THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Typed PHP

Stronger Types for Cleaner Code

Christopher Pitt

Apress®

Typed PHP

Stronger Types for Cleaner Code

Christopher Pitt

Apress®

Typed PHP: Stronger Types for Cleaner Code

Christopher Pitt
Cape Town, Western Cape
South Africa

ISBN-13 (pbk): 978-1-4842-2113-6 ISBN-13 (electronic): 978-1-4842-2114-3
DOI 10.1007/978-1-4842-2114-3

Library of Congress Control Number: 2016948754

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Tri Phan
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com/9781484221136. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Thank you, Squirrel.

Contents at a Glance

Contents

About the Author

Christopher Pitt is a developer and writer, working at SilverStripe. He usually works on application architecture, although sometimes you'll find him building compilers or robots.

About the Technical Reviewer

Tri Phan is the founder of the Programming Learning Channel on YouTube. He has over seven years of experience in the software industry. Specifically, he has worked in many outsourcing companies and has written many applications of many fields in different programming languages such as PHP, Java, and C#. In addition, he has over six years of experience in teaching at international and technological centers such as Aptech, NIIT, and Kent College.

Acknowledgments

I'd like to thank the team at Apress for whipping this up so quickly. It's been a brief, pleasurable experience working with them.

I'd also like to thank the original technical reviewers for their time, effort, and wisdom. Thanks to everyone who bought the original version. Without all of you, this book would not have happened.

Introduction

PHP is a powerful, general-purpose programming language, with a dynamic type system at its core. It has been used by a generation of programmers to create much of the Internet we see today.

It has also been the subject of ridicule, thanks to what some consider to be an inconsistent and incomplete set of core functionality.

This book seeks to address those issues, by building a stronger type system. It's a type system that parallels more modern programming languages and best practices.

As we step through the core PHP functionality, we'll identify inconsistencies and shortcomings, while also exploring viable approaches to overcoming them. We'll build a standard library of our own and see how a number of open source extensions can bring these libraries in harmony with the built-in data types.

Why Write This Book

The purpose of this book is to simplify how we work with strings, numbers, arrays, etc. They're called *Scalar types,* because PHP treats them differently to objects. They have no properties or methods.

PHP has a rich history and a dominant place on the web. It has achieved much despite language inconsistencies and difficulties. Bjarne Stroustrup once said, "There are only two kinds of languages: the ones people complain about and the ones nobody uses". PHP is one of those languages that *everybody* uses, yet that's often seen as a good reason to ignore the bad parts and just get stuff done.

I'm all for getting stuff done, and to that end I have used PHP for many years. It's always bugged me how procedural PHP is, in an ecosystem of OOP libraries and frameworks. So I decided to take a deeper look at building a stronger type system on top of PHP.

In this book, we'll look at how to use *standard* PHP libraries. We look at user-land libraries. We look at using extensions and cross-compilers. All this will contribute toward creating a set of reusable tools that unify and ease the scalar types of PHP.

Who This Book Is For

This book assumes you have working knowledge of PHP. That means you understand the basics of programming and have already used them to write PHP code.

You don't need to know how to set up a PHP stack. We will cover how to do this, using VirtualBox, Vagrant, and Phansible.

You also need to have an open mind. Many of the concepts covered in this book are experimental and none of them is commonplace. That's not to say that you can't use these techniques in production applications. You just need to decide if they are a good fit for your architecture.

Finally, you should have access to a decent Internet connection. The examples in this book work best inside a Vagrant virtual machine. Vagrant creates development environments and it needs to download extra software. This can take a long time on a slow connection.

CHAPTER 1

The State of PHP

PHP has a long and colorful history. Over the years it has become the foundation for the Internet. Along the way it has accumulated many syntactic and semantic oddities, which make it slightly harder to learn and predict.

This book is an exploration of what PHP's type system and standard libraries could become, given the chance to start fresh. But in order to fully appreciate the difference, we need to take a look at what PHP is like today. Only then will we be able to consider what changes would improve it.

In this chapter, we'll take a look at the dichotomy of classes and global functions, and the small irregularities that make the core functions difficult to remember and reason about.

Procedural versus Object Oriented

Procedural and object oriented are programming styles that approach program execution in different ways. It's good to understand how they work and how they define the state of scalar types.

Procedural Programming

Procedural programming describes a top-down approach to program execution. That is, procedural programs consist of a list of steps for the interpreter to take (from top to bottom).

In pseudo-code, a procedural image resize program may resemble these steps:

1. Start execution.
2. Then store a file reference returned by an `open_file` function.
3. Then store a modified image returned by a `resize_image` function.
4. Then close the open file.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2114-3_1) contains supplementary material, which is available to authorized users.

C. Pitt, *Typed PHP*, DOI 10.1007/978-1-4842-2114-3_1

5. Then store a file reference returned by an `open_file` function.
6. Then write the modified image data to the second open file.
7. Then close the open file.
8. Then empty the modified image variable.
9. End execution.

Procedural programs can call on functions (as described in the example) and can define functions. We can change the current line with things like loops and go to statements, but the program is mostly just a set of instructions.

Object Oriented Programming

Object oriented programming is a way to describe programs as interactions between different objects. These objects can have any combination of properties and methods.

Properties are another name for variables belonging to objects. Likewise, methods are another name for functions belonging to objects.

These methods are still executed from top to bottom, but programs still depend on object interaction. In pseudo-code, an object oriented image resize program may resemble these steps:

1. Start execution.
2. Then create a file object.
3. Then create an image resize object.
4. Then pass the file object to the image resize object.
5. Then create another file object.
6. Then write the result of the image resize object's `resize` method to the second file object.
7. Then close the second file object.
8. Then destroy the second file object.
9. Then destroy the image resize object.
10. Then destroy the first file object.
11. End execution.

Which Is the Best?

This is the wrong question to ask. Both have their strengths and weaknesses. Object oriented programming can lead to more code than procedural programming. Object oriented programming lends itself to better separation of concerns.

Classes (the blueprints on which new objects are based) provide a place to hide reusable, private behavior. Imagine we have a `Transaction` class, which has the job of accumulating the details of items being sold. Now imagine we need to calculate, along with the base cost of each item, the tax and total cost of the transaction.

We could reuse such a function many times, but it may not be relevant to any other part of our application. In that case, it makes sense not to expose it to every other part of our system. It can just be a private method on the `Transaction` class. Classes help to encapsulate the responsibilities of objects.

When we want to change how transactions work, we can tinker around with the private functions in the `Transaction` class, and so long as the expected input and output formats are the same, no other part of our system is the wiser.

Without classes and objects, every function is public, and can be used from anywhere. That has the potential for chaos.

Stated another way: It's easier to think of objects and how they interact with each other than to think of the whole flow of a program as a complex list of instructions. In the end, procedural code can be as clean as object oriented code, but it takes a lot more work.

> *"Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function."*
>
> —John Carmack

Which Is PHP?

PHP is procedural. That's how Rasmus built it and that's how it has stayed. The distinction is clear, when it comes to dealing with scalar type variables. Scalar types are not objects. They don't have methods or properties. If you want to do something to a PHP scalar type variable (string, int, float, or bool), you pass it to a function.

Before PHP 5.3, there were no namespaces. As a result, these type-specific methods are in the global namespace. They are available everywhere and, as we're about to see, they are inconsistent. This makes scalar type code ugly code.

Native Function Inconsistencies

PHP is often decried because of the inconsistencies in the native functions. PHP is often praised because of the quality of the documentation. These are related! The documentation has evolved so well because native PHP functions are inconsistent.

This section is going to make me sound like a PHP hater. That couldn't be further from the truth! I love PHP and I'm committed to using it and helping others use it. To know what we're building, we have to know what we're trying to avoid building. That's the point of what's to follow.

Sporadic Underscores

Underscores are used sporadically throughout the core function library. It's difficult to remember (without the help of an integrated development environment) whether you need an underscore when you want to strip tags or strip slashes in a string.

- `parse_str`
- `printf`
- `str_pad`
- `strcmp`
- `strip_tags`
- `stripslashes`

These functions are described at `http://php.net/manual/en/ref.strings.php`.

The full list contains 98 functions, 30 of which use one or more underscores. Sometimes functions clearly composed of many full words (like `setlocale`) don't have underscores. Sometimes functions that do almost exactly the same things (`strlen` vs. `str_word_count`) are inconsistently named.

Sporadic Abbreviation

Most of the string functions use abbreviations of some kind. This is fine when applied consistently. Yet, the inclusion of a few non-abbreviated functions makes the string API difficult to memorize.

- `addslashes`
- `chr`
- `htmlentities`
- `lcfirst`
- `number_format`
- `stroll`

This often means a round-trip to the documentation.

Inconsistent Argument Order

Argument order is different, depending on the array and string functions.

- `array_key_exists($needle, $haystack)`
- `stripos($haystack , $needle)`

Rasmus explains this as a result of keeping as close to the underlying C libraries as possible. The problem with this explanation is that it means nothing to developers who have never worked with C and just want to work with PHP.

Array methods are needle/haystack and string methods are haystack/needle.

Regular Expression/Strings

PHP represents regular expressions as strings with a completely different set of functions.

- `preg_filter`
- `str_replace`
- `preg_match`
- `strstr`
- `preg_split`
- `explode`

It would be better if they looked different and had their own methods. Either that or the string methods should work for regular expressions also.

PHP assumes a string is a regular expression if it starts and ends with recognizable delimiters. You can learn more about that at http://www.php.net/manual/en/regexp.reference.delimiters.php.

Regular expressions could have their own representation, which separates them from strings. An example of a language that already has this is JavaScript:

```
"abc".replace(/b/, "123"); // "abc" becomes "a123c"
"def".replace(/[e]/, "456"); // "def" becomes "d456f"
```

These languages make clear the distinction between strings and regular expressions.

Nouns/Verbs

Some of the native functions are verbs (like `echo` and `parse_str`), while others are nouns (like `htmlentities` and `soundex`).

- `echo`
- `htmlentities`
- `lcfirst`
- `md5`
- `parse_str`
- `soundex`

This makes it tricky to reason about what the method is doing.

Strange Return Values

Many of the native functions return multiple types. The `strstr` function returns a string if matched and false if not.

In contrast to this, the `preg_match` function returns `1` when matching a pattern, `0` when not matching, and `false` if an error occurred. This makes for a slew of type checking before using any return values for their intended purpose.

Conclusion

PHP is a great language.

But if you've worked much with PHP, you will either have grown to ignore the sad state of PHP's scalar type handling, or been frustrated by the lack of good alternatives.

And for most small-to-medium sized projects, adding a revised type system is unnecessary. Yet if you learned how to make (or even just use) a well-built type system, wouldn't it make sense to use it in large projects? Or in any projects you cared enough about?

