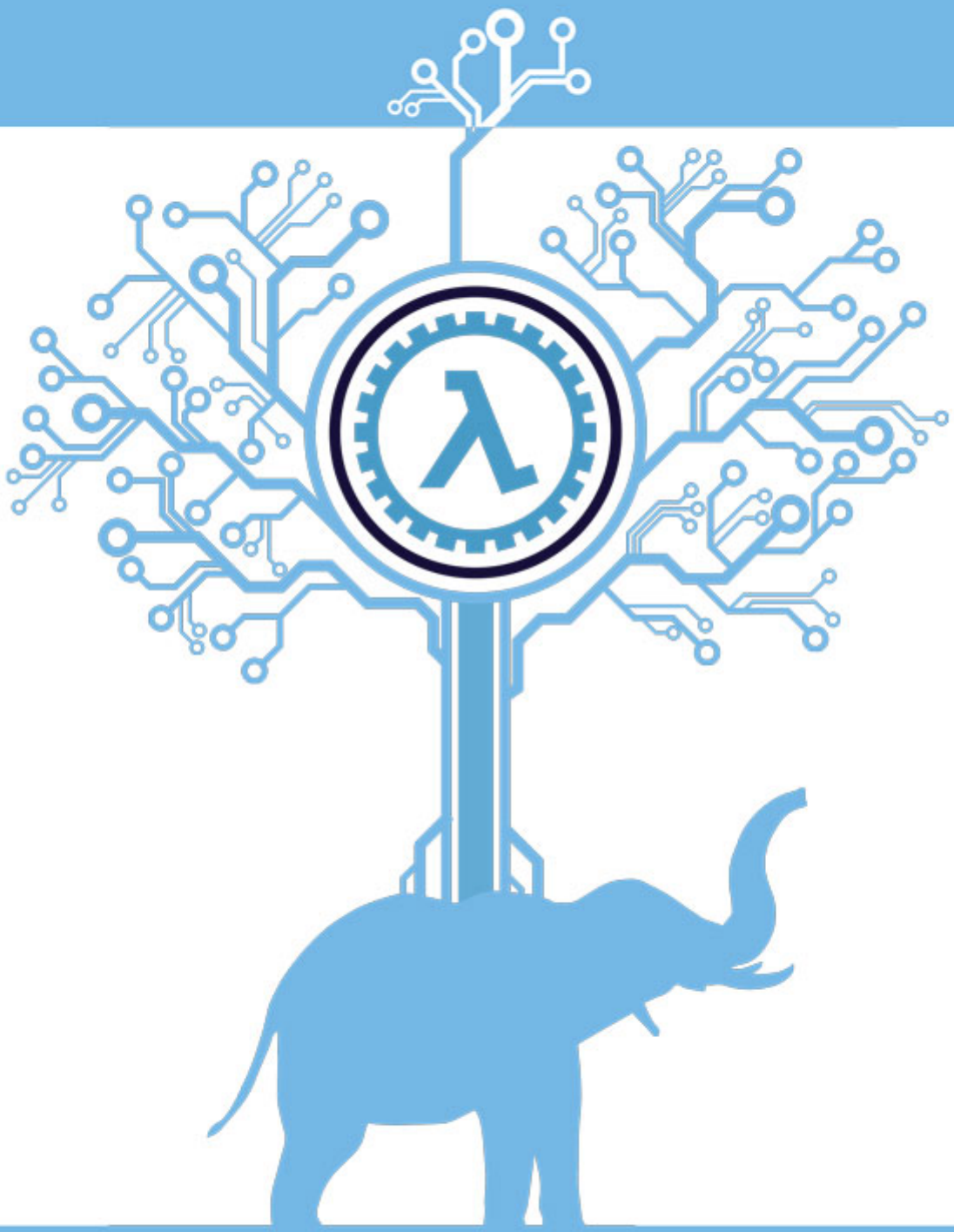
FUNCTIONAL PROGRAMMING IN PHP

A simple, concise primer for building functional apps in php



Lochemem Bruno Michael

Contents

Functional Programming in PHP: A Simple, Concise Primer 4

About the Author 5

About the Reviewers 6

Acknowledgement 8

Book Contents 9

Requirements 11

Target Audience 12

Preface 13

An Introduction to Functional Programming in PHP 14

A Brief History of PHP 14

A Brief History of Functional Programming 15

Defining Functional Programming 16

Functional Programming is a Declarative Paradigm 17

The Benefits of Functional Programming 18

Relevance to a PHP Developer 19

Functional Programming in Other Contexts 20

PHP is Getting Better 23

Summary 24

Functional Programming Core Concepts **25**

Functions 25

Message to Prospective Readers 29

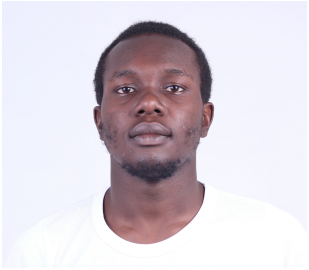
Functional Programming in PHP: A Simple, Concise Primer

This book is available for sale at https://leanpub.com/functionalprogramminginphp.

This version was published on 2020-10-12.

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.

About the Author



Lochemem Bruno Michael, an alum of the United States International University - Africa, is a Software Engineer and occasional Technical Writer from Kampala, Uganda. Mr. Lochemem is proficient in PHP, C++, and JavaScript and loves to develop open-source software in the aforestated languages. His portfolio includes Functional Programming libraries in PHP and JavaScript - respectively named bingo-functional and bingo-functional-js, as well as C++ extensions for the PHP language: php_trie, extjwt-cpp and couchdb-ext.

Aside from writing code, Mr. Lochemem writes articles he publishes to his blog hosted on Medium and is profoundly interested in many things: rap music, basketball, soccer, books, video games, as well as podcasting to mention but a few.

About the Reviewers

Cees Jan Kiewiet

Mr. Cees-Jan Kiewiet is a Senior engineer at Usabilla and core maintainer of React-PHP. Based in the Netherlands, he has over fifteen years of programming experience and is proficient in, besides PHP, Golang, and Java. Mr. Kiewiet loves to play video games, meet with developers at meetups, and bask in the tranquility of his garden when he is not breaking servers.

