

/PHP/object-oriented programming



Paul Hudson takes the blood, sweat and tears out of mastering object-oriented programming, a powerful technique that's otherwise known as OOP

Knowledge needed PHP

Requires PHP

Project time 30 minutes

Mention object-oriented programming to most web developers and you usually get the same effect: blood will drain from their faces and their eyes will glaze over. That's a shame, because OOP is a hugely useful technique in anyone's toolbox. The reason people are usually afraid of it is because OOP has long been the realm of "serious programmers", as opposed to "dabblers". Given that many web developers see PHP as a neat hack to add functionality around the beautiful interface they've designed, you can imagine why OOP gives them jitters.

In this tutorial, I want to show you how object orientation works with the minimum amount of theory and the maximum amount of practical, hands-on code that you can get using straight away. So let's go OOP!

Understanding objects

An object is a bundle of data and functions that you get to define. From a web developer's point of view, the most common object type will be probably be a **Page**, which has some content and maybe a standard header and footer if your site has a uniform design.

First, you're going to build a **Page** object that makes it easy to create new pages on your site. This first object is going to hold one variable (the title), as well as two methods. A method is the fancy name for a function when it's inside an object. Create a file, call it "classes.php" and give it this content:

```
<?php
class Page {
    public $Title;
    public function PrintHeader() {
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<title>MySite</title>
</head>
<body>
<?php
    }
    public function PrintFooter() {
?>
</body>
</html>
<?php
    }
}
?>
```

There are three surprising things in that code. The first is the **class** keyword, which tells PHP you're about to define an object type. The difference between a class and an object is akin to the difference between the blueprint for a car and a car that's parked in front of you – the first is a definition and only exists once, whereas the second is an implementation of that definition, and can be created thousands of times. So, what you're defining here is a **Page** class: what pages should look like. You're not creating a page, just defining it.

The second surprise is the use of the **public** keyword. Ignore this for now – it just means "make this variable or method accessible to everyone who wants it". The third thing is how ugly PHP looks when you jump in and out of PHP mode, but there's not much you can do about that!

So, that's the class definition, but it doesn't do anything with the class. Create a file and call it "index.php". This needs to bring in the code from 'classes.php' so that it understands what a **Page** object means, then it needs to create one to represent itself and output it. Creating an object is easy:

```
<?php
include "classes.php";
$page = new Page();
$page->PrintHeader();
?>
<p>Welcome to my site!</p>
<?php
    $page->PrintFooter();
?>
```

Hopefully you can see now that the code creates an instance of the **Page** class, because it means "create me a new Page". Once that **Page** object is created,

Resources Find out more online



Tutorialized

As PHP is increasingly seen as a serious programming language, the number of PHP object-oriented programming tutorials out there is rapidly increasing. The Tutorialized site has a great selection to help you get started, so head here first.

www.tutorialized.com/tutorials/PHP/OOP/1



Practical PHP Programming

I've only touched briefly on access control modifiers in this tutorial (public, private and so on – see "Private vs public" at the top of page 75), but you'll find more information on these and other options in my online PHP book.

hudzilla.org/phpwiki/index.php?title=Access_control_modifiers



All about XOOPS To check out the power of OOP, pay a visit to xoops.org and discover how you can use this technology to create a powerful CMS – for free!

we can call its methods using the `->` symbol. Don't worry too much about how it works for now – just go ahead and run it, and you should see the header, the page text (“Welcome to my site!”) and the footer, all in one. Magic!

So why bother?

It's usually around now that people start to wonder what the point of OOP actually is, because all the above could be done without classes. Well, OOP makes life easier. The main advantage is encapsulation, which means that an object method should always operate in an opaque manner. Imagine a coffee machine: if you ask for a steaming cup of Java, you expect a cup of coffee. If your company updates the coffee machine, it doesn't matter that it has a new grinding system: you expect the same input to give you same output.

This is how OOP should work. When you call `PrintHeader()`, you expect your page to get the standard HTML header for your site. If you later go back and change `PrintHeader()` so that it adds a bit more functionality, its actual purpose should never change. You can take your classes and give them to someone else to use, and when you make updates, they can just drop in your new code and their own code won't be affected.

Encapsulation is why objects have variables of their own, such as `$Title` in the `Page` class. If you were to create 10 `Page` objects, each one would have its own `$Title` independent from the rest. This is obviously more important for things where you're likely to have multiple active objects in the same page, such as if you had a `Message` object for forum messages – each one would have its own `Title`, `Author`, `Date` and `Content` variables.

