More PHP with PHP 5

Derick Rethans

Table of Contents

Introduction

Since the introduction of PHP 4 in 1999 a lot of feature requests have been made, most notably for the OO support of PHP. With PHP 5 these wishes finally seem to be realized in the form of a revamped Object Oriented model. The new model handles objects itself in a more natural way, and there are a number of additions, such as support for overloading, access modifiers, namespaces and exceptions. All of these will be explained in the text below.

Object Model Changes

One of the most important changes is the way objects are handled. In Zend Engine 1 (the engine that powers PHP 4.x) objects are always copied by-value, which duplicates the object. You could work around it by specifically using references everywhere, but that got nasty pretty fast.

With Zend Engine 2 (the engine version in PHP 5) objects are always passed by reference (actually, the handle to the object is passed, not the object itself) and thus there is more inituitive object handling. Let's demonstrate this with an example; the first shows how objects are passed around with Zend Engine 1.

Example 1. Passing objects to functions with PHP 4

```
<?php
    class example1 {
        var $name;
    }

    function change_name($object, $name) {
        $object->name = $name;
    }

    $obj = new example1();
    $obj->name = "PHP 4";
    change_name($obj, "PHP 5");
    echo $obj->name."\n";
?>
```

The example above will NOT print "PHP 5", but instead it will echo "PHP 4". This is because the object $obj will be copied when it is passed to the function change_name(). In order to 'fix' this, you will need to mark the object parameter to the function as passing-by-reference with the '&' sign. The function declaration then looks like this:

```
<?php
    function change_name(&$object, $name) {
        $object->name = $name;
    }
?>
```

With PHP 5 and Zend Engine 2 this will no longer be needed. The first listing will work as expected and display "PHP 5". Although this is one of the most important changes, the object model in PHP 5 has tons of other new features; a sample follows.

Construct/Destruct

Zend Engine 2 introduces a new way to specify contructors, and also implements destructors. Where in Zend Engine 1 the constructor of the class was a method with the same name as that class, in Zend Engine 2 it becomes a special method (note the '__' prefix): `__construct` for the constructor, and `__destruct` for the destructor.

You might wonder what the reason for this change is, but it's pretty obvious actually. If you want to rename your class, for instance, you don't have to find your constructor and change it's name. Another example of its usefulness becomes apparent when you are extending a parent class. You can always call `parent::__construct` to execute the parent's constructor in your inherited class.

Implementing the destructor was made possible because objects are now manipulated through handles, and it is easy to count how many references there are to an object - unlike in PHP 4 where it was virtually impossible to track how often an object was referenced because there were usually so much copies around. See the next example on constructors and destructors.

Example 2. Constructors and Destructors

```
<?php
    class car {
        public $name;

        function __construct($name) {
            $this->name = $name;
            echo "constructor ran!\n";
        }

        function __destruct() {
            echo "destructor ran, name = {$this->name}!\n";
        }
    }

    $obj = new car('Clio');
    $obj->name = 'Twingo';
    unset($obj);
    echo "end!\n";
?>
```

Which outputs:

```
constructor ran!
destructor ran, name = Twingo!
end!
```

In order for older PHP 4 based scripts to work, the new Zend Engine will still honor a constructor with the same name as its class. If both this method and `__construct` are present PHP 5 will use the latter as it's constructor.

PPP

The three P's represent the three different protection levels of methods and properties: Public, Protected and Private. These three modifiers can be used instead of the PHP 4 modifier `var` for properties, or they can be prepended to `function` in method declarations.