Chun Sheng-Li

Mr. Li is a Taiwanese engineer who holds both a Master's and Bachelor's degree in Computer Science. He is an open-source enthusiast and avid contributor to multiple projects. Mr. Li has many programming interests that include web security, JavaScript, and Blockchain. He currently works as a backend engineer in his native Taiwan.

Acknowledgement

There are many deserving of praise for this monumental effort - my maiden foray into authorship. First, the reviewers of the book whose secondary appraisal of the material is more than apt. Second, the developers I collaborate with: the interactions I have with them has only expedited the finality of this outcome. Third, the creator of the book artwork, Sam Njuguna, a talented visual artist and USIU-Africa alum. Fourth, my mentors Paul Bombo, and Dr. Patrick Kanyi Wamuyu. The former is a cerebral engineer and visionary whose ideas I have developed a reverence for in the three years I have known him; the latter is a college professor whose teachings have proved to be insightful. Fifth is USIU-Africa staffer, Dan Muchai - a wise polymath and good friend.

Finally, my family - all two parents and three siblings of it. I appreciate all the encouragement they have offered - financial and emotional - since I embarked on writing the book. I am grateful for the concern my father, Prof. Bruno Ocaya, sisters, Consolate, Faith and Frances, have shown throughout the writing process and the unrelenting support my mother, Bernadette Ocaya has imbued in nurturing my nascent potential.

Book Contents

An Introduction to Functional Programming in PHP, provides a brief history of PHP, a definition of Functional Programming, an explanation of the perks on offer, and spells out its relevance to a PHP developer.

Functional Programming Core Concepts, brings to focus the idea of writing pure functions, immutability as a concept, and grading functions based on referential transparency.

