

**Discover
object-oriented
programming**

using WORDPRESS

Carl Alexander

This is a sample from

Discover object-oriented programming using WordPress

To learn more about the book and read the rest, head over to
<https://carlalexander.ca/object-oriented-programming-wordpress/>

Contents

<i>Foreword</i>	<i>vii</i>
<i>Preface</i>	<i>x</i>
<i>Is this book right for you?</i>	<i>xiii</i>



THE OBJECT-ORIENTED MINDSET	1
Let's talk about "learning object-oriented programming"	3
So why can't you learn it?	4
You're never done learning about something	5
You're here because of one of those objectives	6
How can you get back on track?	7
What makes for a good objective?	8
Find a problem that you care about	10
How do you write code right now?	11
A practical example	12
Steps don't change with the context	14

Contents

This isn't a bad way to program	15
How object-oriented programming solves problems	16
Programming with Legos	16
Why is it hard to think this way?	17
Creativity and object-oriented programming	19
Managing your expectations	22
	
FEATURES OF OBJECT-ORIENTED PROGRAMMING	23
Encapsulation	25
What is encapsulation?	25
Encapsulation in real life	26
Classes vs objects	27
How PHP supports encapsulation	28
Inheritance	35
Inheritance is about relationships	35
Inheritance in PHP	36
Interfaces	38
Abstract classes	44
Concrete classes	48
Overriding methods	49

Contents

The final keyword	51
Getting objects of the right class	53
Polymorphism	58
Polymorphism explained	58
Polymorphism in the previous chapter	59
A great way to detect polymorphism	60
More on this in the next section	61
	
PUTTING IT INTO PRACTICE	62
Using polymorphism with abstract classes	64
WordPress admin pages	64
A note on the options class	69
Analyzing our admin page classes	71
What's the same?	72
What's different?	74
Creating our abstract class	78
Adding a constructor	79
Registering an admin page	80
Rendering our form	85
MyPlugin_OptionsPage class	88

Contents

MyPlugin_ShortcodePage class	94
One more thing	96
We're entering design territory	99
Using polymorphism with interfaces	100
Reviewing interfaces	100
How does polymorphism work with interfaces?	101
The challenge of using interfaces	103
An example: WordPress Plugin API Manager	106
Wrapping up our look at polymorphism	124
Sketching	126
Enumerate your classes	126
Turning your sketches to code	127
Add more classes	135
Do this in whichever way you're comfortable with	137
	
CLOSING THOUGHTS	139
Reviewing the process	141
Find a problem	141
Break it down into classes	141
Work on the problem	142

Contents

Review how things went	142
Rince and repeat	143
You still have to find your own way	144
This isn't the end	146
A small favour	146
About the author	148

Encapsulation

Encapsulation is the feature that people tend to associate the most with object-oriented programming. That's because most modern programming languages support encapsulation using classes. And classes are the foundation of everything that we'll do with object-oriented programming.

It's also the reason why we're looking at it first. All the other features of object-oriented programming depend on it. That said, it doesn't do too much on its own either.

What is encapsulation?

The idea behind encapsulation is pretty straightforward. We want to group data and behaviour inside a single entity. (Or capsule! *hint hint*) In most object-oriented programming languages, we call these entities classes.

But why do we want to group data and behaviour inside a class? The answer is to control who has access to them. This is an often misunderstood property of classes.

We often think of classes as just simple containers for our data and behaviour. But they do a bit more than that. They're more like a container with a security system.

Encapsulation

The goal of this security system is to prevent other developers from tampering with things. The class only gives them access to the data and behaviour that you want them to have access to and nothing more. This results in your code being stronger and less error-prone because there are fewer ways to break things.

Encapsulation in real life

So why is encapsulation such an important feature of object-oriented programming? Well, to better understand why that is, let's look at encapsulation from a different angle. Let's imagine that we have an oven.

When you set it to 400 degrees, you expect the displayed temperature to be accurate. You don't want to know how the oven got the information or how it processed it. You just want the ability to set and view your oven's temperature.

That's the purpose behind encapsulation as well. As a user of a class, you don't want to know what goes on inside it. You just want to use it and get a behaviour that's consistent each time. That's all that matters at the end of the day.

And that's also why encapsulation is so important. It's easier to code a class if you control the data and behaviour that's available to others. That way you're sure that other developers can't tamper with them.

Instead, they have no choice but to access that data and behaviour

the way you intended them to. This is the same as the display and controls of the oven. It's the only way that the oven manufacturer lets you control its temperature.

Classes vs objects

At this point, we've talked a lot about classes in the context of encapsulation. But we can't forget that this is object-oriented programming. (And not class-oriented programming!) So this begs the question, "What are objects to classes?"

Classes are like a template for making objects. This template defines the properties and behaviour of all objects that use it. When we create an object, we say that it was "instantiated" because it's an "instance" of a class. (This is terminology that you'll hear a lot with object-oriented programming.)

But if we go back to our Lego analogy, classes would be the concept of the Lego brick type. A Lego brick type is a way to group common Lego brick characteristics. For example, rectangular Lego bricks always have a specific width, length and colour.

Meanwhile, an object would be an individual rectangular Lego brick. We mentioned earlier that one could be a yellow 1x12 brick and another could be a green 2x2 brick. Both of those bricks are objects with different properties.

