PHP 101 (part 7): The Bear Necessities

Alphabet Soup

So now you know how to create your own functions in PHP, and you've spent the last few days busily inspecting your applications and turning repeated code fragments into functions. But functions are just the tip of the software abstraction iceberg. Lurking underneath is a three-letter acronym that strikes fear into the hearts of most newbie programmers.

OOP.

If you've been programming for a while, you've probably heard the term OOP before – it stands for Object Oriented Programming, and refers to a technique whereby you create program "objects" and then use these objects to build the functionality you need into your program. PHP 5 is very big on OOP – it comes with a brand-spanking-new object model which finally brings PHP objects into conformance with standard OOP principles and offers OO programmers a whole bunch of new goodies to play with.

Wondering how you can get in on this? Well, wonder no more. Your prayers have been answered.

Over the course of this tutorial, I'm going to take a brief look at PHP's OO capabilities (both PHP 4 and PHP 5), together with examples and explanations to demonstrate just how powerful it really is. I'll be covering most of the basics – classes, objects, attributes and methods – and a couple of more advanced concepts – constructors, destructors, private methods and properties, and inheritance. And if you're new to object-oriented programming, or just

apprehensive about what lies ahead, don't worry – I promise this will be a lot less painful than you think. And unlike dentists, I don't lie.

Back To Class

Before beginning, though, let's make sure that you have a clear idea of the concepts involved here.

In PHP, a **class** is simply a set of program statements which perform a specific task. A typical class definition contains both variables and functions, and serves as the template from which to spawn specific instances of that class.

These specific instances of a class are referred to as **objects**. Every object has certain characteristics, or **properties**, and certain pre-defined functions, or **methods**. These properties and methods of the object correspond directly with the variables and functions within the class definition.

Once a class has been defined, PHP allows you to spawn as many instances of the class as you like. Each of these instances is a completely independent object, with its own properties and methods, and can therefore be manipulated independently of other objects. This comes in handy in situations where you need to spawn more than one instance of an object – for example, two simultaneous database links for two simultaneous queries, or two shopping carts.

Classes also help you keep your code modular – you can define a class in a separate file, and include that file only in the scripts where you plan to use the class – and simplify code changes, since you only need to edit a single file to add new functionality to all your

spawned objects.

Animal Antics

To understand this better, pick an animal, any animal. I pick the bear, because I like bears. Now ask yourself, can you consider this bear, within the framework of OOP, as an “object”?

Why not? After all, every bear has certain characteristics – age, weight, sex – which are equivalent to object properties. And every bear can perform certain activities – eat, sleep, walk, run, mate – all of which are equivalent to object methods.

Let’s take it a little further. Since all bears share certain characteristics, it is possible to conceive of a template `Bear()`, which defines the basic characteristics and abilities of every bear on the planet. Once this `Bear()` (“class”) is used to create a new `$bear` (“object”), the individual characteristics of the newly-created Bear can be manipulated independently of other Bears that may be created from the template.

Now, if you sat down to code this class in PHP 5, it would probably look something like this:

```
<?php

// PHP 5

// class definition
```

```
class Bear {

    // define properties
    public $name;

    public $weight;

    public $age;

    public $sex;

    public $colour;
    // define methods
    public function eat() {

        echo $this->name." is eating...
";

    }
    public function run() {

        echo $this->name." is running...
";
        }
        public function kill() {
```

```
        echo $this->name." is killing prey...
";

    }
        public function sleep() {
                echo $this->name." is sleeping...
";

    }

}
?>
```