This area of PHP often chases developers into prettier languages. Don't be one of those developers! Learn how to write cleaner code by using a clean abstraction.

CHAPTER 2

Structure

The problems with the core function libraries are mostly about structure. In fact, we could address them by adding encapsulation (classes specifically designed for each scalar variable type), a consistent method naming scheme, and consistent parameter ordering.

In this chapter, we're going to look at how to package a new API. We'll create classes to decorate scalar types, discuss how we can determine the type of scalar variables we're dealing with, and even learn how to deal with the problems introduced by null.

Decorating

Decorating is a term given to the practice of wrapping data in a class, so that we can add properties and methods to the underlying data. Let's look at an example:

```
class StringBox
{
  /**
   * @var string
   */
  protected $data;

  /**
   * @param string $data
   */
  public function __construct($data)
  {
    $this->data = $data;
  }

  /**
   * @return string
   */
  public function toString()
  {
    return (string) $this->data;
  }
```


C. Pitt, *Typed PHP*, DOI 10.1007/978-1-4842-2114-3_2

```
  /**
   * @return string
   */
  public function toUpperCase()
  {
    return strtoupper($this->data);
  }

  /**
   * @return string
   */
  public function toLowerCase()
  {
    return strtolower($this->data);
  }

  /**
   * @param string $needle
   * @param mixed  $offset
   *
   * @return int
   */
  public function getIndexOf($needle, $offset = null)
  {
    $index = strpos($this->data, $needle, $offset);

    if ($index === false) {
      return -1;
    }

    return $index;
  }
}

$box = new StringBox("Hello World");

$box->toString();          // "hello world"
$box->toUpperCase();       // "HELLO WORLD"
$box->toLowerCase();       // "hello world"
$box->getIndexOf("foo");   // -1
$box->getIndexOf("World"); // 6
```

PHP supports this approach, without any extra extensions or dependencies. It's a simple concept, if you think about it. We're using protected properties and exposing public methods for changing and accessing the data.

The difficulty with it is that it leads to a lot more code than just using the native functions. You need to define wrappers. You need to define setters and getters. You have to use them every time you want to put *native* types in and get *native* types out. What do I mean by that last statement?

```
$helloBox = new StringBox("Hello");
$worldBox = new StringBox("World");

$helloWorldBox = new StringBox(
  $helloBox->toString() . " " . $worldBox->toString()
);
```

This quickly becomes unwieldy. It's also more memory intensive and slower than just using the native functions and types. Decoration can be helpful, but it needs to be built on top of a solid foundation and well supported by extensions we'll learn about.

With extensions, we can wrap and unwrap scalar variables transparently, which means we can use objects to extend the functionality of scalar variables without always having to call `new StringBox()` and `$box->toString()` whenever we want to use functions that expect normal strings.

PHP allows the use of a method called `__toString`. When a class has this method, using it in a string operation will invoke this method.

This isn't enough for us to simulate an extensible type system, or this would be a short book.

Resolving Types

PHP is a dynamically typed language. We can declare variables without specifying a type and change their type at any point. We can cast them into different types on demand.

If we want stronger type handling, we need to be able to identify the type of a variable. PHP provides many functions that help with this, but they have a few issues to overcome.

```
function isNumber($variable)
{
    return is_integer($variable) or is_float($variable);
}

function isBoolean($variable)
{
    return is_bool($variable);
}

function isNull($variable)
{
    return is_null($variable);
}
```

```
function isResource($variable)
{
    return is_resource($variable);
}

function isArray($variable)
{
    return is_array($variable);
}
```

Most of these change the underscore type functions to a camel case style. A notable exception is the `isNumber` function. PHP's `is_numeric` function will return true even if the value is a string. What about when the variable contains a callback? In that case, we can use functions.

Functions

The `is_callable` function checks to see if something is a callable function. It can be a string or an anonymous function.

```
function debug($result) {
  print $result ? "true" : "false" . "\n";
}

debug(is_callable("is_callable")); // "true"
debug(is_callable(function(){}));  // "true"
debug(is_callable(null));          // "false"

class Foo
{
  public function bar()
  {

  }

  public function identify()
  {
    return is_callable([$this, "bar"]);
  }
}

$foo = new Foo();

debug($foo->identify()); // "true"
```

`is_callable` identifies a valid argument to any callback-accepting function in PHP. The following are all valid for these kinds of functions:

- An actual function, like `function(){}`
- An array of context and method name, like `[$this, "bar"]`
- The name of a function as a string, like `"is_callable"`

Due to how permissive this method is, we need to be careful when trying to identify strings and functions in the same resolver function. If we use `is_string` and `is_callable` at the same time, our results may vary depending on the order in which we call them.

For example:

```
$variable = "is_callable";

if (is_string($variable)) {
  die("variable is a string");
}

if (is_callable($variable)) {
  die("variable is callable");
}
```

The script terminates with the string `"variable is a string"`, because it is a string. It is also the name of a callable function. Swapping the conditional statements will cause the script to terminate with `"variable is callable"`. We can restrict this a bit with:

```
function isObject($variable)
{
    return is_object($variable) and !isFunction($variable);
}

function isFunction($variable)
{
    return is_callable($variable) and is_object($variable);
}

debug(isFunction(function(){}));  // "true"
debug(isFunction("is_function")); // "false"
debug(isFunction(new stdClass));  // "false"
debug(isObject(new stdClass));    // "true"
```

Regular Expressions

PHP has many string functions (some for normal strings and some for regular expressions). It would be great if we had a way to differentiate between things that look like regular expressions and things that don't.

It's important that we can tell regular expressions apart from strings that look similar. Just because something looks like a regular expression doesn't mean that it is. Nor does it mean that the intention of the author was for it to be a regular expression.

The PHP documentation (at http://www.php.net/manual/en/intro.pcre.php) describes expressions as (a string) enclosed in delimiters. These delimiters can be any non-alphanumeric that isn't also a backslash or null byte.

We can cover a large majority of cases, using the following:

```
function isString($variable)
{
    return is_string($variable) and !isExpression($variable);
}

function isExpression($variable)
{
    $isNotFalse = @preg_match($variable, "") !== false;
    $hasNoError = preg_last_error() === PREG_NO_ERROR;

    return $isNotFalse and $hasNoError;
}

debug(isExpression("/^.*$/"));          // "true"
debug(isExpression("/hello world/"));   // "true"
debug(isExpression("/hello world/i"));  // "true"
debug(isExpression("hello world"));     // "false"
debug(isExpression("\\hello world\\")); // "false"
debug(isExpression("\\x00foo\\x00"));   // "false"
debug(isExpression("1foo1"));           // "false"
debug(isExpression("afooa"));           // "false"
```

The `preg_match` function returns a `1` for a match, `0` for no match, and `false` if an error occurred. Assuming `preg_match` returns false, the error could be for any number of reasons. So then we use the `preg_last_error` function to rule out all other errors.

Using error suppression (`@`) is usually a bad idea. Not so here. A reasonable reason for it to raise a warning is because the string isn't a valid expression.

All Together!

Sometimes we just don't know what type a variable should be, and calling all of these type methods would be inefficient. In that case, we can use a helper method:

```
function getVariableType($variable)
{
    $functions = [
        "isNumber" => "number",
        "isBoolean" => "boolean",
        "isNull" => "null",
        "isObject" => "object",
        "isFunction" => "function",
        "isExpression" => "expression",
        "isString" => "string",
        "isResource" => "resource",
        "isArray" => "array"
    ];

    $result = "unknown";

    foreach ($functions as $function => $type) {
        $function = $function;

        if ($function($variable)) {
            $result = $type;
            break;
        }
    }

    return $result;
}
```

So, if we want to know if a variable is a string, we can use the `isString` function. If we don't know what type is should be, then we can use the `getType` function. It'll return `unknown` if the type can't be determined, although the chance of that is slim.

Namespace Functions

Namespaces are a simple alternative to global namespace pollution. You've probably used namespaces for classes, but they work just as well to isolate functions:

```
namespace Type\String {
  function length($string) {
    return strlen($string);
  }
}
```

```
namespace {
  print Type\String\length("Hello World"); // 11
}
```

Composer Autoload

This kind of function definition doesn't follow normal autoload patterns. To have Composer autoload these kinds of files, we need to define them. If we can autoload them, we won't have to constantly `require` them in every script that seeks to use them. To load them, we just need to add a file path to `composer.json`:

```
{
  "autoload" : {
    "files" : [
      "namespace-functions.php"
    ]
  }
}
```

Following this, we'll have to dump the old autoloader, with:

```
$ composer dump-autoload

Generating autoload files
```

Composer will now automatically load these namespace functions.

Importing Namespaced Functions

PHP 5.6 introduced support for importing functions into another namespace. Before that, we had to refer to the namespace of the function whenever we called it:

```
use Type\String;

print String\length("Hello World");
```

Now we can import the full function path to save ourselves that extra bit of typing:

```
use function Type\String\length;

print length("hello world");
```

Optional Types

Dynamic languages suffer particularly due the problem of null references. Null references are errors that happen when a method is called on a `null` value. This usually happens because some code expects an object (on which to run the method) but instead gets `null`.

The Null Problem

To illustrate this problem, let's imagine what happens in the following pseudo-code:

```
while(true) {
  if ($deferredProcess->complete()) {
    print $deferredProcess->result;
    break;
  }

  sleep(1);
}
```

This loop will run forever. That is until `$deferredProcess->complete()` returns `true`. At that point, the result will be printed and the loop will end. We add `sleep(1)` so the machine it's running on doesn't melt!

What if `$deferredProcess` isn't the object we expect? What if it's another object, which doesn't have a `complete()` method? What if it's `null`? In these circumstances, the script will end with a fatal error.

So, what often tends to happen (with some defensive programming) is that we add many checks, as the next bit of pseudo-code does. If the object is an object, and if the object has a `completed()` method, and if…

```
while(true) {
  if (!is_object($deferredProcess)) {
    break;
  }

  if (!method_exists($deferredProcess, "complete")) {
    break;
  }

  if ($deferredProcess->complete()) {
    print $deferredProcess->result;
    break;
  }

  sleep(1);
}
```

This bloats up the code we write, but it's one of the few ways a dynamic language can be safely used. Let's consider another pseudo-code example:

```
$user = $database
  ->table("user")
  ->where("id", "=", $id)
  ->first();

if (!$user) {
  print "Error: User not found";
}

$address = $user->address;

if (!$address) {
  print "Error: Address not found";
}

print "City: " . $address->city;
```

The more we chain potentially null variables, the more we need to check (or hope) that the variables aren't `null`.

Optional Values

One way around this problem it to decorate variables and intercept method calls on `null` before they generate errors.

■ **Note** There are many implementations and variations of the `Optional` class presented here in pseudo-code. The important thing to know is what these classes enable, not how to implement any specific variation. There are links to downloadable code toward the end of this section and we'll make our own implementation in later chapters.