Composition and Helper Functions, serves as a detailed exploration of function composition. It also provides commentary on transducers, and the map, filter, and fold operations - for manipulating list data, as well as helper functions whose utility is their conceptual abstraction.

Error Handling in Functional Programming, showcases the weaknesses of PHP exceptions and highlights the pros and cons of alternative flow control mechanisms - default values, callbacks, and error messages. Furthermore, the chapter demonstrates the use of Sum types - Either and Maybe - as additional alternatives to exceptions.

Functors, focuses on the Functor construct. It offers an insight into Category Theory, demonstrates how to roll out functors in PHP, and explains monads of various kinds - IO, State, Reader, and List.

Parallelization of tasks in PHP, offers insights into achieving concurrency via task parallelization in PHP using RabbitMQ, PCNTL routines, and the PECL pthreads

extension.

Recursion, Pattern Matching, and Property Testing, demonstrably explains the aforestated concepts as additional techniques worth using in PHP.

A Simple Project, is a walkthrough on how to create a simple console phonebook application in the Functional Programming style.

Requirements

Principally, many versions of PHP, supported and otherwise, can aid in making the most of the material in this opus. More recent releases of the language (7 and newer) are, however, better suited to applying Functional Programming principles as they are better maintained and widely industry-prescribed. I, therefore, recommend that you the reader install any of the newer supported releases of the language. Consider downloading Composer, the de-facto dependency manager for PHP, to go with your chosen PHP variant to operationalize all the programs in the book that warrant its usage.

I urge that you give heed to the varied chapter requirements from Packagist-hosted, Composer-installable libraries to PHP language extensions and job worker tools. The code that features throughout the volume is also available for download: it is hosted on Github in the repository fp-php-book.

Lastly, I advise that you have an arbitrarily chosen dependable text editor installed on your system to cater for any tinkerings with any of the code in the subsequent text.

Target Audience

I have written this book with intermediate to advanced developers with PHP experience in mind. Better suited for the contents of this book are those interested in adopting a new paradigm, the Functional Programming style.

Preface

Functional Programming, a paradigm whose teachings are rooted in academia, is rapidly soaring in popularity. The implementations of Functional Programming's many tenets and principles are just as vast as they are varied. Pure functions, state containers, monads, and even thunks are a staple of various programming languages as either organic features or syntax embellishments. The dramatic rise in the popularity of the paradigm is not without a certain degree of polarization - one premised on the underlying complexity of the paradigm's offerings.

This book is an attempt to break down the arcane verbiage of the Functional Programming discourse while demonstrating its effectiveness in writing readable, parsable PHP code. PHP is still really popular and has a nexus of communities with varied interests that galvanize around many paradigms, mostly those not exclusively of the Functional Programming ilk. Functional Programming is, as emphasized in the previous paragraph, not only possible in PHP because of its various long-standing features and all-around improvements, but also adaptable as a cognitive-burden reducer. I hope that the eight chapters of this volume prove to be more enlightening than uninstructive and dispel various fallacies associated with Functional Programming while advertising the robustness of PHP.

An Introduction to Functional Programming in PHP

A Brief History of PHP

PHP is, depending on your proclivities and general programming knowledge, a decent scripting language for the web. Though some might insist that its relatively gentle learning curve and somewhat casual syntax are emblematic of the language's poor design, PHP remains a language whose depths are worth exploring.

Once upon a time, a twenty-six-year-old programmer who had nothing better to do with his life than write C (mostly because it was normative to do so then), put together CGI scripts in the language to create a templating engine that could run on a web server. This templating language, PHP, soared in popularity shortly after its inception because of its intuitive syntax - pseudo-developers were emboldened as a clamor for more features ensued.

The desired features extended the scope of the templating language and once implemented, rendered it business-logic affable. PHP has, over the years improved through each versioned iteration - a progression from version 2.0 to 3.0, co-authored by Andi Gutmans and Zeev Suranski which looks a lot more like the version written today was followed by transitions from version 4.0 to 5.0 and then to 7.0 onwards.