Given this class, it’s now simple to spawn as many Bears as you like, and adjust the individual properties of each. Take a look:

```
<?php
// my first bear

$daddy = new Bear;

// give him a name

$daddy->name = "Daddy Bear";
// how old is he
```

```
$daddy->age = 8;

// what sex is he

$daddy->sex = "male";

// what colour is his coat

$daddy->colour = "black";
// how much does he weigh

$daddy->weight = 300;
// give daddy a wife

$mommy = new Bear;

$mommy->name = "Mommy Bear";
$mommy->age = 7;

$mommy->sex = "female";

$mommy->colour = "black";
$mommy->weight = 310;
// and a baby to complete the family

$baby = new Bear;
```

```
$baby->name = "Baby Bear";
$baby->age = 1;

$baby->sex = "male";

$baby->colour = "black";
$baby->weight = 180;
// a nice evening in the Bear family

// daddy kills prey and brings it home

$daddy->kill();
// mommy eats it

$mommy->eat();
// and so does baby

$baby->eat();
// mommy sleeps

$mommy->sleep();

// and so does daddy

$daddy->sleep();
// baby eats some more
```

```
$baby->eat();
?>
```

As the illustration above shows, once new objects are defined, their individual methods and variables can be accessed and modified independent of each other. This comes in very handy, as the rest of this tutorial will show.

Going Deeper

Now that you've got the concepts straight, let's take a look at the nitty-gritty of a class definition.

```
<?php

// PHP 5
// class definition

class Bear {
    // define public properties

    public $name;

    public $age;
    // more properties
    // define public methods
```

```
    public function eat() {

        echo $this->name." is eating...
";

        // more code

    }
    // more methods

}
?>
```

Every class definition begins with the keyword class, followed by a class name. You can give your class any name that strikes your fancy, so long as it doesn’t collide with a reserved PHP word. A pair of curly braces encloses all class variables and functions, which are written as you would normally code them.

PHP 5 also introduces the concept of visibility to the object model. Visibility controls the extent to which object properties and methods can be manipulated by the caller, and plays an important role in defining how open or closed your class is. Three levels of visibility exist, ranging from most visible to least visible: **public**, **private** and **protected**. Within the class definition, you can mark the visibility of a property or method by preceding it with one of the keywords – `public`,

`private`, or `protected` .

By default, class methods and properties are `public`; this allows the calling script to reach inside your object instances and manipulate them directly. If you don't like the thought of this intrusion, you can mark a particular property or method as `private` or `protected`, depending on how much control you want to cede over the object's internals (more on this shortly).

Since the PHP 4 object model does not include support for visibility, the class definition above would not work in PHP 4. Instead, you would need to use the following:

```
<?php

// PHP 4
// class definition

class Bear {

    // define properties

    var $name;
        var $weight;

    var $age;
```

```
    var $sex;

    var $colour;
        // define methods

    function eat() {
            echo $this->name." is eating...
";

    }
        function run() {

        echo $this->name." is running...
";
        }
        function kill() {

        echo $this->name." is killing prey...
";

    }
        function sleep() {
            echo $this->name." is sleeping...
";
```

```
  }

}
?>
```

From the above, it should be clear that class properties and methods in PHP 4 are always public …and there ain't nuttin' you can do about that!

In order to create a **new** instance of a class, you use the `new` keyword to assign the newly created object to a PHP variable.

```
<?php
$daddy = new Bear;
?>
```

In English, the above would mean "create a new object of class `Bear()` and assign it to the variable `$daddy` ".

You can now access all the methods and properties of the class via this variable. For example, the code

```
<?php
$daddy->name = "Daddy Bear";
?>
```

would mean "assign the value `Daddy Bear` to the variable `$name` of this specific instance of the class `Bear()`", while the statement

```
<?php
```

`$daddy->sleep();`

`?>`

would mean “execute the function `sleep()` for this specific instance of the class `Bear()`“.

Note the `->` symbol used to connect objects to their properties or methods, and the fact that the `$` symbol is omitted when accessing properties of a class instance (例如`$daddy->name` 中的 name 前的$省掉了).