Whenever you call a method of an object, it automatically uses the variables from that object. To illustrate this, I want to introduce you to the concept of a “constructor”: a special method that gets run when an object is created. For a page, this needs to do everything you want on each page, such as starting a session and connecting to a database. Constructor methods can also accept parameters, which is a great way to make page titles work. To make a constructor, define a method in your class with the name `__construct()` (that's two underscores), and do what you want in there. For example:

```
public function __construct($title) {
    mysql_connect("localhost", "net", "net");
    mysql_select_db("dotnet");
    $this->Title = $title;
}
```

The `Page` constructor now accepts a variable called `$title`, which gets used to set the local `$Title` variable. It also connects to the database so that the rest of the page can make SQL queries as needed. Once `$Title` is set, you can start using it wherever you want inside the class, such as amending the `PrintHeader()` method so that the `<title>` tag contains the title, as follows:

```
<title>MySite - <?php echo $this->Title; ?></title>
```

There are two important things here: `$this` and `->`. The `$this` variable is a special, reserved variable name, used to refer to the current object (and

Private vs public

Control access to your variables and methods

In the example code for this tutorial, all the variables and methods are public, which basically means that anyone can access them. An alternative is, rather predictably, private, which means that only code from inside the class can access the variable or method. When working with larger codebases, using public is recommended for all internal functions, because it means you can control how your data is accessed. For example, if setting the `$Title` property should automatically update a site cache, you definitely don't want people to write to `$Title` directly. Instead, make `$Title` private, then create a public function called `SetTitle()` that changes `$Title` and updates the cache for you.

no others of the same class). If you don't use `$this`, PHP will look for a local method variable called `$Title`, rather than the object's `$Title` variable. The second potential hiccup is the use of `Title` rather than `$Title`. A lot of people try writing `$this->$Title`, meaning “refer to the `$Title` variable of the `$this` object”. But PHP doesn't work like that. Each variable in PHP should have precisely one dollar sigil. If you use two, as with `$this->$Title`, what PHP does is look up the value of `$Title` and treat that as the variable to read.

The final change to make constructors work is to pass in the variable in ‘index.php’, so change the `new Page()` line to this:

```
$page->PrintHeader("Welcome!");
```

Loading classes the easy way

One of the key features of OOP is the ability to define a class inheritance, which is where you say, “OK, I've defined `Page`, but now I want to define a `UserProfilePage`, which is just like a normal `Page` except that I want to add a few bits.” OOP lets you do just that, and more importantly, it lets you do it without duplicating all the code from `Page` into `UserProfilePage`.

But before I show you how to inherit one class from another, I want to introduce you to a neat trick: class autoloading. Basically, when PHP tries to create an object, it looks to see whether the required class has been defined already. If it hasn't, you can tell it to call a function you defined, then try again. Essentially, you get a second attempt to get the class installed. This is all done very quickly so that there's basically no overhead. When you have just one class you generally don't have to bother with class autoloading, because you can store your class in one file and only include that (as with our ‘classes.php’ file). However, when you have several classes, it's good programming form to split them into individual files so they are easier to maintain, easier to read, and less likely to step on each other's toes. But splitting them up also makes autoloading work, which is the real win here!

First, rename your ‘classes.php’ file to something else, such as ‘stdlib.php’. This will contain any essential non-OOP functions your site needs, but it won't contain any classes. Move the `Page` class into its own file, calling it “Page.php” (you need the capital P). In place of the `Page` class in ‘stdlib.php’, add this:

```
function __autoload($class) {
    include "$class.php";
}
```

The double underscore at the beginning should give away the fact that this is a “magic” PHP function – something PHP calls for you at the right time, as opposed to something you call yourself. This particular function gets called whenever PHP tries to create an object of a class that isn't defined, and it gets passed the name of the missing class. So, if you try and create a `Page` class, PHP will say “Page? I have no record of such a class!”, then call >>

Encapsulation means that an object method should always operate in an opaque manner



Get the point If you want to learn more about OOP, then Sitepoint is a great resource. Visit www.sitepoint.com/article/php-paging-result-sets for more information

>> `__autoload()`, passing `Page` as its only parameter. Now you can see why `Page` is in its own file called 'Page.php': that `__autoload()` function will automatically include the PHP file for the missing class, meaning that when PHP tries to load the class a second time (which it will do as soon as `__autoload()` is finished), it will find and load `Page` correctly.