Interesting to note is that PHP did not have a proper specification till 2014. A specification, a canonical blueprint containing natural language, mathematical, as well as programmatic expressions of a language's capabilities is vital to the programming lifecycle. PHP's specification can be found online, at https://php.net.

A Brief History of Functional Programming

The roots of Functional Programming are traceable to the seminal works of scientists Alonzo Church and Alan Turing. Church's Lambda Calculus and Turing's Universal Machine - both formalized sometime in the 1930s - form the very foundation of Functional Programming. Church advanced - through his Calculus, a proto-Functional syntax: one that emphasizes the principles of abstraction and generalization. Turing's conceit was an emulation of a computing device capable of transforming calculable inputs - typically those used in human-performable logic operations - into outputs.

Machine-aided programming soon followed the advancement of the aforedescribed ideas as Lisp - the first modern Functional language developed circa 1958 - heralded a new epoch of using functions and algebraic structures to define computer programs. Considered to be the ancestor of most modern Functional languages - Clojure, Erlang, and Scheme to mention but a few - Lisp continues to exist today. Its effervescence, and Functional Programming's as a whole, diminished somewhat in the period that followed.

In the 70s, when more procedural and Object-Oriented strategies became more commonplace, Functional Programming ceded its popularity to the likes of C and Smalltalk. The popularity of Functional Programming only continued to decline in the following decades as runtimes like C++ and Java made significant headway in programming circles. The paradigm was, before a more recent resurgence, effectively pushed to the mainstream periphery and academia. The latter is the arena in which Haskell - arguably one of the more notable pure Functional programming

languages, named after famed Mathematician Haskell Curry - was developed.

FP has recently experienced a favorable uptick in prominence - as everything from a mention of a paradigm technique to the integration of full-blown conceptual frameworks centered around more elaborate paradigm ideas have started to feature more. There is likely some sort of Functional Programming-associable library present in your favorite and second-favorite programming language's ecosystem - a signifier of a rekindled interest in expressing programs as a juxtaposition of function composites and immutable state.

The legacies of Alonzo Church's Lambda notation and Turing's analytical machine live on - universally. In PHP, in particular, the first-class function support and expressive syntax align well with the conceit of capturing the computational aspects of functions.

Defining Functional Programming

Functional programming is a programming paradigm. A paradigm is an approach to doing something, which in this case, is writing code. Unique to Functional Programming is the notion of writing pure functions - those free of side-effects and side-causes (more on this later).

Functions are at the epicenter of Functional Programming and are therefore a significant denizen. Functional programming has many siblings: among them are Object-Oriented programming (OOP), imperative programming, declarative programming and reactive programming. Each paradigm in the programming nexus has unique, salient characteristics. To put this into perspective, consider all of Functional Programming's siblings: OOP is class and object-driven, imperative programming is side-effect inducing-statement-based, declarative programming's predicate is action statements, and reactive programming is asynchronous-stream oriented.

Computation is, in Functional Programming, premised on transforming state, the values of all programming variables at a given point in time, through combining mathematical functions. Abstraction, a means of tidying away things not needed, is a focal point of Category Theory from which Functional Programming draws a good number of its axioms and Mathematics in general. Contextualize a problem as trivial as adding two numbers. The problem is computing the sum of two operands which can be solved by abstracting away the arcane addition statement. How is this abstraction achieved? By using a function.

```
function add(int $x, int $y): int
{
  return $x + $y;
}

add(1, 2); // evaluates to 3
```

Functional programming is, therefore, a paradigm predicated on using mathematical functions to evaluate state.

Functional Programming is a Declarative Paradigm

The lingua franca of Functional Programming is expressions. Expressions are a hallmark of declarative programming, which, itself, is a paradigm centered around describing actions and not necessarily their underlying intricacies - their inner workings. Declarative programming is, therefore, a paradigm of abstraction: it conditions those who use it to focus more on what they intend to achieve and not necessarily entirely on the constructs that constitute the desired actions.