`public` is equivalent to using `var` for properties, and is the default access level for methods. When something is marked `public` it can be accessed by any other class or function:

Example 3. The dirty way

```
<?php
    class engine {
        function giveGas () {
            echo "vrooom!!\n";
        }
    }

    class car extends engine {
        public $wheels;

        function drive() {
            $this->giveGas();
        }

        function setWheels($count) {
            if ($count == 3 || $count == 4) {
                $this->wheels = $count;
            }
        }
    }

    $obj = new car();
    $obj->wheels = 3;
    $obj->giveGas();
?>
```

Which outputs:

```
vroom!!
```

Of course this is not what you want! For a start, you don't usually want to instruct the car to 'giveGas', but rather the engine. This means you need to disallow direct execute access to the `giveGas` method, but allow it when it is called from the extended class (through the `drive` method). This is what the `protected` is for. If we modify the definition of the `giveGas` method to:

```
        protected function giveGas () {
            echo "vrooom!!\n";
        }
```

then you cannot use `$obj->giveGas()` anymore, but must call the method through the `drive` method of the `car` class.

However, this is still not enough, as you can still set the number of wheels to something other than 3 or 4 by modifying the property instead of using the `setWheels` method which ensures with a check that `car` can only have 3 or 4 wheels. This is where `private` is useful. By marking the `wheels` property as `private`, the only modification that will be accepted comes through calling the `setWheels` method. Here is the new updated code, with added modifiers:

Example 4. public, protected and private modifiers

```
<?php
    class engine {
        protected function giveGas () {
            echo "vrooom!!\n";
        }
    }

    class car extends engine {
        private $wheels;

        public function drive() {
            $this->giveGas();
        }

        public function setWheels($count) {
            if ($count == 3 || $count == 4) {
                $this->wheels = $count;
            }
        }
    }

    $obj = new car();
    $obj->setWheels(3);
    $obj->drive();
?>
```

Setters and Getters

When you have some sort of generic class that stores configuration data, you usually want to control which data can be set and don't want to allow the 'users' of your class to write directly to the properties. PHP 5 allows you to define `__set` and `__get` functions which are called for properties that you access, but which don't exist in the class definition.

The following example shows how to use 'setters' and 'getters' to control the access to a specified set of 'properties' which are stored in the `config` property of your class (which is of course a private property).

Example 5. Overloaded properties

```
<?php
    class config {
        protected $data;

        function __get($name) {
            return isset($this->data[$name]) ? $this->data[$name] : 'not set';
        }

        function __set($name, $value) {
            if (in_array($name, array('path', 'base_dir', 'admin_dir'))) {
                $this->data[$name] = $value;
                return TRUE;
            } else {
                return FALSE;
            }
        }
    }

    $cfg = new config();
    $cfg->path = "/home/httpd/html";
    $cfg->base_dir = "/foo";
    $cfg->image_dir = "/images";
```

```
    echo $cfg->path. $cfg->base_dir. "\n";
    echo $cfg->admin_dir. "\n";
?>
```

Which outputs:

```
/home/httpd/html/foo
not set
```

Clone

In PHP4, if you wanted to pass objects by value to a function you didn't need to do anything as it was done by default. Because this default behaviour changed in PHP 5 to pass-by-reference, you might need to change your code if you were counting on your objects being copied if you passed them to a function. There are two ways to get the old behaviour back. The first is to make the setting `zend2.implicit_clone = 1`; the other is to clone your object yourself. If your class provides a special `__clone` method, that will be called rather than the default clone method which simply copies the object's properties to the clone. If you specify your own method, the properties of the orginal object can be accessed with the `$that` pseudo-object, and you can store the new data in the cloned `$this` object.

In the example below we specify our own `__clone` method to set the properties of the cloned object.

Example 6. __clone

```
<?php
    class counter {
        public $id; /* Not unique */
        public $windows;
        public $doors;

        function __clone() {
            $this->doors = $that->doors;
            $this->windows = $that->windows;
            $this->id = $that->id + 1;
        }
    }

    $o = new counter();
    $o->id = 1;
    $o2 = $o->__clone();
    echo $o2->id. "\n";
?>
```

Which outputs:

```
2
```

Dereferencing

If in PHP 4 a method call returned an object, and you wanted to call a method on the returned object, you had to do so in two steps:

```
<?php
    $obj = $foo->bar();
    echo $obj->bar_too();
?>
```

With PHP 5 you don't need to do this anymore, as you can directly use the returned object to call a method from:

```
<?php
    echo $foo->bar()->bar_too();
?>
```

In addition to this, the dereferencing will work on overloaded objects, for example Java objects, or COM components.

