

PHP VERSION 5

Practical PHP

PHP 5 heralds a new era for PHP programmers, or so the developers would have us believe. Paul Hudson separates the outstanding features from the outrageous hype and jumps on the bandwagon...

Happy New Year, PHPers – it's January at last, and with the new year we also get a major new release of our favourite programming language. Having been in development for much of 2002 and all of 2003, PHP 5 brings with it a healthy swathe of new features, bug fixes, enhancements, and optimisations that, if carefully harnessed, can and will improve your scripts. The purpose of this article is to kick off a tour of PHP 5, examining what's changed and why, as well as providing a comprehensive introduction to its biggest new features. Before we get started, though, I want to supplement *LXF46*'s exclusive from PHP developer Zeev Suraski by briefly outlining exactly how we got to the current situation...

Personal home pages

PHP 3 was the first version to achieve widespread use on the Net, largely because it finally had a reliable script parser – before that, any errors reported in your scripts were as likely to be your fault as it was the developers'! Most of the advantages to PHP 3 are still the key advantages of PHP 5 today: quick and easy script programming, easy database access, and a wide range of built-in functions to let you focus on the hard stuff yourself.

The big changes in PHP 4's lifespan were based around performance and security – the execution paradigm for scripts

was changed from “execute each line as it's read” to “parse first, execute later”, which is a great deal faster. On the security front, the PHP 4 series also saw `register_globals` being disabled by default, which, while causing a lot of anguish from existing programmers, actually serves to make everyone's scripts safer in the long run.

PHP 4 also managed to add a variety of new functions and thousands (literally) of bug fixes, but the biggest difference felt was in speed and security.

What's new?

With PHP 5, the focus is on features, features, features, and quite rightly – we're not likely to see as big a change as PHP 3 to PHP 4 ever again, which has given the developers more time to focus on all the requests they've received for additions to the language. Syntactically speaking, PHP 5 is largely similar to PHP 4.3.4 and you should be able to import your scripts across wholesale.

If anything, the change that is most likely to catch you out is the big reworking of object orientation. Since this feature was introduced in PHP 3, it was really just a hack – you could define classes and instantiate objects, but many features of the OOP paradigm were missing, such as access control, destructors, abstract and final classes, and static class variables.

Of course, there are a variety of other features in PHP 5, but this issue we're covering the OOP changes alone, because, as I mentioned, they are most likely to break backwards compatibility.

In subsequent issues we'll be looking at the new XML module, *SimpleXML*, and also the new flat-file database back-end, *SQLite*.

Handling objects

The biggest difference in the PHP 5 object model is the way in which it handles object references. In PHP 4, in order to keep objects working like other variables, when you assigned one variable to another you performed a copy. That is:

```
$foo = "bar"
$wombat = $foo;
$foo = "baz";
```

In this situation, **\$wombat** would still be set to **bar** even after **\$foo** has been changed to **baz** because the variable data was copied in its entirety and not as a reference. In PHP 4, copying an object did just that – you had a unique and individual instance of that object that you could manipulate independent of the original. However, this led to quite a lot of confusion from most developers, and also left many scripts running a great deal slower than they needed to.

From PHP 5, objects are now referenced via a unique handle that serves as an identifier. An object variable, instead of holding the actual object, now holds the handle number of that object. When the object is used, the handle is resolved to the actual object, but when the object is copied it is the handle that is copied not the object. In PHP 4, this is the equivalent of always copying objects by reference, and means that you need to go to extra lengths if you want to actually copy the object data. More on that later – first let's look how the new reference-by-default action helps.

```
$foo = new myobject;
dobar($foo);
```

In that code, we pass the handle to **\$foo** into the **dobar()** function, which means that any changes made to **\$foo** inside

dobar() are made to the actual object itself. This solves a regular problem encountered by novice programmers whereby copies are passed in and copies are returned by default – a problem that usually confuses a great deal! By referencing objects by default, PHP eliminates a great deal of object copying – traditionally a very slow task – which will therefore help speed up the majority of scripts once any code-breaking upgrade problems are fixed.