Please do not conflate abstraction with a total lack of complexity, as that would be misconstruing the idea altogether. Complexity, according to the concept, should be neatly tidied away and not removed from code as that would defeat the purpose

of writing operational programs with real utility.

Expressions are blocks of code that evaluate to a value. They typically take on the form of one of either function calls, or operations with any number of operands. It is possible to write many expressions in PHP: the following are examples.

```
2 * 2;
array_filter(range(1, 5), fn (int $x): bool => $x % 2 == 0);
'foo' . 'bar';
SplFixedArray::fromArray(['foo', 'baz']);
```

Expressions are storable in memory: by placing them in some container identifiable by name. In fact, this is something you probably do all the time and a practice you should not cease doing when writing programs in a Functional Programming style.

The Benefits of Functional Programming

Functional programming reduces the cognitive burden of writing code. The mental effort required to perform tasks is lessened by composing pure functions.

There exists a variably small, fixed number of shared memory resources in the human brain for discrete working memory quanta. Working memory, that is, short-term memory optimized for transient storage, is a limited resource. Each variable written in a computer program occupies at least one quantum of available memory. In the snippet below, there are three variables declared and evaluated by the PHP interpreter.

```
$first  = 2;
$second = 3;
$third  = $first + $second; // evaluates to 5
```

Functional programming alleviates the burden of processing variable information. A simple function can suffice to this end.

```
1 $add = fn (int $x, int $y): int => $x + $y;
2
3 $add(2, 3); // still evaluates to 5
```

The anonymous `add` function assigned to the `add` variable eliminates the need for the variables named first and second - the lambda function above effectively frees up working memory and renders the problem of adding two numbers tractable. An article written by Eric Elliott, a brilliant technologist, clearly highlights the ameliorative powers of Functional Programming - I suggest you peruse it.

Naturally, cognitively amenable code is easy to evaluate, transform, and test. Furthermore, functional code is easy to split among multiple processors the goal of which is often concurrent parallelization (see Chapter 6). It is therefore clear that Functional Programming's many advantages stem from its ability to attenuate cognitive load.

Relevance to a PHP Developer

Object-Oriented Programming (OOP) is the most widely acknowledged paradigm in PHP, and is firmly embellished in PHP programming canon. The Object-Oriented style is effective though subtly underconceptualized and is a nuanced offshoot of Functional Programming (more on its seminality in chapter 2). The style does not fundamentally antagonize the Functional Programming ethos, and can, with more studious interrogation, offer a gateway into Functional Programming.

Further still, PHP like most multi-paradigm runtimes, not only has first-class function support, but also simulates a Turing machine (see Appendix): it is an apt enabler of Functional Programming. The major implication of first-class citizenry is multiplicity-of-use. PHP's functions, discussed in great detail in Chapter 2, are manipulable to the extent of being usable in a variety of contexts: as arguments and return values as well as variable-assignable entities. Being Turing complete means that

PHP allows for logic construction via formalized syntax: the language is, by design, Functional Programming-compliant.

Functional programming is no novelty but has rapidly increased in popularity in recent times - it is in the current zeitgeist in programming. Programmers whose preferred programming arsenal features non-functional languages like PHP have, in recent times, adopted the Functional Programming paradigm to the extent of writing Functional Programming-affable libraries and toolkits - everything from simple helper function libraries to pattern-matching tools and interactive REPLs.

Said tools mirror the functionality of functional languages and are, in many ways, embellishments of the Lambda Calculus: the said faculties continue to pervade various eco-systems, reaching multiple programmers and impacting many product design decisions. The various PHP-exclusive Functional Programming toolkits set to feature throughout the text should prove immensely useful in harnessing the paradigm's cogencies.

Also, the litany of Functional Programming perks enumerated and alluded to in the previous section is another definite impetus for exploration of Functional Programming's offerings. Not only is Functional Programming a different approach to writing code, but it is also soundly organized.