We could do something like this:

```
$optional = new Optional($user);

print $optional->address()->city()->value();
```

What looks almost magical is really just a case of implicit null-checking. An `Optional` class could implement `__get()` and `__call()` methods so that they are completely avoided on null values. Sure, you'd still get `null` as the result of that, assuming `$user` is `null`. You could avoid the fatal errors though.

We could even introduce a kind of error handling such that specific error handing could be possible:

```
$optional = new Optional($user);

$optional
  ->address()
  ->none(function() {
    print "Error: Address not found";
  })
  ->city()
  ->none(function() {
    print "Error: City not found";
  })
  ->value(function($value) {
    print "City: " . $value;
  });
```

We call this a *fluent interface* (http://en.wikipedia.org/wiki/Fluent_interface#PHP). We can use it to protect methods and properties from `null` errors. There are two trade-offs though.

The first is that the methods and properties need to implement the *null object pattern* (http://en.wikipedia.org/wiki/Null_Object_pattern). If they do not, we're stuck with the same null-reference errors as before.

The second trade-off is that these values cannot be automatically unwrapped. Every interaction with these objects will end in a call to a `value()` method.

I have chosen an interface that loosely resembles *Promises* (http://en.wikipedia.org/wiki/Promise_(programming)). Promises represent a future value, and are often used for concurrent or asynchronous programming. These objects do not involve either of those paradigms. It's just a neat model for managing type uncertainty.

In The Wild

I am not the first person to think/dabble in this way. The term *Promise* (which inspired this interface) was proposed in 1976. Nullable types were invented in 1965.

More recently, Johannes Schmitt devised a library (https://github.com/schmittjoh/php-option) that implements these ideas. His implementation has different interface than the one I have described. Igor Wiedler wrote an article (see https://igor.io/2014/01/10/functional-library-null.html) about his library.

Simple as it may look, implementing it will be tricky. We will see it again in the same implementation. You should also check out Johannes Schmitt's implementation.

Conclusion

Even today, it's possible to create a cleaner scalar type system/abstraction. We don't need special extensions or compilation steps. Just a little bit of work will clean the code right up.

Extensions can make our lives easier, while still allowing us to create consistent interfaces. We'll look at a few of these in the following chapter.

CHAPTER 3

Extensions

So far we've looked at a few problems with the current core functions, and potential solutions to them, using ordinary PHP code. These could go a long way to improving our quality of life, but we're not done yet.

Let's look at what we can achieve when we combine our ordinary PHP code with extraordinary C extensions. We'll learn how to set up the perfect environment for these extensions to be installed into, and how to tap in to the enhanced functionality they provide.

Vagrant + Phansible

Many of the libraries we will be working with need a bit of special installation. Instead of discussing each operating system, we'll look at how to use Vagrant. It will provide a consistent environment for all the libraries and installation instructions.

Vagrant is a programmatic interface for managing virtual machines. If you've ever set up a virtual machine, you will know how much time it takes to do well. Vagrant can automate this process.

Vagrant depends on underlying virtualization providers and provisioners. We'll look at using VirtualBox as the virtualization provider and Ansible as the provisioner.

Installing

To get VirtualBox installed, go to `https://www.virtualbox.org/wiki/Downloads` and download the installer for your operating system (see Figure 3-1).

C. Pitt, *Typed PHP*, DOI 10.1007/978-1-4842-2114-3_3

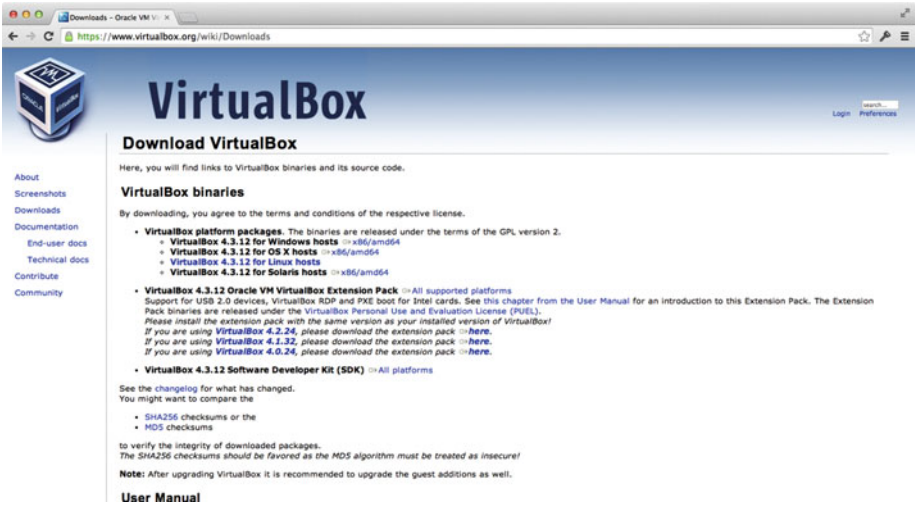

Figure 3-1. *Download VirtualBox for your operating system*

Once you have downloaded and installed VirtualBox (see Figure 3-2), you should be able to install Vagrant.

Figure 3-2. *Follow all the installation steps*

Next, go to `http://www.vagrantup.com/downloads.html` and download the installer for your operating system (see Figure 3-3).

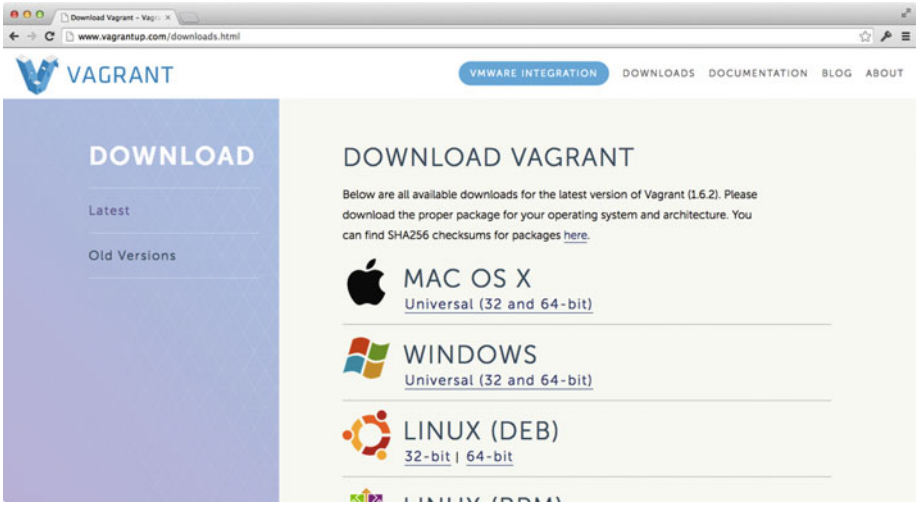

Figure 3-3. *Download Vagrant for your operating system*

Now follow the installation instructions (see Figure 3-4).

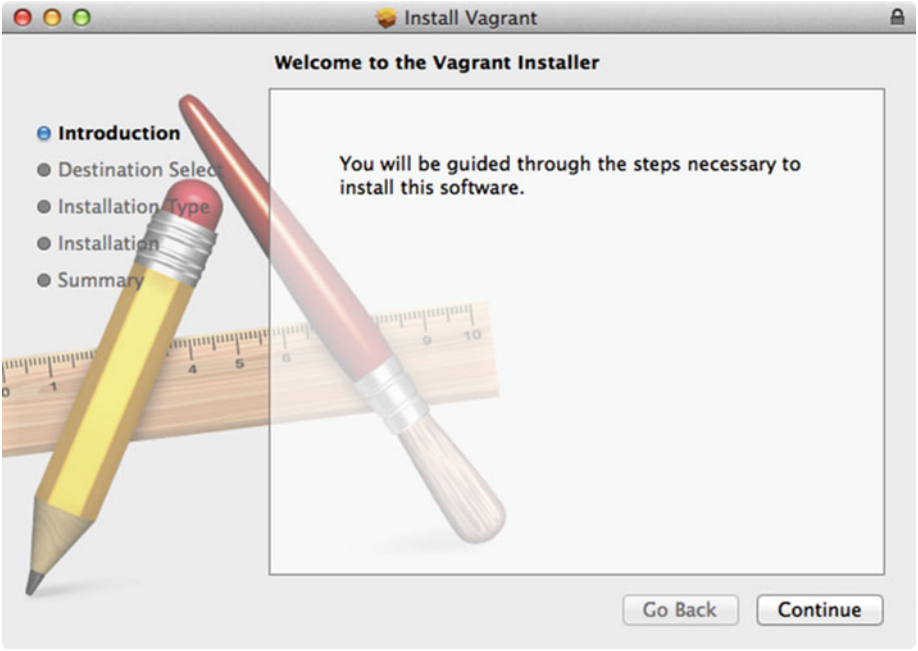

Figure 3-4. *Follow all the installation steps*

We'll also need to install Ansible so we can use play books to provision the virtual machine. Go to `http://docs.ansible.com/intro_installation.html` and download the installer for your operating system (see Figure 3-5).

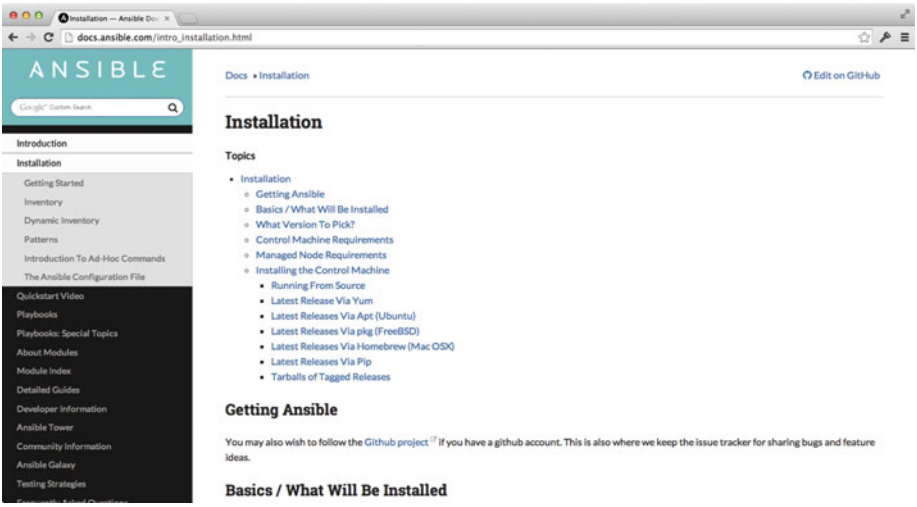

Figure 3-5. *Download Ansible for your operating system*

Provisioning

Provisioning scripts tell Vagrant which dependencies to install. There are different kinds of Vagrant provisioners, but Ansible is the one we will use.

