Learn Object Oriented Programming (OOP) in PHP

By: Stefan Mischook - September 07 2007

www.killerphp.com - www.killersites.com - www.idea22.com

Preamble:

The hardest thing to learn (and teach btw,) in object oriented PHP … is the basics. But once you get them under-your-belt, the rest will come much, much easier.

But don't be discouraged! You just found the easiest to understand tutorial out there on OOP and PHP. It may sound like a boastful claim … I know. But that's what the nerd zeitgeist is saying.

… Or so I've been told.

Videos:

As an extra bonus, I've created a few video tutorials for you. They cover the same material as the written article and are designed to reinforce the article.

- Introduction to Object Oriented PHP (4:05)
- Why learn Object Oriented PHP (14:46)
- Objects and Classes in PHP (5:26)
- Build Objects in PHP - Part 1 (9:14)
- Build Objects in PHP - Part 2 (9:41)
- Build Objects in PHP - Part 3 (6:18)

If you have questions/comments, you can contact me at: **stefan@killersites.com**

Thanks,

Stefan Mischook

Learn Object Oriented Programming (OOP) in PHP

Object-Oriented Programming (OOP) is a type of programming added to php5 that makes building complex, modular and reusable web applications that much easier.

With the release of php5, php programmers finally had the power to code with the 'big boys'. Like Java and C#, php finally has a complete OOP infrastructure.

In this tutorial, you will be guided (step-by-step) through the process of building and working with objects using php's built-in OOP capabilities. At the same time you will learn:

- The difference between building a php application the old fashioned (procedural) way, versus the OOP way.
- What the basic OOP principles are, and how to use them in PHP.
- When you would want to use OOP in your PHP scripts.

People run into confusion when programming because of some lack of understanding of the basics. With this in mind, we are going to slowly go over key OOP principles while creating our own PHP objects. With this knowledge, you will be able to explore OOP further.

For this tutorial, you should understand a few PHP basics: functions, variables, conditionals and loops.

To make things easy, the tutorial is divided into 23 steps.

Step 1:

First thing we need to do is create two PHP pages:

index.php
class_lib.php

OOP is all about creating modular code, so our object oriented PHP code will be contained in dedicated files that we will then insert into our normal PHP page using php 'includes'. In this case all our OO PHP code will be in the PHP file:

class_lib.php

OOP revolves around a construct called a 'class'. Classes are the cookie-cutters / templates that are used to define objects.

Step 2:

Create a PHP class

Instead of having a bunch of functions, variables and code floating around willy-nilly, to design your php scripts or code libraries the OOP way, you'll need to define/create your own classes.

You define your own class by starting with the keyword 'class' followed by the name you want to give your new class.

```
<?php
```

```
class person {



}

?>
```

Step 3:

Add data to your class

Classes are the blueprints for php objects - more on that later. One of the big differences between functions and classes is that a class contains both data (variables) and functions that form a package called an: 'object'. When you create a variable inside a class, it is called a 'property'.

```
<?php

class person {

	var name;



}

?>
```

Note: The data/variables inside a class (ex: *var name;*) are called 'properties'.

Step 4:

Add functions/methods to your class

In the same way that variables get a different name when created inside a class (they are called: properties,) functions also referred to (by nerds) by a different name when created inside a class - they are called 'methods'.

A classes' methods are used to manipulate its' own data / properties.

```
<?php

class person {

	var $name;

	function set_name($new_name) {

		$this->name = $new_name;

	}
```

```
	function get_name() {

		return $this->name;
	}

}
?>
```

Note: Don't forget that in a class, variables are called 'properties'.

Step 5

Getter and setter functions

We've created two interesting functions/methods: get_name() and set_name().

These methods follow a common OOP convention that you see in many languages (including Java and Ruby) - where you create methods to 'set' and 'get' properties in a class.

Another convention (a naming convention,) is that getter and setter names should match the property names.

```
<?php

class person {

	var $name;

	function set_name($new_name) {

		$this->name = $new_name;

	}


	function get_name() {

		return $this->name;

	}


}

?>
```

Note: Notice that the getter and setter names, match the associated property name.

This way, when other PHP programmers want to use your objects, they will know that if you have a method/function called 'set_name()', there will be a property/variable called 'name'.