If you explicitly want to copy an object rather than reference it, there's a magic function just for you. Magic functions, if you've yet to meet them, are PHP functions built into each object by default that start with two underscores. To copy an object, call the **__clone()** magic function, which returns a copy of the object. Here's an example:

```
$foo = new myobject;
$bar = $foo->__clone();
```

That will create a new instance of the object and return its handle. If you want to update your PHP 4 scripts to work smoothly in PHP 5, you'd use code like this:

```
$foo = new myobject;
dobar($foo->__clone());
```

The **__clone()** function is just the beginning, though – there are quite a few other magic functions introduced with PHP 5 that make things all the more interesting...

Ha ha this-a-way

Magic functions are PHP's way of providing some base functionality to all objects, and might seem a little confusing and unusual at first until you think that this sort of functionality has been around for quite some time in other languages – compare the all-mighty **TObject** in *Kylix*, for example, which includes a selection of basic functions by default.

There are six new magic functions introduced in PHP 5 and they serve a variety of purposes. I've ordered them here easiest first to get you started quickly...

First off the bat there's **__toString()**, a peculiar function that, if you define it, is automatically called by PHP if you attempt to use your object as if it were a string. If you're questioning why **__toString()** is important, there are a variety of possible uses – perhaps the most interesting is to make it print out a selection of information about the object, sort of like a filtered **var_dump()**.

A magic function for classes that is global as opposed to being attached to a class is **__autoload()**, which is another unusual but helpful function that, if you have defined it, is called whenever you try to instantiate an object of a class that doesn't exist. For example:

```
<?php
function __autoload($class) {
    echo "$class: No such class defined!\n";
}

class Dog() {
    function whatever() {
    }
}

$mrjingles = new mouse();
?>
```

Here we create a new mouse object despite only having a **Dog** class defined. The end result is that **__autoload()** is called, with the name of the non-existent class passed in as a parameter, and



OOP: WHY BOTHER?

Object Orientation *not* slow!

Despite the OOP paradigm having been commonly used since *Smalltalk*, many people still shun objects and consider OOP to be slow and unwieldy. It's true that, back in the early days, object orientation was pretty poor in C++ because compilers did a hashed job, but this has improved a long way. In PHP 3 and 4, OOP was also a pretty hashed job, but again it has been improved massively in PHP 5.

OOP is a great way to give a limited amount of intelligence to your code whilst also forming coding contracts with yourself and other programmers. It makes a lot more sense to say **Player->Draw()** and have the **Player** object know how to handle its own drawing than have to have your main **draw()** function know how to draw the **Player** object, simply because it means that if you ever find your player needs to be have some special drawing code that you hadn't anticipated earlier, you only need amend the **Player->Draw()** function – all the other code doesn't care what **Player->Draw()** does because it's all handled internally. Using this thinking allows you to pick up the **Player** class and drop it into other projects easily, which promotes code re-use.

The code contracts idea is also key, because OOP allows you to define what variables and functions can be accessed and where, which means that if you (or someone else) tries to do something that you previously decided shouldn't be allowable, they will be stopped dead in their tracks.

TUTORIAL PHP

◀◀ the code outputs a warning. While this is all well and good, what makes it *very cool* is the fact that, after calling `__autoload()`, PHP will attempt to instantiate the class again. In practice this means that you can attempt to create a class, and, using `__autoload()`, automatically **include()** the script containing the class definition if it's not included already, like this:

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

Using the code above, attempting to create an object of class **mouse** will fail, the PHP will attempt to include `mouse.php`, and finally attempt to instantiate the **mouse** object again.

Build it up

Handling the creation and deletion of objects in PHP has always been a hassle, with various hacks being using by PHP 4 programmers to attempt to simulate a destructor. In PHP 5, this is now fixed with a unified constructor/destructor system. For the uninitiated, a constructor function is called when an object is

created, and a destructor when deleted, and are helpful to initialise and clean up your objects.

