Beginning Object Oriented Programming in PHP

by Timothy Boronczyk

2003-11-11

Synopsis

In this tutorial you will explore OOP in a way that'll start you on the fast track to polished OOP skills.

Introduction

Object-Oriented Programming tutorials are generally bogged down with programming theory and large, metaphysical words such as encapsulation, inheritance and abstraction. They attempt to explain things by comparing code samples to microwaves or automobiles which only serves to confuse the reader more.

But even when all the hype and mystique that surrounds Object-Oriented Programming (OOP) has been stripped away, you'll find it is indeed a good thing. I won't list reasons why; this isn't another cookie-cutter tutorial only serving to confuse you. Instead we'll explore OOP in a way that'll start you on the fast track to polished OOP skills. As you grow in confidence with your OOP skills and begin to use them more in your projects, you'll most likely form your own list of benefits.

The Magic Box (Objects)

Imagine a box. It can be any type of box you want... a small jewelery box, a large crate, wooden, plastic, tall and thin, Short and wide...

Next, imagine yourself placing something inside the box... a rock, a million dollars, your younger sibling...

Now wouldn't it be convenient if we could walk up to the box and just ask it to tell us what's inside instead of opening it ourselves? Actually, we can!

```
<?php

  $mybox = new Box("Jack");
  echo $mybox->get_whats_inside();

?>
```

Here the variable $mybox represents our self-aware box, which is also known as an object, which will be built by new--the world's smallest engineering and construction team! We also want to place Jack inside the box when it is built. When we want to ask the box it's contents, we'll apply a special function to $mybox, get_whats_inside().

But the code won't run quite yet, though; we haven't supplied new with the directions for him and his team to construct our box and PHP doesn't know what the function get_whats_inside() is supposed to do.

Classes

A class is technically defined as a representation of an abstract data type. In laymen's term, a class is a blueprint from which new will construct our box. It's made up of variables and functions which allow our box to be self-aware. With the blueprint, new can now builds our box exactly to our specifications.

```
<?php

  class Box
  {
    var $contents;

    function Box($contents) {
      $this->contents = $contents;
    }

    function get_whats_inside() {
      return $this->contents;
    }
  }

  $mybox = new Box("Jack");
  echo $mybox->get_whats_inside();

?>
```

Let's take a closer look at our blueprint: it contains the variable $contents which is used to remember the contents of the box. It also contains two functions, Box() and get_whats_inside().

When the box springs into existence, PHP will look for and execute the function with the same name as the class. That's why our first function has the same name as the class itself. And if we look closer still, we'll notice the whole purpose of the Box() function is to initialize the contents of the box.

$this is used to tell Box() that contents is a varable that belongs to the whole class, not the function itself. $contents is a variable which only exists within the scope of the function Box(). $this->contents is a variable which was defined by as part of the overall class.

The function get_whats_inside() returns the value stored in the class' contents variable, $this->contents.

When the entire script is executed, the class Box() is defined, new constructs a box and passes "Jack" to it's initialization function. The initialization function, which has the same name as the class itself, accepts the value passed to it and stores it within the class's variable so that it's accessable to functions throughout the entire class.

Now we've got a nice, brand new Jack in the Box (yes, I've been waiting the entire tutorial to say that).

Methods

To ask the box what it contains, the special get_whats_inside() function was used. The functions defined in the class are known as methods; they act as a method for communicating with and manipulating the data within the box.

The nice thing about methods is that they allow us to separate all the class coding from our actual script.

```
<?php

  include("class.Box.php");

  $mybox = new Box("Jack");
  echo $mybox->get_whats_inside();

?>
```

We could save all of the class code in a separate file (in this case I've named the file class.Box.php) and then use include() to import it into our script. Our scripts become more streamlined and, because we used descriptive names for our methods, anyone else reading our code can easily see our train-of-thought.

Another benefit of methods is that it provides our box, or whatever other objects we may build, with a standard interface that anyone can use. We can share our classes with other programmers or even import them into our other scripts where the functionality they provide is needed.

```
<?php

  include("class.Box.php");

  $mybox = new Box("Suggestion");
  echo $mybox->get_whats_inside();

?>


<?php

  include("class.Box.php");

  $mybox = new Box("Shoes");
  echo $mybox->get_whats_inside();

?>
```

Extends

A smart box is a wonderful object to have, but so far it's only capable of telling us what its content is. We don't have create an entirely new class to add new functionality... instead, we can build a small extention class based on the original.

```
<?php

  include("class.Box.php");

  class ShoeBox extends Box
  {
```

```
    var $size;

    function ShoeBox($contents, $size) {
      $this->contents = $contents;
      $this->size = $size;
    }

    function get_shoe_size() {
      return $this->size;
    }
  }

  $mybox = new ShoeBox("Shoes", 10);
  echo $mybox->get_whats_inside();
  echo $mybox->get_shoe_size();

?>
```

With the extends keyword, our script now has a ShoeBox class based on our original Box class. ShoeBox has the same functionality as Box class, but with extra functions specific to a special kind of box.

The ability to write such modular addons to your code gives great flexability in testing out new functions and saves time by reusing the same core code.

```
<?php

  include("class.Box.php");
  include("extentions.Shoe.php");
  include("extentions.Suggestion.php");
  include("extentions.Cardboard.php");

  $mybox = new ShoeBox("Shoes", 10);
  $mySuggestion = new SuggestionBox("Complaints");
  $myCardboard = new CardboardBox('', "corrugated", "18in",
"12in", "10in");

?>
```

Conclusion

You can see now how Object Oriented Programming got it's name--by focusing on building programs as a set of self-aware or smart objects.

The ability to design modular code which OOP practices afford will help you save time, reduce stress and easily share your work with others. It isn't necessarily difficult, but rather it's the technical and philosophical jargon associated with it that can cause confusion for beginners. But, with perseverance and practice, your understanding of will grow... as so will your confidence!

About the Author

Timothy Boronczyk lives in Syracuse, NY, where he works as an E-Services Coordinator for a local credit union. He has a background in elementary education, over 5 years experience in web design and has written tutorials on web design, PHP, Ruby, XML and various other topics. His hobbies include photography and composing music.