Step 6:

The '$this' variable

You probably noticed this line of code:

$this->name = $new_name.

The $this is a built-in variable (built into all objects) which points to the current object. Or in other words, $this is a special self-referencing variable. You use $this to access properties and to call other methods of the current class.

```
function get_name() {

            return $this->name;

      }
```

Note: This may be a bit confusing for some of you … that's because you are seeing for the first time, one of those built in OO capabilities (built into PHP5 itself) that automatically does stuff for us.

For now, just think of $this as a special OO PHP keyword. When PHP comes across $this, the PHP engine knows what to do.

… Hopefully soon, you will too!

Step 7:

Include your class in your main PHP page.

You would never create your PHP classes directly inside your main php pages - that would help defeat the purposes of object oriented PHP in the first place!

Instead, it is always best practice to create separate php pages that only contain your classes. Then you would access your php objects/classes by including them in your main php pages with either a php 'include' or 'require'.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
1" />
<title>OOP in PHP</title>

<?php include("class_lib.php"); ?>

</head>

<body>
```

```
</body>
</html>
```

Note: Notice how we haven't done anything with our class yet. We will do that next.

Step 8:

Instantiate/create your object

Classes are the blueprints/templates of php objects. Classes don't actually become objects until you do something called: instantiation.

When you instantiate a class, you create an instance of it, thus creating the object. In other words, instantiation is the process of creating an **instance** of an object in memory. What memory? The server's memory of course!

```
<?php include("class_lib.php"); ?>

</head>

<body>

      $stefan = new person();

</body>
</html>
```

Note: The variable $stefan becomes a handle/reference to our newly created person object. I call $stefan a 'handle', because we will use $stefan to control and use the person object.

If you run the PHP code now, you will not see anything displayed on your pages. The reason for this, is because we have not told PHP to do anything with the object we just created …

Step 9:

The 'new' keyword

To create an object out of a class, you need to use the 'new' keyword.

When creating/instantiating a class, you can optionally add brackets to the class name, as I did in the example below. To be clear, you can see in the code below how I can create multiple objects from the same class.

… From the PHP's engine point of view, each object is its' own entity. Does that make sense?

```
<?php include("class_lib.php"); ?>
```

```
</head>

<body>

	$stefan = new person();

	$jimmy = new person;

</body>
</html>
```

Note: When creating an object, be sure not to quote the class name. For example:

$stefan = new 'person';

… will get you an error.

Step 10:

Set an objects properties

Now that we've created/instantiated our two separate 'person' objects, we can set their properties using the methods (the setters) we created.

Please keep in mind that though both our person objects ($stefan and $nick) are based on the same 'person' class, as far as php is concerned, they are totally different objects.

```
<?php include("class_lib.php"); ?>

</head>

<body>

<?php
	$stefan = new person();

	$jimmy = new person;


	$stefan->set_name("Stefan Mischook");
	$jimmy->set_name("Nick Waddles");

?>

</body>
</html>
```

Step 11:

Accessing an object's data

Now we use the getter methods to access the data held in our objects … this is the same data we inserted into our objects using the setter methods.

When accessing methods and properties of a class, you use the arrow (->) operator.

```
<?php include("class_lib.php"); ?>

</head>

<body>

<?php
	$stefan = new person();

	$jimmy = new person;


	$stefan->set_name("Stefan Mischook");
	$jimmy->set_name("Nick Waddles");


	echo "Stefan's full name: " . $stefan->get_name();
	echo "Nick's full name: " . $jimmy->get_name();

?>

</body>
</html>
```

Note: The arrow operator (->) is not the same operator used with associative arrays: =>.

Congratulations, you've made it half way through the tutorial! Time to take a little break and have some tea … OK, maybe some beer.

In a short period of time, you've:

- Designed a PHP class.
- Generate/created a couple of objects based on your class.
- Inserted data into your objects.
- Retrieved data from your objects.

… Not bad for your first day on the OO PHP job.

If you haven't already, now is a great time to write out the code and watch it in action in your own PHP pages.

Step 12:

Directly accessing properties - don't do it!

You don't have to use methods to access objects properties; you can directly get to them using the arrow operator (->) and the name of the variable.

For example: with the property $name (in object $stefan,) you could get its' value like so:

$stefan->name.

Though doable, it is considered bad practice to do it because it can lead to trouble down the road. You should use getter methods instead - more on that later.