We'll use `http://phansible.com` to do most of the heavy lifting (see Figure 3-6).

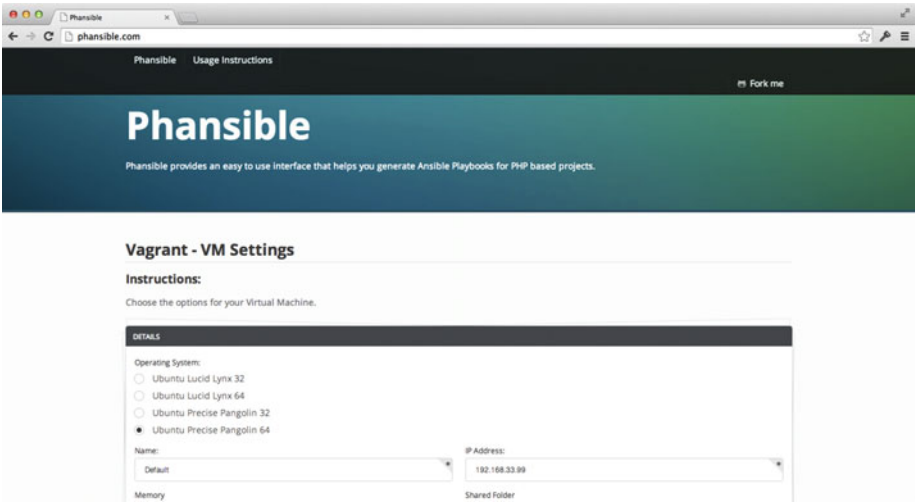

Figure 3-6. *Build a set of provisioning scripts on phansible.com*

Set the following options:

- Operating system: Ubuntu Trusty Tahr 64
- Webserver: Nginx + PHP5-FPM
- PHP: 5.6 (the most recent version tested and supported by Phansible and the extensions)
- Composer: Enabled
- PHP modules: php-pear php5-cli php5-common

When you click Generate, you'll start downloading an archive of files. Extract these into a working directory and start up Terminal.

These files are instruction files that describe which dependencies Vagrant must install. You shouldn't need to change them to get the PHP stack working, but feel free to familiarize yourself with what they are doing.

At the time of writing, Phansible doesn't yet support provisioning PHP 7 servers. That's okay since these extensions have not yet thoroughly been tested to work with PHP 7, but the concepts explained should work in PHP 5.x and PHP 7.x once they are.

To start the virtual machine, run the following command:

```
$ vagrant up

Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'trusty64'...
==> default: Matching MAC address for NAT networking...
==> default: Setting the name of the VM: Default
==> default: Clearing any previously set network interfaces...
```

Vagrant might ask you to provide an administrator password as part of setting up the virtual machine. This will allow the NFS shared folders to be set up.

Once the virtual machine is set up, you can log in:

```
$ vagrant ssh

Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-77-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

 System information disabled due to load higher than 1.0
```

```
  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.
vagrant@default:~$
```

You can also check the installed version of PHP with:

```
$ php -v

PHP 5.6.22-1+donate.sury.org~trusty+1 (cli)
Copyright (c) 1997-2016 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2016 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2016, by Zend
Technologie
```

Vagrant Commands

There are a few Vagrant commands you're likely to use often:

```
$ vagrant up
```

This command will start the Vagrant virtual machine and do any outstanding provisioning. That means the first time you run this command, it might take longer boot.

```
$ vagrant halt
```

This command will shut the virtual machine down gracefully.

```
$ vagrant destroy
```

This command will remove the virtual machine and clean up settings applied during setup. If you break something inside the virtual machine, you might want to reset it to the default state. You can do this by running `vagrant up`.

```
$ vagrant ssh
```

This command will take you inside the virtual machine, just as if you were connecting to a remote server. Inside the virtual machine, you can run any of the commands usually supported by the guest operating system. This includes executing PHP scripts against the packages installed on the virtual machine.

`http://phansible.com` is the brain-child of Erika Heidi. She is also the author of *Vagrant Cookbook* (`https://leanpub.com/vagrantcookbook`). I recommend reading this book if you have any questions or want to know more about Vagrant!

SPL Types

SPL (or Standard PHP Library) is a library of extra types to augment those native to core PHP. There are some popular classes (like `LogicException` and `ArrayObject`). Some of the SPL ships with *standard* PHP installations. The parts we're going to look at shortly do not.

You can find these mysterious libraries at `http://www.php.net/manual/en/book.spl-types.php`.

This section assumes you're using the Vagrant box explained earlier. If not, please set that up first. These commands are Linux-specific and depend on the pre-installed modules explained earlier.

Installing SPL Types

To install the libraries, run the following commands:

```
$ sudo apt-get install libpcre3-dev php5-dev

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  autoconf automake autotools-dev build-essential...

$ sudo pecl install SPL_Types

downloading SPL_Types-0.4.0.tgz ...
Starting to download SPL_Types-0.4.0.tgz (8,388 bytes)
.....done: 8,388 bytes
6 source files, building
running: phpize...
```

These two commands install the prerequisites for compiling PECL extensions. PECL is an extension repository just like PEAR and Packagist.

```
$ sudo bash -c "echo extension=spl_types.so >> /etc/php5/cli/php.ini"
```

This command appends `extension=spl_types.so` to the `php.ini` file (as per the installation instructions).

```
$ sudo service php5-fpm restart

php5-fpm stop/waiting
php5-fpm start/running, process...
```

This command restarts PHP-FPM. It's the process that interprets PHP command line instructions and Nginx web requests. These commands should have installed the SPL types, but just to be sure, run the following command:

```
$ php -i | grep SPL_Types
SPL_Types
```

If you see that `SPL_Types` line, you should be good to go!

Using SPL Types

Let's look at a few examples of how these classes can be used:

```
class NumberType extends SplFloat
{
  /**
   * @return float
   */
  public function toInteger()
  {
    return round($this);
  }

  /**
   * @return string
   */
  public function toString()
  {
    return (string) $this;
  }
}

$number = new NumberType(13.86);

print $number->toInteger(); // 14
print $number->toString();  // "13.86"

print (float) $number + 1.00; // 14.86
print $number * 12;           // 156
```

The `toInteger` and `toString` methods do similar things to the box classes. The magic happens when we do basic arithmetic with the `$number` object. SPL types are automatically unboxed when used in arithmetic expressions, cast, or concatenated. Any operator that would normally work with a scalar type will work with the corresponding SPL type.

Here's another example:

```
class StringType extends SplString
{
  /**
   * @param int   $start
   * @param mixed $length
   *
   * @return StringType
   */
  public function slice($start = 0, $length = null)
  {
    if ($length === null) {
      return new static(substr($this, $start));
    }

    return new static(substr($this, $start, $length));
  }
}

$string = new StringType("Hello World");

print $string->slice(6); // "World"
```

We can design our types so that they return new instances. This gives us a simple *chaining* interface.

Be careful when assuming the return type of native PHP functions. Be sure to check them before the call to `new static()` or you may encounter fatal errors.

Scalar Objects

Nikita Popov is a prolific contributor to PHP (both core and user-land). He's made libraries such as PHP-Parser (https://github.com/nikic/PHP-Parser), which many popular frameworks also use. He's championed many RFCs that have become parts of core PHP.

He's also created a custom extension that allows the registration of custom type handlers. You can find it at https://github.com/nikic/scalar_objects.

We're going to install and use this module to get even closer to our ideal type handling situation.

This section assumes you're using the Vagrant box explained earlier. If not, please set that up first. These commands are Linux-specific and depend on the pre-installed modules explained earlier.

Installing Scalar Objects

First up, we need to install the Git command-line tool:

```
$ sudo apt-get install git

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  git-man liberror-perl...
```

Following this, we can clone and build the extension:

```
$ git clone https://github.com/nikic/scalar_objects.git

Cloning into 'scalar_objects'...
remote: Reusing existing pack: 213, done.
remote: Total 213 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (213/213), 75.36 KiB, done.
Resolving deltas: 100% (112/112), done.
$ cd scalar_objects && phpize && ./configure && make && sudo make install
```

These commands will the Scalar Objects extension, but we still need to add it to the configuration:

```
$ sudo bash -c "echo extension=scalar_objects.so >> /etc/php5/cli/php.ini"
```

This command resembles the one we used to install the SPL types. We're doing the same thing, so we need to restart PHP5-FPM:

```
$ sudo service php5-fpm restart

php5-fpm stop/waiting
php5-fpm start/running, process...
```

This should complete the process of installing the Scalar Objects extension. We can make sure it's working by running the following command:

```
$ php -i | grep scalar

scalar_objects
scalar-objects support => enabled
```

Using Scalar Objects

The new extension adds a method we can use to register these type handlers. This is how we can use it:

```
class StringHandler
{
  /**
   * @param int   $start
   * @param mixed $length
   *
   * @return StringType
   */
  public function slice($start = 0, $length = null)
  {
    if ($length === null) {
      return substr($this, $start);
    }

    return substr($this, $start, $length);
  }
}

register_primitive_type_handler("string", "StringHandler");

$string = "Hello World";

print $string->slice(6); // "World"
```

This is easier than boxing scalar types, as we don't have to pull native types out of class instances. This is easier than SPL types, as we don't have to put native types into class instances.

There are seven supported types:

- `null`
- `bool`
- `int`
- `float`
- `string`
- `array`
- `resource`

You may be wondering whether this extension plays nicely with SPL types. The answer is *probably not*. You shouldn't mix these extensions. Since the Scalar Objects extension does everything that SPL types do, you won't need both.

Zephir

Zephir is a framework for writing extensions. It uses a superset of PHP language, sharing some similarities with C code. Zephir isn't a PHP extension, nor are Zephir libraries written in true PHP.

It's part of the same collective from which the Phalcon framework comes, and Phalcon is itself a PHP extension. Zephir allows us to use a language similar to PHP but which can be compiled down to a C PHP extension. This allows for very fast code without the need to understand or wrestle with the internals of a PHP interpreter.

Installing Zephir

Zephir requires a few libraries to compile. We can install these with:

```
$ sudo apt-get install git gcc make re2c php5 php5-json php5-dev libpcre3-dev
```

Next, we need to install the JSON-C library (which Zephir uses to compile extensions):

```
$ git clone https://github.com/json-c/json-c.git && cd json-c && sh
autogen.sh && ./configure && make && sudo make install
```

These commands will clone the JSON-C repository, and then configure and compile it. Finally, we need to install Zephir:

```
$ git clone https://github.com/phalcon/zephir && cd zephir && ./install -c
```

That should have installed a usable version of Zephir. You can check that it's working by heading into the clone folder and running:

```
$ zephir version
```

Using Zephir

Using Zephir is easy (considering the work that it does). Let's begin by initializing a new extension skeleton project:

```
$ zephir init type
```

This will create a skeleton project folder in the current working directory. Navigate into the new `type` directory and run the following:

```
$ ls -la

ext/ type/ config.json
```

Extension classes go in the `type` folder (it's specific to the name of the extension, which we gave the `init` command). Make a file in there, called `StringType.zep`, and open that file in your editor.