This And That

In case you need to access functions or variables within the class definition itself, both PHP 4 and PHP 5 offer the `$this` keyword, which is used to refer to “this” class. To see how this works, let’s alter the `eat()` method to accept a number of food units and then add that to the bear’s weight.

```
<?php

// PHP 5
// class definition

class Bear {

    // define properties
```

```
    public $name;

    public $weight;
        // define methods

    public function eat($units) {
                echo $this->name." is eating ".$units." units of food...
";

        $this->weight += $units;
        }

}
?>
```

In this case, the `$this` prefix indicates that the variable to be modified exists within the class – or, in English, “add the argument provided to `eat()` to the variable `$weight` within this object”. The `$this` prefix thus provides a convenient way to access variables and functions which are “local” to the class.

Here’s an example of how it works:

```
<?php
// create instance
```

```
$baby = new Bear;

$baby->name = "Baby Bear";
$baby->weight = 1000;
// now create another instance

// this one has independent values for each property

$brother = new Bear;

$brother->name = "Brother Bear";
$brother->weight = 1000;
// retrieve properties

echo $baby->name." weighs ".$baby->weight." units
";
echo $brother->name." weighs ".$brother->weight." units
";
// call eat()

$baby->eat(100);
$baby->eat(50);

$brother->eat(11);
// retrieve new values

echo $baby->name." now weighs ".$baby->weight." units
```

";

echo $brother->name." now weighs ".$brother->weight." units

";

?>

The output of this will read:

```
Baby Bear weighs 1000 units
[redacted]
Baby Bear is eating 100 units of food...
[redacted]
Brother Bear is eating 11 units of food...
[redacted]
Brother Bear now weighs 1011 units
```

Under Construction

It’s also possible to automatically execute a function when the class is called to create a new object. This is referred to in geek lingo as a **constructor** and, in order to use it, your PHP 5 class definition must contain a special function, `__construct()`.

For example, if you’d like all newly born bears to be brown and weigh 100 units, you could add this to your class definition:

```
<?php

// PHP 5

// class definition
```

```
class Bear {

    // define properties

    public $name;
        public $weight;

    public $age;

    public $colour;




    // constructor

    public function __construct() {

        $this->age = 0;
                $this->weight = 100;

        $this->colour = "brown";

    }
```

```
    // define methods

}
?>
```

In PHP 4, your constructor must have the same name as the class.

Here’s the equivalent code for PHP 4:

```
<?php

// PHP 4
// class definition

class Bear {
        // define properties

    var $name;

    var $weight;

    var $age;

    var $colour;
        // constructor

    function Bear() {
            $this->age = 0;
```

```
        $this->weight = 100;

        $this->colour = "brown";
        }


    // define methods

}
?>
```

Now, try creating and using an instance of the class:

```
<?php
// create instance

$baby = new Bear;

$baby->name = "Baby Bear";

echo $baby->name." is ".$baby->colour." and weighs ".$baby->weight." units at
birth";
?>
```

Here, the constructor automatically sets default properties every

time an object of the class is instantiated. Therefore, when you run the script above, you will see this:

```
Baby Bear is brown and weighs 100 units at birth
```

Hands Off

As noted previously, PHP 5 makes it possible to mark class properties and methods as `private`, which means that they cannot be manipulated or viewed outside the class definition. This is useful to protect the inner workings of your class from manipulation by object instances. Consider the following example, which illustrates this by adding a new `private` variable, `$_lastUnitsConsumed`, to the `Bear()` class:

```
<?php

// PHP 5
// class definition

class Bear {

    // define properties

    public $name;

    public $age;
```

```
    public $weight;

    private $_lastUnitsConsumed;
    // constructor

    public function __construct() {

        $this->age = 0;
        $this->weight = 100;

        $this->_lastUnitsConsumed = 0;

    }



    // define methods

    public function eat($units) {
        echo $this->name." is eating ".$units." units of food...
";

        $this->weight += $units;
        $this->_lastUnitsConsumed = $units;

    }
```

```
    public function getLastMeal() {

    echo "Units consumed in last meal were ".$this->_lastUnitsConsumed."
";
    }


}
?>
```

Now, since the `$_lastUnitsConsumed` variable is declared as `private`, any attempt to modify it from an object instance will fail. Here is an example:

```
<?php
$bob = new Bear;

$bob->name = "Bobby Bear";

$bob->eat(100);
$bob->eat(200);

echo $bob->getLastMeal();

// the next line will generate a fatal error

$bob->_lastUnitsConsumed = 1000;
```

```
?>
```

In a similar way, class methods can also be marked as `private` – try it out for yourself and see.