```
<?php include("class_lib.php"); ?>

</head>

<body>

<?php
	$stefan = new person();

	$jimmy = new person;


	$stefan->set_name("Stefan Mischook");
	$jimmy->set_name("Nick Waddles");

	// directly accessing properties in a class is a no-no.
	echo "Stefan's full name: " . $stefan->name;


?>

</body>
</html>
```

Step 13:

Constructors

All objects can have a special built-in method called a 'constructor'. Constructors allow you to initialise your object's properties (translation: give your properties values,) when you instantiate (create) an object.

Note: If you create a __contruct() function (it is your choice,) PHP will automatically call the __contruct() method/function when you create an object from your class.

The 'construct' method starts with two underscores (__) and the word 'construct'. You 'feed' the constructor method by providing a list of arguments (like a function) after the class name.

```
<?php

class person {

	var $name;


	function __construct($persons_name) {

	$this->name = $persons_name;

	}


	function set_name($new_name) {

		$this->name = $new_name;

	}

	function get_name() {

		return $this->name;

	}


}

?>
```

For the rest of this tutorial, I'm going to stop reminding you that:

- Functions = methods
- Variables = properties

… Since this is an OO PHP tutorial I will now use the OO terminology.

Step 14:

Create an object with a constructor

Now that we've created a constructor method, we can provide a value for the $name property when we create our person objects.

For example:

$stefan = new person("Stefan Mischook");

This saves us from having to call the set_name() method reducing the amount of code. Constructors are common and are used often in PHP, Java etc.

```
<?php include("class_lib.php"); ?>
```

```
</head>

<body>

<?php
	$stefan = new person("Stefan Mischook");


	echo "Stefan's full name: " . $stefan->get_name();


?>

</body>
</html>
```

This is just a tiny example of how the mechanisms built into OO PHP can save you time and reduce the amount of code you need to write. Less code means less bugs.

Step 15:

Restricting access to properties using ' access modifiers '

One of the fundamental principles in OOP is 'encapsulation'. The idea is that you create cleaner better code, if you restrict access to the data structures (properties) in your objects.

You restrict access to class properties using something called 'access modifiers'. There are 3 access modifiers:

1. public
2. private
3. protected

Public is the default modifier.

```
<?php

class person {


	var $name;

	public $height;
	protected $social_insurance;
	private $pinn_number;




	function __construct($persons_name) {

	$this->name = $persons_name;
```

```
	}

	function set_name($new_name) {
		$this->name = $new_name;
	}
	function get_name() {
		return $this->name;
	}

}
?>
```

Note: When you declare a property with the 'var' keyword, it is considered 'public'.

Step 16:

Restricting access to properties: part 2

When you declare a property as 'private', only the same class can access the property.

When a property is declared 'protected', only the same class and classes derived from that class can access the property - this has to do with inheritance …more on that later.

Properties declared as 'public' have no access restrictions, meaning anyone can access them.

To help you understand this (probably) foggy aspect of OOP, try out the following code and watch how PHP reacts. Tip: read the comments in the code for more information:

```
<?php include("class_lib.php"); ?>
</head>
<body>
<?php
	$stefan = new person("Stefan Mischook");
	echo "Stefan's full name: " . $stefan->get_name();
	/*
```

```
        Since $pinn_number was declared private, this line of code
       will generate an error. Try it out!
       */
       echo "Tell me private stuff: " . $stefan->$pinn_number;


?>

</body>
</html>
```

Step 17:

Restricting access to methods

Like properties, you can control access to methods using one of the three access modifiers:

1. public
2. protected
3. private

Why do we have access modifiers?

The reason for access modifiers comes down to control: it is makes sense to control how people use classes.

The reasons for access modifiers and other OO constructs, can get tricky to understand … since we are just beginners here. So give yourself a chance!

That said, I think we can summarize and say that many OOP constructs exist, with idea the many programmers may be working on a project …

```
<?php

class person {


       var $name;

       public $height;
       protected $social_insurance;
       private $pinn_number;



       function __construct($persons_name) {

       $this->name = $persons_name;