The Zephir syntax is quite like PHP, but with a twist of C style. You can find a reasonable amount of documentation at http://www.zephir-lang.com/index.html.

Create the following class:

```
namespace Type;

class StringType
{
  protected data;

  public function __construct(var data)
  {
    let this->data = data;
  }

  public function length()
  {
    return strlen(this->data);
  }
}
```

Other than the missing $ symbols and the `var`/`let` keywords, this is pretty understandable. Save the file (from the base extension folder) and then run:

```
$ zephir build

Compiling...
/bin/bash /vagrant/zephir/type/ext/libtool --mode=compile gcc
-I. -I/vagrant/zephir/type/ext -DPHP_ATOM_INC-
I/vagrant/zephir/type/ext/include -
I/vagrant/zephir/type/ext/main -I/vagrant/zephir/type/ext -
I/usr/include/php5 -I/usr/include/php5/main -
I/usr/include/php5/TSRM -I/usr/include/php5/Zend -
I/usr/include/php5/ext -I/usr/include/php5/ext/date/lib -
```

```
DHAVE_CONFIG_H  -O2 -fvisibility=hidden -Wparentheses -flto -
c /vagrant/zephir/type/ext/type/stringtype.zep.c -o
type/stringtype.lo...
```

Zephir cross-compiles the extension class files to "Plain Ol' C," and adds the class loading code. The end of the build output should look something like this:

```
Installing...
Extension installed!
Add extension=type.so to your php.ini
Don't forget to restart your web server
```

We need to add the extension to the `php.ini` file:

```
$ sudo bash -c "echo extension=type.so >> /etc/php5/cli/php.ini"
```

This will install the type extension we just created. We can check that it's installed by running:

```
$ php -i | grep "type => enabled"

type => enabled
```

If you see that line returned, you know the extension is installed and ready to go.

Using this new extension is as simple as running:

```
$string = new Type\StringType("Hello World");

print $string->length();
```

The namespace and class exist completely within the compiled extension file. Zephir extensions can use preexisting core and extension namespaces/classes. They can be used by plain PHP code (provided the extension is registered by the time it's used).

Zephir extensions can even override core functions, with better-performing versions.

Conclusion

Extensions make our lives easier by handling things like boxing and unboxing for us. They let us create better-performing code (as in the case of Zephir) and stricter types (as in the case of SPL types).

We don't have to use these to make a cleaner system. If we do, we can expect to have a much strong type system, without the hard work that library-only code expects of us.

CHAPTER 4

Design

Now that we're familiar with the tools at our disposal, it's time to put them together into a library we can proudly reuse.

In this chapter we'll organize the "core" functions we want to keep and get rid of the noise. We'll also learn how to package the functions we want with the C extensions we saw in the previous chapter.

Which Method to Use

We've had a look at some methods and extensions that can help us to abstract away the inconsistent type handling PHP presents to us. Part of creating this abstraction is deciding on which methods and/or extensions to use.

Namespace Methods

We should avoid any method that would expose a significant amount of functions in the global scope. We don't want to create any more clutter than there already is. Being able to use our abstraction alongside the standard stuff is definitely beneficial.

We should endeavor to have all our code exist in namespaces.

We should implement our code so that we can use it in procedural environments and object-oriented environments. We should contain the business logic within functions and call those functions from within an object oriented framework.

We should build our logic in functions and add those functions (as methods) to scalar type objects.

C. Pitt, *Typed PHP*, DOI 10.1007/978-1-4842-2114-3_4

Scalar Objects/SPL Types

This means we will want to use the namespace functions we covered earlier, together with scalar objects or the SPL types.

I mentioned that we shouldn't use scalar objects and SPL types together, because they do the same thing. It's not possible to delegate to the functional code without repetition for each extension.

This is because scalar objects boxes scalar types, while SPL types expect us to do the boxing. It's the difference between `return new static(substr($this, $offset, $length));` and `return substr($this, $offset, $length);`.

While I love scalar objects, i am inclined to suggest we go with SPL types. The main reason is that it doesn't interfere with how PHP handles scalar types. This means we will need to box scalar types ourselves. We'll still get to enjoy the benefits of object types and automatic unboxing.

Zephir

As cool as Zephir is, it's not PHP. That means any developers you want to work on your code will need to know or learn another language. It also increases the time between coding, testing, and shipping.

So, for the rest of the book, I will show code that is built on top of SPL types and not translated into Zephir extensions.

Which Functions to Keep

We'll start by deciding which core PHP functions should be kept in our abstraction. We're only interested in the functions that apply to scalar types, and we're going to coalesce some of them into the following list:

- String
- Number (integer + float)
- Boolean

We'll also reimplement the array type using a number of interfaces (like `Countable` and `IteratorAggregate`).

String Functions

Of the many listed string methods, the following is a list of methods I think we should keep:

- addslashes
- chop
- chr
- chunk_split
- explode
- ltrim
- money_format
- number_format
- ord
- rtrim
- sprintf
- str_ireplace
- str_pad
- str_repeat
- str_replace
- str_split
- strchr
- stripos
- stripslashes
- stristr
- strlen
- strpos
- strrev
- strstr
- strtok
- strtolower
- strtoupper
- substr
- trim

- `ucfirst`
- `ucwords`
- `vsprintf`
- `wordwrap`

These can be grouped a number of ways, as described next.

addslashes and stripslashes

These functions add and remove slashes for the purposes of quoting special characters. These are double quotes, single quotes, and backslashes (and null bytes). A good reason for this is if you want to run a string through `eval` and the contained quotes render the syntax otherwise invalid.

chop, trim, ltrim, and rtrim

These functions strip characters from the beginning and end of a string. By default, these characters are whitespace characters, but other characters can be removed also.

chr and ord

These functions convert to and from the numeric values of ASCII characters. Think of the numeric values used every time you press a keyboard key.

money_format, number_format, sprintf, and vsprintf

These functions change the format of a string, according to a series of special characters. Some of them deal with numeric representations, while the rest are just arbitrary formatters.

strchr, stripos, stristr, strpos, and strstr

These functions identify the presence of a substring, or its position in a larger string.

chunk_split, explode, str_split, and strtok

These functions not only find a smaller string within a larger one, but they use the position of these smaller strings to break the larger strings up on each occurrence.

str_ireplace, str_replace, and substr

These functions modify a string, either by replacing parts of them or slicing them up by offset and character count.

str_pad and str_repeat

The functions increase the length of a string, either by repeating the characters of which it is composed, or additional characters, always towards a fixed length.

strtolower, strtoupper, ucfirst, and ucwords

These functions modify the case of some (or all) of the characters in the string.

strlen, strrev, and wordwrap

These are the remaining functions: `strlen` returns the length of a string, `strrev` reverses a string, and `wordwrap` truncates a string.

Number Functions

Of the many listed number methods, the following is a list of methods I think we should keep:

- `abs`
- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atan2`
- `atanh`
- `bindec`
- `ceil`
- `cos`
- `cosh`
- `decbin`
- `dechex`
- `decoct`
- `deg2rad`
- `exp`
- `expm1`

- floor
- fmod
- getrandmax
- hexdec
- log
- log1p
- max
- min
- mt_getrandmax
- mt_rand
- mt_srand
- octdec
- pi
- pow
- rad2deg
- rand
- round
- sin
- sinh
- sqrt
- srand
- tan
- tanh

These can be grouped a number of ways, discussed next.

abs, ceil, floor, and round

These functions remove the decimal values from floating point numbers, either by rounding or truncating them. Since we're trying to coalesce floats and integers into a single set of functions, we'd better allow for this kind of conversion.

acos, acosh, asin, asinh, atan2, atan, atanh, cos, cosh, sin, sinh, tan, and tanh

These functions are used in geometric calculations to determine the size of angles. They return radian values, which means we also need to implement the following.

deg2rad and rad2deg

These functions convert values between radians and degrees. These are especially useful for geometric calculations.

bindec, decbin, hexdec, dechex, octdec, and decoct

These functions perform similar conversions (based on base number system) between the popular base numbers systems decimal, binary, hexadecimal, and octal.

mt_getrandmax, mt_rand, mt_srand, getrandmax, rand, and srand

These functions deal with the generation of (pseudo) random numbers. We'll want to reduce the number of methods, but that will only be possible by constructing a facade around at least three of them.

exp, expm1, log, and log1p

These functions deal with exponents and logarithms, not that the signatures make this obvious.

pow, sqrt, fmod, and pi

These are the remaining functions: `pow` raises a value to a power, `sqrt` calculates the square root of a value, `fmod` calculates modulo (remainder of division) values, and `pi` returns an approximation of the value of pi.

Array Functions

Of the many listed array methods, the following is a list of methods I think we should keep:

- `array_chunk`
- `array_column`
- `array_combine`
- `array_count_values`

- array_diff
- array_diff_assoc
- array_diff_uassoc
- array_diff_ukey
- array_fill
- array_fill_keys
- array_filter
- array_flip
- array_intersect
- array_intersect_assoc
- array_intersect_key
- array_intersect_uassoc
- array_intersect_ukey
- array_key_exists
- array_keys
- array_map
- array_merge
- array_merge_recursive
- array_multisort
- array_pad
- array_pop
- array_product
- array_push
- array_rand
- array_reduce
- array_replace
- array_replace_recursive
- array_reverse
- array_search
- array_shift
- array_slice

- array_splice
- array_sum
- array_udiff
- array_udiff_assoc
- array_udiff_uassoc
- array_uintersect
- array_uintersect_assoc
- array_uintersect_uassoc
- array_unique
- array_unshift
- array_values
- array_walk
- array_walk_recursive
- arsort
- asort
- count
- in_array
- key_exists
- krsort
- ksort
- natcasesort
- natsort
- rsort
- shuffle
- sizeof
- sort
- uasort
- uksort
- usort

These can be grouped a number of ways, discussed next.

array_chunk, array_column, array_slice, array_keys, and array_values

These functions reduce arrays into smaller arrays, either by returning a subset or by offset and length.

array_combine, array_merge, array_merge_recursive, array_fill_keys, array_fill, and array_pad

These functions create new arrays by swapping keys/values, combining multiple arrays or adding fillers to arrays.

array_diff_assoc, array_diff_uassoc, array_diff_ukey, array_diff, array_udiff_assoc, array_udiff_uassoc, and array_udiff

These functions calculate the differences between multiple arrays. There are so many of them because there are many ways in which arrays can differ. Ideally, these should be condensed into fewer functions, while still offering the same flexibility.

array_intersect_assoc, array_intersect_key, array_intersect_uassoc, array_intersect_ukey, array_intersect, array_uintersect_assoc, array_uintersect_uassoc, and array_uintersect

These functions are similar to the `diff` functions, but instead of calculating the difference, they calculate which values the arrays have in common (the intersection points).

array_key_exists, array_search, in_array, and key_exists

These functions check if keys or values are present in an array. Think of them as search functions for a collection of items.

sizeof, count, and array_count_values

These functions all return a count of the values.

array_filter, array_map, array_reduce, array_replace, array_replace_recursive, array_walk_recursive, array_walk, and array_splice

These functions iterate over an array and do something for each item. Some whittle down an array (like `array_filter` and `array_reduce`), while others alter an array (like `array_map` and `array_replace`).

array_multisort, arsort, asort, krsort, ksort, natcasesort, natsort, rsort, uasort, uksort, usort, sort, shuffle, array_reverse, and array_flip

These functions change the order of an array. They mostly sort, except for the last three. `array_flip` swaps keys and values. `shuffle` and `array_reverse` are self-explanatory.

array_pop, array_push, array_shift, and array_unshift

These functions add or remove individual items from either end of the array. PHP arrays are ordered, so you can depend on the position of items you add via these methods.

array_product, array_rand, array_sum, and array_unique

These methods perform aggregate functions on all items in an array.