Now we're all set to get stuck into some good, old-fashioned OOP inheritance. If `Page` is a normal page on the site, then a `UserProfilePage` ought to have some special formatting and functionality, specific to its needs. With class autoloading in place, all you have to do is create a new PHP file, name it "UserProfilePage.php", and you can start using that class immediately! Of course, you'll want to make maximum reuse of code, which means building the `UserProfilePage` on top of the standard `Page` class. That can be done with two new PHP keywords: `extends` and `parent`. The first one means "this class builds on another one – use all the variables and methods from the other class, as well as these new ones", and the second means "refer to the class I inherited from". Here's how it looks in PHP:

```
<?php
class UserProfilePage extends Page {
    public $User;
    public function __construct($user) {
        parent::__construct("User profile for $user");
        $this->User = $user;
    }
    public function PrintHeader() {
        parent::PrintHeader();
        echo "<h2>$this->User's profile</h2>";
    }
}
?>
```

So, `UserProfilePage` builds on `Page`, but adds a new variable: `$User`. You'll notice, though, that both `__construct()` and `PrintHeader()` use the same names as the `Page` class, so what happens? Well, PHP always calls the method from the first class it finds that implements it. For example, if you create an object of type `UserProfilePage`, PHP will call the `UserProfilePage` version of `PrintHeader()`, which is referred to as `UserProfilePage::PrintHeader()` to distinguish it from `Page::PrintHeader()`.

If `UserProfilePage` doesn't implement a method, such as `PrintFooter()`, PHP looks to the class it inherits from to see if that implements it. Here, the method `Page::PrintFooter()` exists, so it gets called, but if you have a chain of class inheritance (for example, `UserProfilePage` inherits from `ProfilePage`, which inherits from `DynamicPage`, which inherits from `Page`), PHP will keep looking at parent classes until it finds the first one to implement `PrintFooter()`.

In the code example above, `UserProfilePage` overrides the `__construct()` and `PrintHeader()` methods provided by the `Page` class, but actually I want to use them anyway, because `Page::__construct()` gives me database access and `Page::PrintHeader()` gives me the standard template. This is where the `parent` keyword comes in. Saying `parent::PrintHeader()` means "call the next

Magic functions

Go further with objects and functions

This tutorial shows you the `__construct()` and `__autoload()` magic functions, which have double underscores before their names. There are quite a few others available, particularly for objects: `__call()`, `__get()` and `__set()` are the most popular, but there's also `__toString()`. The first one gets called if PHP tries to call a method that doesn't exist – giving you the chance to pull in the method, just as with `__autoload()` – and the second two get called when a variable that doesn't exist is read from or written to. The `__toString()` function gets called whenever your object is read as a string, such as `echo $someobject;`, and gives you the chance to return some text more meaningful than `object`. So, in a PHP game with a `Player` class, you might have `Player::__toString()` return the player's name.

`PrintHeader()` method up the inheritance chain, then continue in my method" – essentially reusing all the code written in 'Page.php' without having to copy and paste at all. You'll notice, though, that `UserProfilePage` accepts a username as the parameter to its constructor, but then passes a fixed page title up to the `Page` class, which shows just how flexible classes can be.

Abstract art

The next step is to create "abstract" classes, which are classes that can't be created. Consider this: is there such thing as a dog without a breed? No. You get pedigrees and mixed breeds, but the dog still has a breed. Equally, all dogs share characteristics: they run around, bark, and so on, so having individual `Bark()` methods for every breed would just be silly. The solution? Create an abstract `Dog` class with all the basic doggy functionality, without being able to be created. You have to create a new class that inherits from `Dog`, such as `Labrador`, and only change what's specific to that breed.

To make a class abstract, you put the word "abstract" before the `class` keyword. If, for instance, you never wanted people to create a raw `Page` object, and instead they had to create specific types of page, you'd put `abstract class Page` into 'Page.php'. You can also make individual functions abstract, but an abstract function cannot contain code. For example:

```
abstract public function PrintFooter();
```

That method will get inherited by classes that build on the `Page` class, which means that your `UserProfilePage` will now contain an abstract method. But it turns out that having even one abstract method in a class makes the whole class abstract, leaving you with two options: make all of `UserProfilePage` abstract (uncreateable!), or implement `PrintFooter()`. It's this last option that abstract methods are really good for: they enable you to say, "OK, I've defined a basic structure of a class, but if you want to inherit from it you absolutely must create a `PrintFooter()` method, because I depend on its existence."

Abstract methods have an important part to play in OOP. Remember, the goal of classes is that they ought to be opaque, and using abstract methods ensures that people understand precisely what's required of them, without you having to jump through all kinds of hoops. ●

All dogs bark, so having individual `Bark()` methods for every breed would just be silly



About the author

Name | Paul Hudson
Site | www.hudzilla.org
Areas of expertise | PHP, MySQL, Linux
Published | *PHP in a Nutshell*, *Fedora Core 5 Unleashed*, *Ubuntu Unleashed*
Favourite holiday food | My mum's lemon meringue pie