       }

       private function get_pinn_number() {

             return $this->$pinn_number;

       }
```

```
}

?>
```

Notes: Since the method get_pinn_number() is 'private', the only place you can use this method is in the same class - typically in another method. If you wanted to call/use this method directly in your PHP pages, you would need to declare it 'public'.

Nerd Note: Again, it is important (as we go along,) that you actually try the code yourself. It makes a HUGE difference!

Step 18:

Reusing code the OOP way: inheritance

Inheritance is a fundamental capability/construct in OOP where you can use one class, as the base/basis for another class … or many other classes.

Why do it?

Doing this allows you to efficiently reuse the code found in your base class.

Say, you wanted to create a new 'employee' class … since we can say that 'employee' is a type/kind of 'person', they will share common properties and methods.

… Making some sense?

In this type of situation, inheritance can make your code lighter … because you are reusing the same code in two different classes. But unlike 'old-school' PHP:

1. You only have to type the code out once.
2. The actual code being reused, can be reused in many (unlimited) classes but it is only typed out in one place … conceptually, this is sort-of like PHP includes().

Take a look at the sample code:

```
// 'extends' is the keyword that enables inheritance
class employee extends person {


      function __construct($employee_name) {


      }


}
```

Step19:

Reusing code with inheritance: part 2

Because the class 'employee' is based on the class 'person', 'employee' automatically has all the public and protected properties and methods of 'person'.

Nerd note: Nerds would say that 'employee' is a **type** of person.

The code:

```
// 'extends' is the keyword that enables inheritance
class employee extends person {


      function __construct($employee_name) {

            $this->set_name($employee_name);
      }



}
```

Notice how we are able to use set_name() in 'employee', even though we did not declare that method in the 'employee' class. That's because we already created set_name() in the class 'person'.

Nerd Note: the 'person' class is called (by nerds,) the 'base' class or the 'parent' class because it's the class that the 'employee' is **based** on. This class hierarchy can become important down the road when your projects become more complex.

Step 20:

Reusing code with inheritance: part 3

As you can see in the code snippet below, we can call get_name on our 'employee' object, courtesy of 'person'.

This is a classic example of how OOP can reduce the number of lines of code (don't have to write the same methods twice) while still keeping your code modular and much easier to maintain.

```
<title>OOP in PHP</title>

<?php include("class_lib.php"); ?>

</head>

<body>

<?php
// Using our PHP objects in our PHP pages.


$stefan = new person("Stefan Mischook");
```

```
echo "Stefan's full name: " . $stefan->get_name();

$james = new employee("Johnny Fingers");

echo "---> " . $james->get_name();

?>


</body>
</html>
```

Step 21

Overriding methods

Sometimes (when using inheritance,) you may need to change how a method works from the base class.

For example, let's say set_name() method in the 'employee' class, had to do something different than what it does in the 'person' class.

You 'override' the 'person' classes version of set_name(), by declaring the same method in 'employee'.

The code:

```
<?php
class person {

      // explicitly adding class properties are optional - but is
good practice

      var $name;

      function __construct($persons_name) {

            $this->name = $persons_name;
      }


      public function get_name() {

            return $this->name;

      }

      //protected methods and properties restrict access to those
elements.

      protected function set_name($new_name) {

            if (name != "Jimmy Two Guns") {
            $this->name = strtoupper($new_name);
            }

      }
```

```
}

// 'extends' is the keyword that enables inheritance

class employee extends person {


	protected function set_name($new_name) {

		if ($new_name == "Stefan Sucks") {
			$this->name = $new_name;
		}


	}

	function __construct($employee_name) {

		$this->set_name($employee_name);
	}



}



?>
```

Notice how set_name() is different in the 'employee' class from the version found in the parent class: 'person'.

Step 22:

Overriding methods: part 2

Sometimes you may need to access your base class's version of a method you overrode in the derived (sometimes called 'child') class.

In our example, we overrode the set_name() method in the 'employee' class. Now I've now used this code :

person::set_name($new_name);

… to access the parent class' (person) version of the set_name() method.

The code:

```
<?php
class person {

	// explicitly adding class properties are optional - but is
good practice
```

```
	var $name;

	function __construct($persons_name) {

		$this->name = $persons_name;
	}


	public function get_name() {

		return $this->name;