Construction and destruction are handled using the `__construct()` and `__destruct()` magic functions respectively, which are called automatically as you work with the object. Take a look at the following script, for example:

```
<?php
class mouse {
    function __construct() {
        echo "This is a mouse\n";
    }

    function __destruct() {
        echo "This is an ex-mouse\n";
    }
}

echo "Stage 1\n";
$mightymouse = new mouse();
```

ACCESS CONTROL

Private, public and protected modifiers

So far we've covered about half of the new OOP features in PHP 5, and hopefully you can already appreciate that it's a worthy upgrade. One of the key advantages to OOP is that it lets you form code contracts, that is, it lets you specify precisely which parts of your scripts may access what variables inside your objects, which acts to re-enforce your design specification at the code level.

There are three types of access control: private, public, and protected, which respectively make variables and functions available only to the class, available to everyone, and available only to the class and classes inheriting from it.

The public modifier was implicitly used by PHP 4, which meant that all properties and functions defined in a class could be used from anywhere in your script. In PHP 5, this would be written like this:

```
<?php
class sheep {
    public function __construct() {
        echo "Baa!\n";
    }

    public function __destruct() {
        echo "Aab!\n";
    }

    public function do_cheese() {
        echo "Baa baa black sheep\n";
        echo "Have you any wool?\n";
    }
}

$dolly = new sheep();
$dolly->do_cheese();
?>
```

As seen in the other scripts shown so far, the public keyword isn't required, because PHP 5 continues to implicitly use public access if you don't specify otherwise. While

this does mean that scripts are compatible with both PHP 4 and PHP 5 simultaneously, it does leave that little bit extra to human error – I'd recommend you specify public explicitly, so as to make your intentions clear.

Now, try changing the `do_cheese()` function to this:

```
private function do_cheese() {
    echo "Baa baa black sheep\n";
    echo "Have you any wool?\n";
}
```

When you run the script this time you should get this error back:

```
Fatal error: Call to private method
sheep::do_cheese() from context "
```

Now that the function is private, we can't call it from elsewhere in the script, because private functions are only supposed to be called from inside the object itself. Similarly, we can declare variables as private, like this:

```
<?php
class koala {
    private $Name;

    public function Name() {
        return $this->Name;
    }

    public function setName($NewName) {
        $this->Name = $NewName;
        return true;
    }
}

$fuzzy = new koala;
$fuzzy->setName("Fuzzy");
echo "{$fuzzy->Name()}\n";
?>
```

Here we have a classic setup in the OOP paradigm – a private variable with public accessor functions that are used to set and get the variable.

The difference between private and protected variables is important to understand, as both have their uses when it comes to contract programming. A privately declared variable is accessible only to the class in which it is declared, which is very useful if you only want the functions of that object to have access to those variables – this stops inherited classes playing around with the parent class. On the other hand, protected variables are available to use from the child classes, which allows much more flexibility at the possible expense of code reliability.

At the time of writing, PHP has a bit of an irregularity in its OOP system. Take a look at this script:

```
<?php
class dog {
    private $Name;
}

class labrador extends dog { }

$dozer = new labrador;
$dozer->Name = "Dozer";
print_r($dozer);
?>
```

That will output this:

```
labrador Object
(
    [Name:private] =>
    [Name] => Dozer
)
```

General consensus is that this is part of the plan because PHP does generally let you define variables on the fly, but it does make things confusing – PHP sets a variable called `$Name` to the value you pass, and many might think that this is it overriding the private declaration. In actual fact it ends up with two `Name` variables in the same object, which is almost certainly not a desired outcome. Be careful!

```
echo "Stage 2\n";
unset($mightymouse);
echo "Stage 3\n";
$dangermouse = new mouse();
echo "Stage 4\n";
?>
```

(Note: No mice were hurt in the production of this tutorial!)