How PHP supports encapsulation

Alright, so we've done a pretty good job going over encapsulation as a concept. Now, let's take a look at how PHP supports this object-oriented feature. The best way to do that is by using it in an example!

The "class" keyword

The `class` keyword is central to using encapsulation with PHP. It's what we use to combine data and behaviour. Without that capability, we can't build something using object-oriented programming.

```
class MyPlugin_Options
{
    // ...
}
```

An empty class like the `MyPlugin_Options` class above is the same as a blank page in object-oriented programming. It's always our starting point when building class. You'll see a lot of those throughout this book.

You'll also notice that we didn't name our class `Options` but `MyPlugin_Options`. We always want to prefix our classes with something like the name of our plugin. That's because PHP doesn't allow classes to have the same name. So prefixing our classes helps to prevent any naming conflicts.

Now, what does the `MyPlugin_Options` class do? Well, its job is

going to be to store the options of our plugin. But that's hard to see at the moment. We still haven't built it out yet.

Properties

Now that we have a class, we need to store data inside it. In PHP, that's done using properties. Properties are variables that are specific to a class.

We alluded to them earlier in our Lego analogy. They're the specific width, length and colour of a Lego piece. This is good for a Lego piece. But, if we go back to our `MyPlugin_Options` class, what properties should it have?

```
class MyPlugin_Options
{
    /**
     * Plugin options.
     *
     * @var array
     */
    var $options = array();
}
```

Well, it should have a way to store all the options of our plugin. So let's create a property to do that. To keep things simple, we're going to make that property an array, and we'll name it `options`.

You'll also notice that we initialized it as an empty array. If we hadn't done that, PHP would have given us an error when we would've tried to use `options` as an array. That's why it's important to initialize properties of a class.

Methods

Methods are the other important piece of the encapsulation feature in PHP. Methods are where you define the behaviour of a class. They're also the primary way that others will interact with your class and its properties.

From a technical point of view, methods are like the PHP functions that you use every day. The big difference is that they're specific to a class. You can't use them without using the class or object where you defined them.

Methods also have direct access to properties of the class where you defined them. This is why we can use them to protect our class properties from tampering. Let's go back to our `MyPlugin_Options` class to see how we can do that.

```
class MyPlugin_Options
{
    /**
     * Plugin options.
     *
     * @var array
     */
    var $options = array();

    /**
     * Get the plugin option with the given name.
     *
     * @param string $name
     * @param null $default
     *
     * @return mixed|null
     */
    function get($name, $default = null)
    {
        if (!$this->has($name)) {
            return $default;
        }

        return $this->options[$name];
    }
}
```

Encapsulation

```
/**
 * Set the plugin option using the given name.
 *
 * @param string $name
 * @param mixed $value
 */
function set($name, $value)
{
    $this->options[$name] = $value;
}
}
```

As you can see, we added two methods to the `MyPlugin_Options` class above. These methods control access to the `options` variable that we defined earlier. Let's go over them to see what they do.

The first method is the `get` method. It's a type of method that we call a getter method. We use it to fetch an option from the `options` array.

Meanwhile, the second method in the `MyPlugin_Options` class is the `set` method. It let us change an option in the `options` array. Like the `get` method, we often refer to methods like it as a setter method.

Visibility

So far, we haven't protected the `options` property that we added earlier. Sure, we created methods so that others could interact with it. But they still can access the variable without using those methods if they want to.

This is where the concept of visibility comes in. It's a concept that's unique to object-oriented programming. And it's what lets us con-

Encapsulation

control access to the properties and methods of our class.

Visibility uses access levels to determine if you can access a property or method of a class. You can only assign an access level on a per-property or per-method basis. There's no way to assign an access level to a group of properties or methods.

There are three possible access levels that you can use with visibility. In order of least restrictive to most restrictive, they are:

- Public
- Protected
- Private

`Public` gives anyone access to a property or method. It's also the default access level if you don't specify one. That's why anyone could access the `options` variable that we created earlier. Since we didn't assign it an access level, it defaulted to `public`.

On the other end of the spectrum, we have `private`. This is the most restrictive access level. It says that you can only access a property or method within the class that defined it.

Meanwhile, `protected` is almost the same as `private`. We use it with inheritance which is the object-oriented feature that we're going to look at next. We'll look at the difference between `private` and `protected` then.

How to use visibility

So we've seen what visibility is. But let's not forget about our `MyPlugin_Options` class! We still need to fix it.

```
class MyPlugin_Options
{
    /**
     * Plugin options.
     *
     * @var array
     */
    private $options = array();

    /**
     * Get the plugin option with the given name.
     *
     * @param string $name
     * @param null $default
     *
     * @return mixed|null
     */
    public function get($name, $default = null)
    {
        if (!$this->has($name)) {
            return $default;
        }

        return $this->options[$name];
    }

    /**
     * Set the plugin option using the given name.
     *
     * @param string $name
     * @param mixed $value
     */
    public function set($name, $value)
    {
        $this->options[$name] = $value;
    }
}
```

So above is our secured `MyPlugin_Options` class. We made two changes to it. First, we made the `options` variable private. This prevents anyone but the `MyPlugin_Options` class from having direct access to it.

Encapsulation

Second, we added the `public` keyword in front of the `get` and `set` methods. This is technically an optional step since the two methods were already public by default. That said, it's a good practice to define the visibility of your properties or methods. (In fact, you should always try to do it!)