around the function as it's being inlined into, say, a callable argument. For instance, I could use an anonymous function to perform the safe divide (I would recommend you do this only when the function is unique to the problem you're solving and you don't anticipate reusing it somewhere else):

[6] http://php.net/manual/en/functions.anonymous.php

```
apply2(
    function (float $a, float $b): SafeNumber {
           return SafeNumber::of(empty($b) ? NAN : $a / $b);
        })
        (7, 2); //-> Container [3.5]
```

They are also used extensively in modern MVC frameworks, like Laravel:

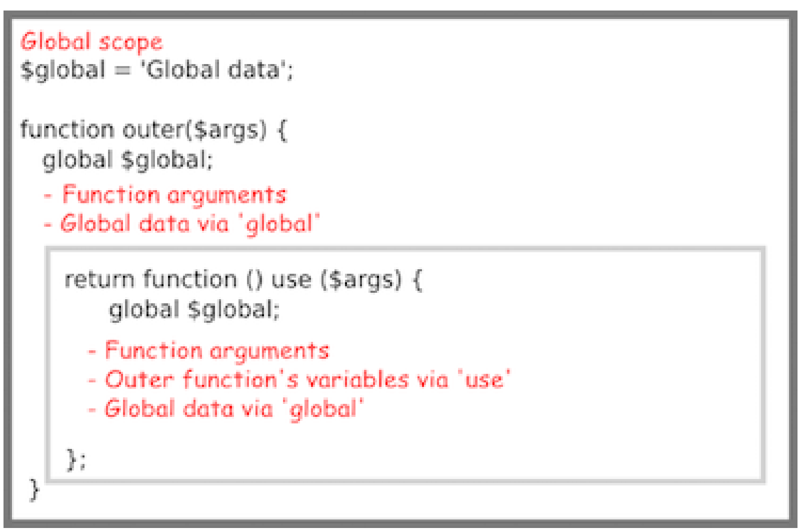
```
Route::get('/item/{id}', function ($id) {
  return Item::findById($id);
});
```

But this doesn't apply exclusively to the anonymous case, all functions create a closure. Unfortunately, at this point PHP doesn't have support for "lambda expressions," syntactically simpler anonymous functions, but there's currently an open proposal that I hope will be included soon. Check out this RFC here[7] for more information.

With lambda expressions and omitting type declarations, the code above would look pretty slick:

```
apply2(function ($a, $b) => SafeNumber::of(empty($b) ? NAN : $a / $b))(7, 2);
```

Similarly, lambda functions would also be implemented to form a closure around it. Let's understand what this means. A closure is defined as the lexical scope created around a function declaration, anonymous or otherwise. This gives function knowledge of the state of the variables lexically declared around it. The term lexical refers to the actual placement of the variable in the code. Here's a diagram that explains what a function inherits:



[7]https://wiki.php.net/rfc/arrow_functions

The data structure used to encapsulate the state that makes up a function's closure is the `[Closure class]`(http://php.net/manual/en/class.closure.php) class, hence the name. Here's the definition for it:

```
class Closure {
  private __construct ( void )
  public static Closure bind ( Closure $closure , object $newthis [, mixed $new\
scope = "static" ] )
  public Closure bindTo ( object $newthis [, mixed $newscope = "static" ] )
  public mixed call ( object $newthis [, mixed $... ] )
}
```

Recall the function `concatWith` used earlier, and look closely at the inner function:

```
function concatWith(string $a): callable {
  return function (string $b) use ($a): string {
    return $a . $b;
  };
}
```

PHP's closure mechanism has an advantage over that of JavaScript's in that, instead of simply capturing all of the state around an anonymous function declaration, you can explicitly declare which variables the it's permitted to access via the `use` keyword.

Hint:`use` does not work for functions in the outer scope.

This is incredibly powerful and allows you to have more control over side effects. Under the hood, the variables passed into `use` are used to instantiate its closure.

Here are some examples that reveal PHP's internal closure object. The `add` function can be invoked as globally using `Closure::call`[8]:

```
$add->call(new class {}, 2, 3); //-> 5
```

The use of PHP 7's anonymous class here is to set the owning class of this function (in case `$this` is used) to something concrete. Since I don't use `$this`, I can provide a meaningless object to receive `$this`.

You can also dynamically bind functions to object scopes using `Closure::bindTo`[9]. This allows you to easily mixin additional behavior of an existing object. As example, let's create a function that

[8]http://php.net/manual/en/closure.call.php
[9]http://php.net/manual/en/closure.bindto.php

allows us to validate the contents inside a container before mapping an operation to it. In functional programming, conditional validation is applied using an operator called `filter`, just like `array_filter`[10] is used to remove unnecessary elements. But instead of adding `filter` to all containers, I can mixin a properly scoped closure (in some ways this is similar to using `Traits`[11]):

```
function addTo($a) {
	return function ($b) use ($a) {
		return $a + $b;
	};
}

$filter = function (callable $f): Container {
	return Container::of(call_user_func($f, $this->_value) ? $this->_value : 0);
};

$wrappedInput = Container::of(2);

$validatableContainer = $filter->bindTo($wrappedInput, Container);

$validatableContainer('is_numeric')->map(addTo(40)); //-> 42
```

Now consider what happens when the input is invalid according to our filter predicate:

```
$wrappedInput = Container::of('abc);
$validatableContainer('is_numeric')->map(addTo(40)); //-> 40
```

Now that you've learned about higher-order functions (and then some more), we're ready to begin creating functional programs. In chapter 3 we'll learn about function composition and point-free programs, as well as how to break problems apart into manageable, testable units.

Lastly, if you have an OOP background, here's a nice table to keep handy that compares both paradigms which will make the rest of the material easier to digest.

Functional vs Object-Oriented Summary Table

[10] http://php.net/manual/en/function.array-filter.php
[11] http://php.net/manual/en/language.oop5.traits.php

Traits	Functional	Object-oriented
Unit of composition	Functions	Objects (classes)
Programming style	Declarative	Mostly imperative
Data and Behavior	Loosely coupled into pure, standalone functions	Tightly coupled in classes with methods
State Management	Treats objects as immutable values (minimize state changes)	Favors mutation of objects (state) via methods
Control flow	Higher-order functions and recursion	Loops and conditionals
Tread-safety	Enables concurrent programming	Difficult to achieve

Point-free Data Flows and currying

Under Construction

Error handling with Monads

Under Construction

React: The Promise Monad

Under Construction

Reactive Programming in PHP

Under Construction

Functional Domain Modeling

Under Construction

Object-oriented in the large, functional in the small

Under Construction