Extending Yourself

Two of the best things about OOP, whether in PHP 4 or in PHP 5, are extensibility and inheritance. Very simply, this means that you can create a new class based on an existing class, add new features (read: properties and methods) to it, and then create objects based on this new class. These objects will contain all the features of the original parent class, together with the new features of the child class.

As an illustration, consider the following `PolarBear()` class, which **extends** the `Bear()` class with a new method.

```
<?php

// PHP 5

// class definition

class Bear {

    // define properties

    public $name;
```

```
    public $weight;

    public $age;

    public $sex;
        public $colour;
        // constructor

    public function __construct() {

        $this->age = 0;

        $this->weight = 100;
        }
        // define methods

    public function eat($units) {

        echo $this->name." is eating ".$units." units of food...
";
                $this->weight += $units;

    }
        public function run() {

        echo $this->name." is running...
```

```
";
        }
        public function kill() {

        echo $this->name." is killing prey...
";

    }
        public function sleep() {
                echo $this->name." is sleeping...
";

    }

}
// extended class definition

class PolarBear extends Bear {
        // constructor

    public function __construct() {

        parent::__construct();

        $this->colour = "white";
```

```
        $this->weight = 600;
    }
    // define methods

    public function swim() {

        echo $this->name." is swimming...
";

    }

}
?>
```

The `extends` keyword is used to extend a parent class to a child class. All the functions and variables of the parent class immediately become available to the child class. This is clearly visible in the following code snippet:

```
<?php
// create instance of Bear()

$tom = new Bear;
$tom->name = "Tommy Bear";
// create instance of PolarBear()
```

```
$bob = new PolarBear;

$bob->name = "Bobby Bear";
// $bob can use all the methods of Bear() and PolarBear()

$bob->run();

$bob->kill();

$bob->swim();
// $tom can use all the methods of Bear() but not PolarBear()

$tom->run();
$tom->kill();

$tom->swim();
?>
```

In this case, the final call to `$tom->swim()` will fail and cause an error, because the `Bear()` class does not contain a `swim()` method. However, none of the calls to `$bob->run()` or `$bob->kill()` will fail, because as a child of the `Bear()` class, `PolarBear()` inherits all the methods and properties of its parent.

Note how the parent class constructor has been called in the `PolarBear()` child class constructor – it's a good idea to do this so that all necessary initialization of the parent class is carried out when a

child class is instantiated. Child-specific initialization can then be done in the child class constructor. Only if a child class does not have a constructor, is the parent class constructor automatically called.

You can do this in PHP 4, too. Here's a PHP 4 version of the `PolarBear` class definition:

```
<?php

// PHP 4
// extended class definition

class PolarBear extends Bear {
    // constructor
    function PolarBear() {
        parent::Bear();
        $this->colour = "white";
        $this->weight = 600;
    }
    // define methods
    function swim() {
```

```
        echo $this->name." is swimming...
";
    }

}
?>
```

To prevent a class or its methods from being inherited, use the **final** keyword before the class or method name (this is new in PHP 5 and will not work in older versions of PHP). Here's an example, which renders the `Bear()` class un-inheritable (if that's actually a word):

```
<?php

// PHP 5
// class definition

final class Bear {

   // define properties
    // define methods

}
```

```
// extended class definition

// this will fail because Bear() cannot be extended

class PolarBear extends Bear {
    // define methods

}
// create instance of PolarBear()

// this will fail because Bear() could not be extended

$bob = new PolarBear;

$bob->name = "Bobby Bear";

echo $bob->weight;
?>
```