Which Functions to Add

In the years since the first set of core functions were introduced, a few different frameworks and smaller libraries have added their own useful functions to the mix. Here are some I found while browsing the documentation of my favorite frameworks and libraries.

String Functions

We deal with strings so often in programming, yet we often need to repeat common functionality or deal with slight variations in the interfaces of common function libraries. The following are a few functions we could create common interfaces for in our library.

toCamelCase

This returns a string, converted from snake case to camel case.

toSnakeCase

Similar to `toCamelCase`.

endsWith

Returns true if the string ends with the specified substring.

startsWith

Similar to `endsWith`.

complete

Ends a string with only one instance of the specified terminator substring.

Array Functions

In PHP we only really have the magical array when it comes to collections. Let's add a few helpful functions to the list of core functions we're keeping.

separate

This is the opposite of the `array_combine` function, which builds a new array from one of keys and another of values. `separate` breaks an array into one of keys and another of values.

add

This adds a new key/value combination if the key isn't already in the array.

getExcept

This returns a new array, excluding the specified keys.

getOnly

This returns a new array only including the specified keys.

getFirst

This returns the first item that returns true for a provided callback, or simply the first item if no callback is given.

getLast

Similar to `first`, but applies to the last item in an array.

flatten

This flattens a multidimensional array into a single dimensional array.

get

This returns the value of a matching key, or the default value in the event that the key is not matched.

max

This returns the highest number in a numerical array.

min

Similar to `max`.

How to Structure Functions

We've already decided to use namespaced functions as the basis, so we need to decide how to use these from within classes.

Resolving Types

We'll often need to resolve types within the type methods. Any arguments could potentially be the wrong type. It makes sense for the type resolution/conversion functions to be in their own namespace:

- `Type\isString`
- `Type\isStringObject`
- `Type\isExpression`
- `Type\toStringObject`
- `Type\toExpression`
- etc.

We should proxy to the creation methods:

```
class StringObject extends SPLString
{
  public function trim($mask = "\t\n\r\0\x0B")
  {
    $isString       = Type\isString($mask);
    $isStringObject = Type\isStringObject($mask);

    if ($isString or $isStringObject) {
      if (Type\isExpression($mask)) {
        $raw = Type\String\trimWithExpression(
          $this,
          $mask
        );
      } else {
        $raw = Type\String\trimWithString(
          $this,
          $mask
        );
      }

      return Type\toStringObject($raw);
    }

    throw new LogicException("mask is not a string");
  }
}
```

This code expects the `SPL_Types` extension to be installed on the machine running it. Use the same Vagrant box we set up in the previous chapter.

Chaining

One of the benefits of the object oriented approach is that we can chain calls on the types. You may have noticed how this is implemented (from the previous example), but in case you didn't:

```
return Type\toStringObject($raw);
```

The manipulation methods should return plain scalar types. We want people to be able to use the types interchangeably. So it falls to the classes (proxies) to wrap plain PHP types within the SPL types.

Combining Number Types

I've already alluded to my desire of a single number type, and that's exactly what I want to achieve here. For that reason, we'll completely ignore the `SPLNumber` class in favor of `SPLFloat`.

The reason is simple: Numbers should be able to handle decimal points, without a separate set of methods or mental overhead. This will lead to extra casting in numeric operations, but that's not the end of the world.

How to Test

Testing is an essential part of supplanting the native type handling system. The good news is that it will be easy to do!

PHPUnit

PHPUnit is a unit-testing library that makes the process of testing small units of code super easy. To install it, run the following command:

```
$ composer require phpunit/phpunit
```

We can write PHPUnit tests by creating classes resembling the following:

```
class StringObjectTest extends PHPUnit_Framework_TestCase
{
  /**
     * @test
   */
  public function trimWorksWithStrings()
  {
    $subject = Type\toStringObject("Hello World...");

    $this->assertEquals(
      "Hello World",
      $object->trim(".")
    );
  }
}
```

We can do this kind of testing with many different testing frameworks. At the end of the day, as long as you are writing tests, whichever testing framework you use is up to you.

What Should We Test?

The short answer is: everything.

We're aiming to build something solid, and it's pretty low-level. That means we need to be sure things continue to work as expected. It's not even that hard when you consider how simple the methods are that we are going to make.

We should aim to have good coverage of the namespaced functions, and a few tests to ensure that these are correctly called from the object classes.

When Should We Write Tests?

That is up to you. Maybe you want to write your tests first, and then follow the red-green-refactor cycle. Maybe you want to write your library code first, and then make sure everything works as you expect.

The important thing is to write tests.

Recommendations

If you want to learn more about writing cleaner, more testable code, I recommend the following books:

- *Clean Code,* by Robert C. Martin: `http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882`
- *The Grumpy Programmer's PHPUnit Cookbook,* by Chris Hartjes: `https://leanpub.com/grumpy-phpunit`

How to Package

If you want people to use your code, you'd be wise to package it in such a way that others will want to use it.

Make a Readme File

It sounds simple, but it may surprise you how few developers do this! When other developers stumble across your repository, they won't like to see an undocumented mess of code.

Make a `Readme` file and include the following.

A Few Examples

Show what your library can do. Example code helps others understand the problem your library solves. They shouldn't need to go to tests to see that.

Tests and documentation are good places to learn, but they must be sought out. Include examples covering the major aspects of your library, right in the `Readme` file.

In our case, this means an example of the procedural code underpinning the library. It also means an example of the object oriented wrappers.

Testing Instructions

Developers want to know how solid your code is. Sure, you're using SemVer (http://semver.org), but how much test coverage do you have? Do your tests pass? Can I trust you?

The easiest way to answer these questions is to write the unit tests and provide instructions for how to run them. You don't need anything elaborate:

```
$ composer install && phpunit
```

That's how you might be running your unit tests, and it's easy to tell others how to run them.

Installation Instructions

Okay, you've convinced someone to use your well-tested library. The examples prove its value. So how do they install your library?

This is where you make a `composer.json` file and include Composer installation instructions. This will require a trip to Packagist (https://packagist.org), but you'll be all the better for it.

Then just a simple set of instructions is all you need add:

```
$ composer require "vendor/library:1.0.0"
```

License

Include an open source license. Something friendly like MIT (http://opensource.org/licenses/MIT) will do nicely. Specify this in your `Readme` file, and include a clearly-named file (like `LICENSE` or `license.md`) that contains the full license.

Contribution Guidelines

This is optional, but greatly increases the chance that other developers will send compatible pull-requests. If you are fussy about code style, be sure to tell people how you want their submissions to look.

This is usually specified in a file named `CONTRIBUTING` (or something similar).

Conclusion

Building a new type system is only partly about code. There are a lot of design considerations that go into well-crafted libraries. Take the time to decide on a reasonable structure, one that gives consumers the most flexibility.

Test well. Package well. Be clear.

CHAPTER 5

Implementation

Following on from last chapter, we're going to continue with the implementation of our library. In this chapter we're also going to learn a bit about how to extend Composer through plugins.

We'll finish off with the definitions of the most important library functions.

Extending Composer

Composer has become the de facto package manager for PHP. It's a powerful tool with a powerful plugin system. Let's take a look at how to create plugins, so we can automate some of the installation hassles our type library might create.

Example: Path Plugin

I've seen many frameworks create their own installers for things like themes and modules. This is often because they want to use Composer to distribute code, but don't want that code living inside the vendor folder.

Composer's plugin system easily allows for custom install locations.

You can find this section's code at https://github.com/typedphp/composer-path-plugin.

Getting Started

To begin setting up a new plugin, you'll need to create a new working directory and a minimal `composer.json` file:

```
{
  "name" : "typedphp/composer-path-plugin",
  "type" : "composer-plugin",
  "license" : "MIT",
  "authors" : [
```


C. Pitt, *Typed PHP*, DOI 10.1007/978-1-4842-2114-3_5

```
    {
      "name" : "Christopher Pitt",
      "email" : "cgpitt@gmail.com"
    }
  ],
  "require" : {
    "php" : ">=5.5.0",
    "composer-plugin-api" : "1.*"
  },
  "require-dev" : {
    "phpunit/phpunit" : "4.*",
    "mockery/mockery" : "0.*"
  },
  "autoload" : {
    "psr-4" : {
      "TypedPHP\\Composer\\" : "source"
    }
  },
  "autoload-dev": {
    "files" : [
      "tests/TestCase.php"
    ],
    "psr-4" : {
      "TypedPHP\\Composer\\Tests\\" : "tests"
    }
  },
  "extra" : {
    "class" : [
      "TypedPHP\\Composer\\PathPlugin"
    ]
  }
}
```

This is from `composer.json`.

This is like many `composer.json` files. It has a name, a few dependencies, and some autoload directives. There is also a special type (`composer-plugin`) and a few bits in the `extras` object.

With this `composer.json` file, we're telling Composer this is a plugin. We're telling Composer to load the files in `extra.class`.

What do these files look like? Well that depends on what the plugin needs to do. For our purposes, there will be a plugin class and an installer class. The plugin class looks like this:

```
namespace TypedPHP\Composer;

use Composer\Composer;
use Composer\IO\IOInterface;
use Composer\Plugin\PluginInterface;

class PathPlugin implements PluginInterface
{
  public function activate(Composer $composer, IOInterface $io)
  {
    $installer = new PathPluginInstaller($io, $composer);

    $composer
      ->getInstallationManager()
      ->addInstaller($installer);
  }
}
```

This is from `source/PathPlugin.php`.

