Mastering Object Oriented PHP

By Brandon Savage

Copyright

This version of the book was published on March 7, 2013.

Books like this are possible by the time and investment made by the authors. If you received this book but did not purchase it, please consider making future books possible by buying a copy at http://masterobjectorientedphp.com/ today.

Credits

Credits to reviewing parts of Chapters 2 and 3 goes to Anthony Fererra. He also provided the sample code in Chapter 3.

The sample code for Doctrine came from Johnathan Wage, Roman Borschel, Matthew Weier O'Phinney, Kris Wallsmith and Fabien Portencier. The code is licensed under the MIT License and used with permission.

Table of Contents

Chapter 6: Typehinting Your Way To Success

Since the beginning, PHP has always been a "loosely typed" language. What this means that most any type could fairly easily be converted or treated like any other type. This makes perfect sense when dealing with a web language, where all POST and GET data is treated as a string; as a result, PHP has always lacked the sort of typehinting that Java and other languages offered to specifically hint for a particular data type at runtime.

But when PHP introduced it's new object model in PHP 5, there was the option to typehint on objects.[19] Since objects specifically possess a certain type, specific to their class and parent classes, it's possible to definitely enforce which type is being passed around. This offers the object oriented developer an advantage over functional developers.

[19] http://php.net/manual/en/language.oop5.typehinting.php

What Typehinting Is And How It Works

Typehinting is essentially a method of identifying the type of object that is being passed into a method. Most of us already do this to some extent manually when evaluating data that's been passed in from a webform: we examine it to make sure that it contains all numeric characters for a zip code or we ensure that it matches the format for a phone number. But typehinting happens at the PHP parser level, and it is data validation for developers.

Typehints precede variable declarations in the method signature or function signature in objects or functional programs. For example:

```
<?php

class MyClass {
    public function __construct(MyOtherClass $moc) {
        // Do something here
    }
}

?>
```

Notice that in the __construct() method, I have placed the word "MyOtherClass" immediately preceding the $moc variable declaration. When PHP instantiates this object and executes the constructor, if $moc is defined to be any data type or object other than MyOtherClass, a fatal error will be produced.

This is useful because knowing in advance what object type will be passed in allows us to ensure that certain APIs are available to us. For example, if you know that MyOtherClass has a method called printThisString(), you can reliably count on MyOtherClass::printThisString() being available when you

typehint for MyOtherClass. If typehinting did not exist and the method accepted an object of any type, developers could run into a scenario where the expected API was not present, and a fatal error could result.

The Role of Inheritance On Object Types

It's important to understand the role that inheritance plays on object types. Objects that inherit from other objects have multiple types, as many types as they have ancestors. For example:

```
<?php

class MyClass {

}

class MySecondClass extends MyClass {

}

class MyThirdClass extends MySecondClass {

}

$obj = new MyThirdClass();

var_dump(($obj instanceof MyThirdClass));  // true
var_dump(($obj instanceof MySecondClass)); // true
var_dump(($obj instanceof MyClass));       // true
var_dump(($obj instanceof stdClass));      // false

?>
```

Because $obj was an instance of MyThirdClass, which inherited from two parents, it automatically became an instance of those other classes. So, if we were typehinting for MySecondClass, the object would pass:

```
<?php

$obj = new MyThirdClass();

function testFunction(MySecondClass $obj) {
    print get_class($obj);
}

testFunction($obj); // MyThirdClass

?>
```

Even though the object reports being of class MyThirdClass, the typehinting engine permits it to pass as a MySecondClass object, because of its inheritance.

Traits in PHP 5.4 are not considered class types, and typehinting will not work on a trait (though you can test for traits with a class_uses() function).

Using Typehints To Enforce API Constraints

Typehinting is most useful when enforcing a particular API design onto a particular object. For example, you may want to enforce a particular set of database methods onto a data object, and you can use a typehint to do this:

```
<?php

class MyDataObject {

    public function __construct(DatabaseObject $dbo) {

    }

}

?>
```

The enforcement of the particular object type allows you to expect certain methods and rely upon their definition, as long as they remain part of the defined object.

When typehinting for objects that extend other objects, it is important not to typehint too far up the inheritance tree, but instead only typehint for the API required. For example, many databases contain similar behaviors; if your application supports both PostgreSQL and MySQL, you want to typehint on the lowest common denominator, or BaseDatabaseObject rather than on MysqlDatabaseObject or PostgresqlDatabaseObject. Typehinting on BaseDatabaseObject allows you to enforce the API defined in the base object, but use either MySQL or Postgres.

Using Interfaces In Typehinting

PHP allows the definition of interfaces. These interfaces are not classes, and they cannot be instantiated directly. Interfaces contain only a definition of a public method's signature and the public API for a class that implements the interface. They contain no implementation details, and cannot define protected or private methods.

Interfaces are created differently from classes; they must be designated as interfaces, rather than as classes. For example:

```
<?php
interface DatabaseInterface {
    public function query($sql);
    public function cleanData($data);
```

```
    public function get($sql, $args);

    public function set ($sql, $args);

    public function delete ($sql, $args);

}

?>
```

Interfaces are implemented in classes, and unlike extension, which only permits a single parent class, multiple interfaces may be implemented in a given class.

```
<?php

class BaseDatabaseObject implements DatabaseInterface,
   SecondInterface {
    // All the methods required defined here
}
```

Interfaces require that all defined methods be defined and fleshed out in objects that can be instantiated. A fatal error is produced if an object does not define a method required by the interface.

Interfaces can be typehinted by PHP. This is the true power of defining an interface: any and all objects that implement that particular interface will be types of that interface. This allows you to define common APIs for objects that may implement a given interface, and then typehint for the interface alone and ensure that methods are available.

Because interfaces are essentially the lowest level that can be defined and typehinted against, they are effective ways to ensure that the objects passed

into a function or method are consistent in their API, even if they are varied in their specific implementation details.

The Benefits of Typehinting

Typehinting allows a developer to enforce constraints on their code, which overall improves the interoperability of the code. By enforcing constraints, especially in situations involving dependency injection, developers can know in advance the API that will be included, and utilize that API, even when the underlying code may not yet be developed (especially useful in cases where other developers implement specific APIs, such as in plugins or data object cases).