Functional Programming in Other Contexts

This sub-chapter might be a big red herring considering the focus of the book is PHP. I do, however, intend to by showing how the paradigm has permeated through other languages, the popularity of Functional Programming.

JavaScript, yet another scripting language for the web, has soared recently in popularity. Many web utilities are either languages that transpile to JavaScript or frameworks written in idiomatic JavaScript. PureScript, a functional language, is Haskell-esque and compiles to JavaScript. As of the time of writing, the language's

GitHub popularity stands at over six-thousand stars. PureScript code looks a lot like this:

```
import Prelude
import Control.Monad.Eff.Console (log)

greet :: String -> String // greet function type signature
greet name = "Hello, " <> name <> "!" // greet function declaration

main = log(greet "World") // prints "Hello World!"
```

Redux, a predictable state container for JS applications, follows an action define-dispatch pattern which eases the cognitive burden. Though Redux has a quirky propensity for getting verbose, it remains a potent state management tool. Written by Dan Abramov, Redux, a tool modeled to mimic the likeness of Elm and Flux architectures, has a popularity which stands - at the time of publishing - at over fifty-thousand GitHub stars.

```
import { createStore } from 'redux'

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

```

```
let store = createStore(counter)
store.subscribe(() => console.log(store.getState()))
store.dispatch({ type: 'INCREMENT' })
```

Python, a multi-purpose scripting language, also has support for Functional Programming. The potency of Python is such that it offers multi-paradigm support in addition to powering a multitude of technologies like Reddit and Spotify. As far as Python is concerned, utilities like the pytoolz Functional Programming library are particularly intriguing. The allure of pytoolz and its conceptual counterparts like underscore.js and Ruby's enumerable is the classification of actions - function genealogies based on iterables, higher-order functions, as well as dictionaries are immanent in pytoolz's design. The boilerplate below is that of a simple curry action:

```
from toolz import curry

@curry
def multiply(x, y):
    return x * y

double = multiply(3)
print double(3) // returns 9
```

It is impossible to exhaust the number of Functional Programming utilities in non-PHP contexts in a book like this one. This chapter, though somewhat limited in scope, reinforces vital points: Functional Programming's ubiquity and the universality of its underlying predicates. If you are still not convinced by the promise of the paradigm's cognitive load-reducing abilities, the existence of Functional Programming tools in other languages should be the impetus for engagement.

PHP is Getting Better

If you are reading the third version of this opus, you are probably already familiar with PHP versions 7.4 and newer. The performance and syntax improvements in the aforestated versions are proof of the language's continued improvement. As regards Functional Programming, PHP's new short closure syntax, first-class callable syntax, and typed properties present the most straightforward value. Short closures, in situations that warrant their usage, reduce the verbosity of program code. First-class callable syntax makes shuttling functions with multiple arguments an easier proposition. Typed properties enforce strictness inside class contexts and enhances class code readability. Other features like preloading and the Foreign Function Interface (FFI) present performance improvements and might require finesse to operationalize.

```
1 // class with typed properties
2 class User {
3   public string $username;
4   public int $userId;
5 }
6
7 // array filter operation with arrow-functions
8 $even     = array_filter(range(1, 10), fn (int $x): bool => $x % 2 == 0);
9
10 // array concatenation via unpacking (works with arrays with string keys in
       versions 8.1 and newer)
11 $merged   = ['foo', ...['bar', 'baz' => 'baz']];
```

Summary

Functional Programming, a cognitive load-truncating paradigm premised on using pure functions, has a place in PHP's vast ecosystem. The language, in its multi-paradigm existence, offers a runtime with first-class function citizenry, and has grown significantly since its inception in 1994 to provide more faculties for Functional Programming.

The paradigm is itself a long-standing approach - one which emphasizes using expressions - to architecting computer programs with teachings seminal in the works of individuals like Alonzo Church and Alan Turing. Functional Programming has pervaded through scripting languages similar to PHP - JavaScript, Ruby, and Python - and has, in so doing, proved a viability outside of academia.