Exceptions

Exceptions are usually seen as a vital part of an OO model. PHP 5 features a base exception class "Exception", which implements some basic functions.

The constructor itself accepts up to two parameters. The first is the *`message`*, the second the error*`code`*. The base exception class implements four functions: `getMessage`, which returns the message if it was passed to the contructor; `getCode`, which returns the errorcode if it was passed; `getLine`, which returns the line number on which the exception was thrown; and `getFile` for the filename on which the exception was thrown.

Here is an example of how to throw and catch exceptions:

Example 7. Built-in Exception

```
<?php
   try {
       throw new Exception('Something went wrong!', 9);
   } catch (Exception $e) {
       echo "Exception ". $e->getCode(). ": ". $e->getMessage()."\n\t";
       echo "on ". $e->getFile(). ":". $e->getLine(). "\n";
   }
?>
```

Which outputs:

```
Exception 9: Something went wrong!
 on /home/httpd/html/test/exception.php:3
```

If you want you can also define your own exception classes; they do not need to be extended from the Exception base class, although that might be the most logical thing to do. If, for example, you want to catch FileIO errors separately you can define your own FileIOException class:

Example 8. User Defined Exception

```
<?php
    define('FILENAME', '/tmp/doesntexist');

    class FileIOException extends Exception {

        function __construct($desc, $filename) {
            $this->message = $desc;
            $this->filename = $filename;
        }

        function getFilename() {
            return $this->filename;
        }
    }
```

```
    try {
        $fp = @fopen(FILENAME, 'r');
        if (!$fp) {
            throw new FileIOException("Couldn't open file!", FILENAME);
        }
    } catch (FileIOException $e) {
        echo "FileIOException: ". $e->getMessage(). ": ". $e->getFilename().  "\n";
    }
?>
```

Which outputs:

```
FileIOException: Couldn't open file!: /tmp/doesntexist
```

If you fail to catch an exception, you will get a message similiar to:

```
Fatal error: Uncaught exception 'fileioexception'! in Unknown on line 0
```

Namespaces

The last thing that we'll cover is namespaces. Namespaces enable you to pack several classes and/or functions together into a logical group and help you to solve naming conflicts. Namespace are declared with the `namespace` keyword. The example below shows how to encapsulates class variables, a class wide constant, a function and a class and how to use those from your script.

Example 9. Namespace

```
<?php
    namespace example {
        var $class_var = 'var';
        const class_const = 'const';

        function printBar() {
            echo "bar\n";
        }

        class test1 {

            static function printFoo() {
                echo "foo\n";
            }

            public function printBaz() {
                echo "baz\n";
            }
        }
    }

    echo example::$class_var."\n";
    echo example::class_const."\n";

    example::printBar();
    example::test1::printFoo();

    $test = new example::test1();
    $test->printBaz();
?>
```

Outputs:

```
var
const
```

```
bar
foo
baz
```

This concludes the short introduction to the new object model, for the latest changes to the still moving Zend Engine 2 please refer to the ZEND_CHANGE file, which can be found online [http://cvs.php.net/cvs.php/ZendEngine2/ZEND_CHANGES?login=2].

About the author

Derick Rethans [http://www.derickrethans.nl/] provides solutions for Internet related problems. Working in his own company, JDI Media Solutions he has acquired vast experience with PHP related projects. He has contributed in a number of ways to the PHP project, including the mcrypt extension, bug fixes, additions and leading the QA team. In his spare time he likes to work on SRM: Script Running Machine and Xdebug, watch movies and travel. You can reach him at d.rethans@jdimedia.nl.