The `PathPlugin` class has an `activate()` method (as must all classes that implement `PluginInterface`). When this is called, the plugin will register a new installer instance with the installation manager. The installer does the heavy lifting:

```
namespace TypedPHP\Composer;

use Composer\Installer\LibraryInstaller;
use Composer\Package\PackageInterface;

class PathPluginInstaller extends LibraryInstaller
{
  public function getPackageBasePath(PackageInterface $package)
  {
    return "path/to/install";
  }

  public function supports($type)
  {
    return true;
  }
}
```

This is from `source/PathPluginInstaller.php`.

The plugin we're creating must alter the path in which the package's files are stored. This would usually be in `vendor`, so the value of `getPackageBasePath()` will replace `vendor`. Composer calls this method when deciding where to install package files to.

We want to allow all packages (that depend on this plugin) to be able to set their custom installation directories. Usually `supports()` would check the provided `$type` parameter, but we'll just return true. That way the plugin will activate for every package.

Defining Paths

We want the plugin to be able to match whole package names, or package names with wildcard placeholders. For that to happen, we'll need to define a helper method:

```
public function matches($string, $pattern)
{
  if ($pattern == $string) {
    return true;
  }

  $pattern = preg_quote($pattern, "#");
  $pattern = str_replace("\\*", ".*", $pattern);
  $pattern = "#^" . $pattern . "$#";

  return (boolean) preg_match($pattern, $string);
}
```

This is from `source/PathPluginInstaller.php`.

This method accepts a string and compares it to a pattern. The pattern looks like a string, but it becomes a regular expression. The wildcards (*) get turned into multi-character matches.

To illustrate this point; the pattern `"acme/*"` becomes the regular expression `"#^acme/.*$#"`. This will match the string `"acme/foo"` as well as `"acme/"`.

We can start identifying corresponding package names and using their custom install paths:

```
public function getPackageBasePath(PackageInterface $package)
{
  if ($packagePath = $this->getPackagePath($package)) {
    return $packagePath . "/" . $package->getName();
  }

  return parent::getPackageBasePath($package);
}

public function getPackagePath(PackageInterface $package)
{
  $extra = $package->getExtra();

  if (isset($extra["path"])) {
    return $extra["path"];
  }

  return null;
}
```

This is from `source/PathPluginInstaller.php`.

We check to see whether the `extra.path` key has been set and return it. Otherwise, the parent function is used to generate the default path (something under `vendor/*`).

This enables us to design `composer.json` files that contain code like the following:

```
"require" : {
  "typedphp/composer-path-plugin" : "*"
},
"extra" : {
  "path" : "tests"
}
```

Assuming the `composer-path-plugin` requirement is met, this package should install to the `tests` directory.

Overriding Paths

Building on this, we can add ways of overriding package install paths (from the root `composer.json` file):

```
public function getPackageBasePath(PackageInterface $package)
{
  $root = $this->composer->getPackage();
```

```
  if ($rootPath = $this->getRootPath($root, $package)) {
    return $rootPath . "/" . $package->getName();
  }

  if ($packagePath = $this->getPackagePath($package)) {
    return $packagePath . "/" . $package->getName();
  }

  return parent::getPackageBasePath($package);
}

public function getRootPath(
    PackageInterface $root,
    PackageInterface $package)
{
  $extra = $root->getExtra();
  $name  = $package->getName();

  if (isset($extra["paths"]) and is_array($extra["paths"])) {
    foreach ($extra["paths"] as $pattern => $path) {
      if ($this->matches($name, $pattern)) {
        return $path;
      }
    }
  }

  return null;
}
```

This is from `source/PathPluginInstaller.php`.

`getRootPath()` loops through `extra.paths` and checks each pattern against each dependency. To illustrate this:

```
"extra" : {
  "paths" : {
    "acme/*" : "acme"
  }
}
```

This is from `composer.json`.

This allows the main `composer.json` file to override custom paths set in dependencies. If they're not matched and they define their own custom path, that's where they will be installed to. If they do not define a custom path, it's off to `vendor` with them.

Implications

I designed this plugin just to handle a small set of modified packages. Path overriding and accepting every plugin made it possible to affect the paths of packages we don't control. Imagine we were making a new Laravel project. We could add the following to our `composer.json` file:

```
"require" : {
  "typedphp/composer-path-plugin" : "*",
  "laravel/laravel" : "4.2.*"
},
"extra" : {
  "paths" : {
    "symfony/*"    : ".",
    "illuminate/*" : "."
  }
}
```

This is from `composer.json`.

This would place the `symfony` and `illuminate` folders at the same level as `vendor`. We wouldn't need to change those dependencies to make it so.

Since the wildcards can be at the beginning of package names, you could also do something similar with themes:

```
"require" : {
  "typedphp/composer-path-plugin" : "*"
},
"extra" : {
  "paths" : {
    "*/*-theme" : "public/themes"
  }
}
```

This is from `composer.json`.

Example: Hook Plugin

PHP frameworks like to define configuration files in which to add extension classes and hooks. Laravel (http://laravel.com), for instance, has PHP files that return arrays. When you install a new extension, you need to add a service provider (http://laravel.com/docs/packages#service-providers) class to a specific configuration file. It's easy to install the plugin, but always requires manual intervention.

This technique isn't specific to Laravel. It's such a common problem there and involves some tricky file manipulation. A good example, in other words!

You can find the code for this section at https://github.com/typedphp/composer-hook-plugin.

Getting Started

We begin by creating a new `composer.json` file:

```
{
  "name" : "typedphp/composer-hook-plugin",
  "type" : "composer-plugin",
  "license" : "MIT",
  "authors" : [
    {
      "name" : "Christopher Pitt",
      "email" : "cgpitt@gmail.com"
    }
  ],
  "require" : {
    "php" : ">=5.5.0",
    "composer-plugin-api" : "1.*"
  },
  "require-dev" : {
    "phpunit/phpunit" : "4.*",
    "mockery/mockery" : "0.*"
  },
  "autoload" : {
    "psr-4" : {
      "TypedPHP\\Composer\\" : "source"
    }
  },
```

```
  "autoload-dev": {
    "files" : [
      "tests/TestCase.php"
    ],
    "psr-4" : {
      "TypedPHP\\Composer\\Tests\\" : "tests"
    }
  },
  "extra" : {
    "class" : [
      "TypedPHP\\Composer\\HookPlugin"
    ]
  }
}
```

This is from `composer.json`.

You'll notice that we've added the `type` and `extra` data, just as we did for the path plugin. This will be a common theme in plugins we develop. Following this, we create the basic plugin loader class:

```
namespace TypedPHP\Composer;

use Composer\Composer;
use Composer\IO\IOInterface;
use Composer\Plugin\PluginInterface;

class HookPlugin implements PluginInterface
{
  public function activate(Composer $composer, IOInterface $io)
  {
    $installer = new HookPluginInstaller($io, $composer);

    $composer
      ->getInstallationManager()
      ->addInstaller($installer);
  }
}
```

This is from `source/HookPlugin.php`.

Describing Configuration

This is like what we did with the path plugin. The important differences are in the `HookPluginInstaller` class. What we want to be able to do is define custom `extra` data to be applied to configuration files. Something resembling the following:

```
"require": {
  "typedphp/composer-hook-plugin": "dev-master"
},
"extra": {
  "hooks": [
    {
      "file": "app/config/app.php",
      "key": "providers",
      "classes": [
        "TypedPHP\\ServiceProvider"
      ]
    },
    {
      "file": "app/config/app.php",
      "key": "facades",
      "classes": {
        "Arr": "TypedPHP\\Types\\Facades\\ArrayFacade",
        "Num": "TypedPHP\\Types\\Facades\\NumberFacade",
        "Str": "TypedPHP\\Types\\Facades\\StringFacade",
        "Bool": "TypedPHP\\Types\\Facades\\BooleanFacade"
      }
    }
  ]
}
```

This is from a different `composer.json` file.

This data should then be added to the configuration files, without erasing any comments or formatting from the same configuration file.

This presents a problem. The only easy way to append data to these configuration files is to read the data, append the array items, and rewrite the file.

Altering Configuration Files

Using `var_export()` will remove all comments and formatting, so we're going to have to try harder. The approach I want to take is to read the file data and identify the exact point at which we can insert new data. Something like the following:

```
namespace TypedPHP\Composer;
```

```
use Composer\Installer\LibraryInstaller;
use Composer\Package\PackageInterface;
use Composer\Repository\InstalledRepositoryInterface;

class HookPluginInstaller extends LibraryInstaller
{
  public function install(
    InstalledRepositoryInterface $repository,
    PackageInterface $package
  )
  {
    // 1. get the hooks and pass each to $this->addHookToFile()
  }

  protected function addHookToFile($key, array $classes, $file)
  {
    // 1. get the array and string data of the file
    // 2. get the previous classes in the array
    // 3. find the location to add new classes
    // 4. insert the new classes into the old string content
    // 5. write the new string content to the same file
  }
}
```

This is from `source/HookPluginInstaller.php`.

Listing the objectives like that has the potential to make it sound less complicated. There's still much to do to make this process work. For instance, we need a way to get the last item in the key we want to update. If we're going to add new classes to a specific key, we need to find the last item in the array we want to add the classes to:

```
public function getArrayValueByKey(array $array, $key)
{
  if (isset($array[$key])) {
    return $array[$key];
  }

  foreach (explode(".", $key) as $segment) {
    if (!array_key_exists($segment, $array)) {
      return null;
    }

    $array = $array[$segment];
  }

  return $array;
}
```

This is from `source/HookPluginInstaller.php`.

This function will get all the items from a specific key. To get the last one we'll only need to use `end($items)` on the results of that method. Getting the insertion point can look something like:

```
protected function getInsertionIndex(array $items, $source)
{
  $last   = end($items);
  $single = strpos($source, $last);
  $double = strpos($source, str_replace("\\", "\\\\", $last));

  if (is_numeric($single)) {
    $index = strpos($source, "\n", $single);
  }

  if (is_numeric($double)) {
    $index = strpos($source, "\n", $double);
  }

  if ($index) {
    return $index;
  }

  return -1;
}
```

This is from `source/HookPluginInstaller.php`.

Half the job is using `strpos()` to find the start of the last item and the other is accounting for both class formats (i.e., `Foo\\Bar` and `Foo\Bar`).

We should also throw in a bit of validation, so malformed hooks don't cause exceptions in the installation process:

```
protected function addHook(array $hook)
{
  if (empty($hook["key"])) {
    return;
  }

  if (empty($hook["classes"])) {
    return;
  }
```

```
  if (empty($hook["file"]) or !file_exists($hook["file"])) {
    return;
  }

  $this->addHookToFile(
    $hook["key"],
    $hook["classes"],
    $hook["file"]
  );
}
```

This is from `source/HookPluginInstaller.php`.