The material in all subsequent chapters is pertinent so far as to familiarize you, the reader, with Functional Programming constructs - everything from the rudiments to more advanced constructs - and their application in PHP.

Functional Programming Core Concepts

Already established is the ethos of Functional Programming - central to the paradigm are pure functions, immutability, and referential transparency. This chapter is an elucidation of Functional Programming's nuts and bolts.

Functions

Functions are at the epicenter of Functional Programming. They are ideal for the paradigm because they do not possess a history of any kind and have a default invariable status. Functions are relationships between inputs and outputs - they are composed of a signature, and posses an arity. Functions are predisposed to accept arguments, values intended for transformation, the number of which is the arity. A function signature is a stipulation of return types, argument types, as well as the encapsulated function logic. Mathematics enables one to define functions. The snippet below shows how:

$$f(x) = x + 1$$

$$f(2) = 3$$

$$f(3) = 4$$

The function in the snippet which has an arity of 1 is one which increments the value of the parameter x by 1. x is effectively a placeholder for the transformation predicated on the addition operation defined. Different results are obtained for different values of x because the function offers localized computation. Each result is, therefore, unique to each input.

In programming, in particular, PHP programming, functions take on many forms. Functions in PHP are just as malleable as variables and have a similar characteristic: the keyword function, two sets of parentheses, round brackets for the arguments and curly braces for the function logic, as well as a return type hint as of PHP version 7.0. The different types of functions in PHP are the ubiquitous named functions, anonymous functions, and methods.

```
1 // php function
2 function subtract(int $x, int $y) : int
3 {
4     return $x - $y;
5 }
6
7 subtract(3, 2); // evaluates to 1
```

Named functions in PHP like the one above are no different (outside of a few syntactic differences) from those defined in other programming languages. They typically feature an arbitrary name sandwiched between a function keyword and an argument list followed by a discretionary signature whose constituents are state transformation artifacts.

Anonymous functions, unnamed functions or lambdas, are, on the other hand, descendants of the PHP Closure class and are often bound to variables or placed

inside other data structures. Useful as callback functions, anonymous functions have immense inline-function value and have every syntactic feature of named functions except for the explicit name.

```
// php anonymous function
$subtract = fn (int $x, int $y): int => $x - $y;

$subtract(3, 2); // evaluates to 1
```

Closely related to the anonymous function is the closure, an anonymous function with access to an external scope. Closures are particularly helpful in situations that require interaction with values defined outside an anonymous function's internal environment. The `use` keyword, though regarded by some as a feature that introduces unnecessary verbosity, enables the aforestated outward state access.

```
$divisor = 4; // value in external scope

// php closure
$divide = function (int $dividend) use ($divisor) : int {
    return $dividend / $divisor;
};

$divide(12); // evaluates to 3
```

The last type of function in PHP is the class method encapsulated within the scope of a class. Classes are important in Functional Programming: their significance is the focus of chapters 4 and 5.

```
class Operations
{
    private $x;

    private $y;
```

```
6
7     public function __construct(int $x, int $y)
8     {
9         $this->x = $x;
10        $this->y = $y;
11    }
12
13    public function add() : int
14    {
15        return $this->x + $this->y;
16    }
17 }
18
19 $op = new Operation(1, 2);
20
21 var_dump($op->add()); // evaluates to 3
```

Setters are absent in the snippet above. The toxicity of setters lies in their high propensity for function impurity. Henceforth, all ideas of class setters should be jettisoned....

Message to Prospective Readers

This segment concludes the sample opus. If you feel enthused enough by the subject matter in this book to buy the full edition, please proceed to do so. Regardless of the purchase decision you make, I hope the elucidatory text you have read thus far provides the impetus for further engagement with the book's contents and the paradigm as a whole.

Buyers of the full version will get access to the remainder of the volume that offers insights into Composition, paradigm-affable error-handling, Category Theory, Functors, and much more.

Thank you for making it to this point in the text.

Best regards,

Lochemem Bruno Michael