	}

	//protected methods and properties restrict access to those
elements.

	protected function set_name($new_name) {

		if (name != "Jimmy Two Guns") {
		$this->name = strtoupper($new_name);
		}

	}


}

// 'extends' is the keyword that enables inheritance

class employee extends person {


	protected function set_name($new_name) {

		if ($new_name == "Stefan Sucks") {
			$this->name = $new_name;
		}
		else if($new_name == "Johnny Fingers") {
			person::set_name($new_name);
		}

	}

	function __construct($employee_name) {

		$this->set_name($employee_name);
	}



}



?>
```

Step 23:

Overriding methods: part 3

Using :: allows you to specifically name the class where you want PHP to search for a method - 'person::set_name()' tells PHP to search for set_name() in the 'person' class.

There is also a shortcut if you just want refer to current class's parent: by using the 'parent' keyword.

The code:

```
<?php
class person {

	// explicitly adding class properties are optional - but is
good practice

	var $name;

	function __construct($persons_name) {

		$this->name = $persons_name;
	}


	public function get_name() {

		return $this->name;

	}

	//protected methods and properties restrict access to those
elements.

	protected function set_name($new_name) {

		if (name != "Jimmy Two Guns") {
		$this->name = strtoupper($new_name);
		}

	}


}

// 'extends' is the keyword that enables inheritance

class employee extends person {


	protected function set_name($new_name) {

		if ($new_name == "Stefan Sucks") {
			$this->name = $new_name;
		}
```

```
            else if($new_name == "Johnny Fingers") {
                  parent::set_name($new_name);
            }

      }

      function __construct($employee_name) {

            $this->set_name($employee_name);
      }


}


?>
```

Final Comments:

We've only touched on the basics of OO PHP. But you should have enough information to feel comfortable moving forward.

Remember that the best way to really have this stuff sink in, is by actually writing code for fun.

I would suggest creating say 10 simple objects that do simple things, and then use those objects in actual PHP pages. Once you've done that, you will feel very comfortable with objects.

Why learn OOP in PHP - another take.

For people new to OOP and are comfortable with 'classic' procedural php, you may be wondering why should you even bother to learn object oriented concepts … why go through the trouble?

The PHP world:

PHP is moving in an OOP direction. For example, many important PHP extensions like PEAR and Smarty are OO based. So, to really understand and use these frameworks properly, you need to understand object oriented PHP.

The functional/practical advantages:

For smaller projects, using object oriented PHP may be overkill. That said, object oriented PHP really begins to shine as the project becomes more complex, and when you have more than one person doing the programming.

For example:

If you find that you have say 10-20 or more functions and you find that some of the functions are doing similar things … it is time to consider packaging things up into objects and using OOP.

OOP and your career as a programmer:

OOP is the modern way of software development and all the major languages (Java, PERL, PHP, C#, Ruby) use this method of programming. As a software developer/programmer, it only makes sense (in terms of career,) to keep your skills up-to-date.

Besides making you a more valuable PHP coder, understanding OOP in PHP will give you knowledge (OOP knowledge,) that you will be able to take with you into other languages.

… When you learn OOP in PHP, you'll learn object oriented programming for any OO based language.

You will find with time that creating OOP based PHP projects, will just make your life as a programmer much easier. As an added bonus, soon you will develop your own collection of reusable objects, which you will be able to leverage in other projects.

Finally, You will also find that OOP based PHP is much easier to maintain and update.

OOP Challenges:

OO PHP does present some challenges when you first start out because you'll need to learn to think about your PHP projects in a different way: you will need to conceptualise the project in terms of objects.

More details …

One common way of starting an object-oriented project is to start by drawing up simple diagrams of your objects. As you begin to work with object diagrams, you will find that they help make developing OOP based PHP projects much easier.

Here are a few tips about drawing object-diagrams:

- Use a paper and pencil
- Draw boxes to represent each object
- In those boxes, list your methods and your properties
- Use arrows and lines between boxes to denote relationships (parent - child) between objects.

So if you have been sitting on the fence waiting to jump into OO PHP, now is as good time as any to get started.

Stefan Mischook

www.killerphp.com
www.killersites.com