Finally, we can use all these together to create the method that gets the hooks, finds the insertion point, and builds the new file:

```
public function install(
  InstalledRepositoryInterface $repository,
  PackageInterface $package
)
{
  $hooks = [];
  $extra = $package->getExtra();

  if (isset($extra["hooks"])) {
    $hooks = $extra["hooks"];
  }

  foreach ($hooks as $hook) {
    $this->addHook($hook);
  }

  parent::install($repository, $package);
}

protected function addHookToFile($key, array $classes, $file)
{
  $data     = include($file);
  $source   = file_get_contents($file);
  $previous = $this->getArrayValueByKey($data, $key);

  if (empty($previous)) {
    return;
  }
```

```
  $index    = $this->getInsertionIndex($previous, $source);
  $append   = $this->addClasses($classes, $previous);
  $modified = "";

  if (count($append)) {
    $modified .= substr($source, 0, $index);

    if ($modified[strlen($modified) - 1] == ",") {
      $modified .= "\n";
    } else {
      $modified .= ",\n";
    }

    $new = "";

    foreach ($append as $key => $value) {
      if (is_string($key)) {
        $new .= "'{$key}' => {$value},\n";
      } else {
        $new .= "{$value},\n";
      }
    }

    $modified .= trim($new);
    $modified .= substr($source, $index);

    file_put_contents($file, $modified);
  }
}

protected function addClasses(array $classes, array $previous)
{
  $append = [];

  foreach ($classes as $key => $value) {
    if (is_string($key)) {
      if (!isset($previous[$key])) {
        $append[$key] = "'{$value}'";
      }
    } else {
      if (!in_array($value, $previous)) {
        $append[] = "'{$value}'";
      }
    }
  }

  return $append;
}
```

This is from `source/HookPluginInstaller.php`.

Libraries only need the hook plugin and define that `extra` data to hook their own classes into configuration files.

I've avoided any code that deals specifically with the indentation/formatting of the modified classes. The goal is to preserve the existing formatting. Consider adding your own indentation/customization features!

Implications

This plugin makes it easy for developers to install extensions/modules in existing applications. It would be easy to change this plugin to ask developers if they would like the hooks applied before applying them.

PHP Implementation

Functions

As mentioned in Chapter 2, the most usable way to package this functionality is first in a set of functions.

Type Functions

These functions are exactly the same as described in Chapter 2, so I'll not go into much detail here. The complete listing is:

```
namespace TypedPHP\Functions\TypeFunctions;

function isNumber($variable)
{
  return is_integer($variable) or is_float($variable);
}

function isBoolean($variable)
{
  return is_bool($variable);
}

function isNull($variable)
{
  return is_null($variable);
}
```

```
function isObject($variable)
{
  return is_object($variable) and !isFunction($variable);
}

function isFunction($variable)
{
  return is_callable($variable) and is_object($variable);
}

function isExpression($variable)
{
  $isNotFalse = @preg_match($variable, "") !== false;
  $hasNoError = preg_last_error() === PREG_NO_ERROR;

  return $isNotFalse and $hasNoError;
}

function isString($variable)
{
  return is_string($variable) and !isExpression($variable);
}

function isResource($variable)
{
  return is_resource($variable);
}

function isArray($variable)
{
  return is_array($variable);
}

function getType($variable)
{
  $functions = [
    "isNumber" => "number",
    "isBoolean" => "boolean",
    "isNull" => "null",
    "isObject" => "object",
    "isFunction" => "function",
    "isExpression" => "expression",
    "isString" => "string",
    "isResource" => "resource",
    "isArray" => "array"
  ];

  $result = "unknown";
```

```
  foreach ($functions as $function => $type) {
    $namespace = "TypedPHP\\Functions\\TypeFunctions\\";
    $function = $namespace . $function;

    if ($function($variable)) {
      $result = $type;
      break;
    }
  }

  return $result;
}
```

You can find the most recent code, along with the tests, on GitHub (https://github.com/typedphp/type-functions).

String Functions

Since expressions are basically special strings, the string functions we need are a little more complicated than the other types. They follow this pattern:

```
function isExpression($variable)
{
  return TypeFunctions\isExpression($variable);
}

function indexOf($haystack, $needle, $offset = 0)
{
  if (isExpression($needle)) {
    return indexOfExpression($haystack, $needle, $offset);
  }

  return indexOfString($haystack, $needle, $offset);
}

function indexOfString($haystack, $needle, $offset = 0)
{
  $index = -1;
  $match = strpos($haystack, $needle, $offset);

  if ($match !== false) {
    $index = $match;
  }

  return $index;
}
```

```
function indexOfExpression($haystack, $needle, $offset = 0)
{
  $index = -1;

  $match = preg_match(
    $needle, $haystack, $matches,
    PREG_OFFSET_CAPTURE, $offset
  );

  if ($match) {
    $index = $matches[0][1];
  }

  return $index;
}
```

In most of the string functions, we'll need to do something different if the string is an expression. It's easy for us to assume a pattern of `action[Of|With]String` and `action[Of|With]Expression` and call the correct one from a common entry function. We have the following entry functions:

```
function startsWith($haystack, $needle)
{
  if (isExpression($needle)) {
    return startsWithExpression($haystack, $needle);
  }

  return startsWithString($haystack, $needle);
}

function endsWith($haystack, $needle)
{
  if (isExpression($needle)) {
    return endsWithExpression($haystack, $needle);
  }

  return endsWithString($haystack, $needle);
}

function matches($haystack, $needle)
{
  if (isExpression($needle)) {
    return matchesExpression($haystack, $needle);
  }

  return matchesString($haystack, $needle);
}
```

```
function slice($string, $offset = 0, $limit = 0)
{
  if ($limit == 0) {
    return substr($string, $offset);
  }

  return substr($string, $offset, $limit);
}

function trim($haystack, $needle)
{
  if (isExpression($needle)) {
    return trimWithExpression($haystack, $needle);
  }

  return trimWithString($haystack, $needle);
}

function trimLeft($haystack, $needle)
{
  if (isExpression($needle)) {
    return trimLeftWithExpression($haystack, $needle);
  }

  return trimLeftWithString($haystack, $needle);
}

function trimRight($haystack, $needle)
{
  if (isExpression($needle)) {
    return trimRightWithExpression($haystack, $needle);
  }

  return trimRightWithString($haystack, $needle);
}
```

There's a special case for splitting strings:

```
function split($haystack, $needle = null, $limit = 0)
{
  if ($needle === null) {
    return splitWithNull($haystack, $limit);
  }

  if (isExpression($needle)) {
    return splitWithExpression($haystack, $needle, $limit);
  }

  return splitWithString($haystack, $needle, $limit);
}
```

When the $needle parameter is not passed, we default to creating an array out of characters of a string. There's also a special case for replacing substrings:

```
function isArray($variable)
{
  return TypeFunctions\isArray($variable);
}

function replace($haystack, $needle, $replacement)
{
  if (isArray($needle) and isArray($replacement)) {
    return replaceWithArray(
      $haystack, $needle, $replacement
    );
  }

  if (isExpression($needle)) {
    return replaceWithExpression(
      $haystack, $needle, $replacement
    );
  }

  return replaceWithString(
    $haystack, $needle, $replacement
  );
}
```

If the $needle and $replacement parameters are both arrays, then we emulate the current behavior of str_replace/preg_replace.

You can find the most recent code, along with the tests, on GitHub (https://github.com/typedphp/string-functions).

Number Functions

The number functions combine integer and floating-point behavior and create a consistent naming convention for many current PHP functions. There are rounding functions:

```
function round($number)
{
  return (float) \round($number);
}
```

```
function ceiling($number)
{
  return (float) \ceil($number);
}

function floor($number)
{
  return (float) \floor($number);
}
```

The built-in rounding functions return integer values. These return floating-point values. When it comes to wrapping this functionality in an object, we're going to use `SplFloat`, so we might as well return floats here. The rounding is still happening, so values like `5.5` become `5.0` or `6.0`.

Then there are a few conversion functions:

```
function degrees($number)
{
  return (float) \rad2deg($number);
}

function radians($number)
{
  return (float) \deg2rad($number);
}
```

We're using more consistent names and returning floating-point values. Then there are wrappers for the trigonometry functions:

```
function sine($number)
{
  return (float) \sin($number);
}

function inverseSine($number)
{
  return (float) \asin($number);
}

function hyperbolicSine($number)
{
  return (float) \sinh($number);
}

function inverseHyperbolicSine($number)
{
  return (float) \asinh($number);
}
```

Finally, there are a few utility methods, such as:

```
function limit($number, $min, $max)
{
  if ($number < $min) {
    return $min;
  }

  if ($number > $max) {
    return $max;
  }

  return $number;
}
```

You can find the most recent code, along with the tests, on GitHub (https://github.com/typedphp/number-functions).

Array Functions

The array functions are mostly simple wrappers around the already extensive set of built-in functions. The hard part will be integrating them into an object later. The complete listing is:

```
namespace TypedPHP\Functions\ArrayFunctions;

use TypedPHP\Functions\NumberFunctions;

function contains(array $haystack, $needle)
{
  return in_array($needle, $haystack);
}

function each(array $array, callable $callback)
{
  array_walk($array, $callback);

  return $array;
}

function exclude(array $array, array $exclude)
{
  return array_diff($array, $exclude);
}
```

```
function filter(array $array, callable $callback)
{
  return array_filter($array, $callback);
}

function length(array $array)
{
  return count($array);
}

function has(array $array, $needle)
{
  return array_key_exists($needle, $array);
}

function join(array $array, $glue)
{
  return \join($glue, $array);
}

function map(array $array, callable $callback)
{
  return array_map($callback, $array);
}

function merge(array $array, array $merge)
{
  return array_merge($array, $merge);
}

function slice(array $array, $offset = 0, $limit = 0)
{
  if ($limit == 0) {
    return array_slice($array, $offset);
  }

  return array_slice($array, $offset, $limit);
}

function random(array $array)
{
  if (length($array) === 0) {
    return null;
  }

  $index = NumberFunctions\random(0, length($array) - 1);

  return $array[$index];
}
```

You can find the most recent code, along with the tests, on GitHub (https://github.com/typedphp/array-functions).

Conclusion

In this chapter, we learned about how to create Composer plugins. These are useful for when we need to perform setup for our library that would otherwise require manual changes on behalf of the developers wanting to use it.

We also looked at some of the core functions for our library. If you've found these useful and have a particular preference for the C extension you'd prefer to package them with, consider it a challenge to extend the basics with your own polish.

Thank you for reading this far. I hope you've found what you've read helpful, and that this knowledge serves you well in years to come.

Index

C. Pitt, *Typed PHP*, DOI 10.1007/978-1-4842-2114-3

V, W, X, Y, Z