That code will output the following text:

```
Stage 1
This is a mouse
Stage 2
This is an ex-mouse
Stage 3
This is a mouse
Stage 4
This is an ex-mouse
```

There are two things to note there. Firstly, calling **unset()** on an object calls its destructor implicitly as the object is deleted, as you can see at **Stage 2**. Secondly, as PHP performs garbage collection at the end of the script, it will automatically destroy the **\$dangermouse** object and call its destructor in the process, hence the ex-mouse notice at **Stage 4**.

Chain constructions

Using **__construct()** on a simple object is fairly straightforward, as you've seen. However, when you work with classes that inherit from other classes, things get a little more complicated. Consider the following class hierarchy:

Animal



Dog



Labrador

The grandfather class, **animal**, will have various properties that are generic to all classes beneath itself, such as age, height, weight, *etc.* The parent class, **dog**, will have properties generic to all breeds of dog, such as `number_of_lampposts_sniffed`, `tail_wag_level`, *etc.*

Finally, the child class, **labrador**, will have variables unique to the labrador breed such as `number_of_andrex_adverts_starred_in`, *etc.* Yes, this is a little contrived, but you should see the point! For a labrador dog (we'll call him **Oscar**) to exist fully, he needs all the variables from the child class, the parent class, and the grandparent class to exist and be initialised properly.

So, how do we go about doing this? Take a look at this chunk of code, which is the PHP representation of above class hierarchy:

```
<?php
class animal {
    function __construct() {
        echo "Animal created\n";
    }

    function __destruct() {
        echo "Animal deleted\n";
    }
}

class dog extends animal {
    function __construct() {
        echo "Dog created\n";
    }
}
```

```
function __destruct() {
    echo "Dog deleted\n";
}

class labrador extends dog {
    function __construct() {
        echo "Labrador created\n";
    }

    function __destruct() {
        echo "Labrador deleted\n";
    }
}

$oscar = new labrador;
?>
```

Save that script and run it using the CLI SAPI – you should see messages about a labrador being created and deleted. However, what you won't see is any message about a dog or an animal being created and deleted, despite Oscar clearly being both. The reason for this is because PHP will call one and only one constructor, which is **labrador::__construct()** – it won't call **dog::__construct()** or **animal::__construct()**, which means if you have anything more substantial in those two functions that just echoing out a minor notice, your code will break.

How to get around this? Well, inside the object you have direct access to the parent class through the object **parent**, which means you can statically call the **__construct()** function of the parent class from the constructor of the child class. Sound confusing? It's not, take a look at this:

```
class labrador {
    function __construct() {
        parent::__construct();
        echo "Labrador created\n";
    }

    function __destruct() {
        echo "Labrador deleted\n";
        parent::__destruct();
    }
}
```

The first action that **labrador::__construct()** does is to call the **__construct()** function of its parent class, **dog**. This runs **dog::__construct()** for the **labrador** object, setting up an variables defined at that level (`number_of_lampposts_sniffed`, *etc.*), then goes ahead and runs **labrador**-specific stuff. It's crucial that you call the **parent** constructor before going ahead and running **child** tasks because it's often the case that the **child** class will use properties of the **parent**, and trying to use things before they have been setup is, of course, bad.

There are two further things to note in this area. Firstly, you'll need to add **parent::__destruct()** and **parent::__construct()** to the dog class also, so that it will in turn call **animal::__construct()** and **animal::__destruct()** – this completes the chain. Secondly, note how **parent::__destruct()** comes *after* the **child** code. This is for similar reasons to why **parent::__construct()** is before the **child** code – if you destroy variables created by the **parent** before you're certain you're finished using them, there's a chance you might run into trouble. Good programming practice is to always put **parent::__destruct()** after the **child**-specific code. [LXF](#)

NEXT MONTH

There are three magic functions we've not yet looked at, which are **__get()**, **__set()**, and **__call()**, as well as static class variables and abstract/final classes – we'll be looking at these next month, as well as the *SimpleXML* extension.