

## Sessions 7 - April 4, 2005

### Outline: OOP in PHP 5

- references
- classes and objects
- class members: methods and variables; arrow syntax
- static versus instance variables; scope resolution operator (::)
- access modifiers and encapsulation
- inheritance
- interfaces
- abstract classes and methods
- Why OOP?
- Assignment: write a class that implements Arithmetic

### References and the & operator

```
<?php
$a = 1;
$b = $a;
$a++;
echo "\$a is now $a and \$b is now $b";
?>
```

Then \$a is be a *copy* of \$b. When this assignment statement is executed, \$a starts life with a value equal to \$b, but thereafter it has its own life and can go its own separate way.

The manual teaches: “References in PHP are a means to access the same variable content by different names. They are not like C pointers, they are symbol table aliases. Note that in PHP, variable name and variable content are different, so the same content can have different names. The most close analogy is with Unix filenames and files - variable names are directory entries, while variable contents is the file itself. References can be thought of as hardlinking in Unix filesystem.” [<http://php.net/manual/en/language.references.php>]

When you do:

```
<?php
$a = 1;
$b =& $a;
$a++;
echo "\$a is now $a and \$b is now $b";
?>
```

it means that  $\$a$  and  $\$b$  point to the same variable. If I now increment  $\$b$  by one and print both  $\$a$  and  $\$b$  I will see that they are both 2.

If I define a function thus:

```
function increment(&$number) {
    $number++;
}

$x = 1;
increment($x);
print $x;
```

The output is 2. The ampersand tells PHP that the incoming parameter is to be passed by reference rather than by value.

To return a reference rather than a value, prepend an ampersand to the function name rather than the parameter. You can put the ampersand in both places -- in front of the incoming parameter as well as the function name.

```
function &increment(&$number) {
    return ++$number;
}

function validate (&$data) {
    $errors = array();
    // code here that might change $data
    return $errors;
}

if ($errors = validate($_POST)) {
    // print errors and
    display_form($_POST);
}
```

This has important implications when it comes to passing data to and returning data from functions. Above, the  $\$_POST$  array, which is used to populate the form, will reflect any changes that `validate()` made to it because we *passed it by reference*, not by *copy* or *value*). Again, we did that by prepending an ampersand to the incoming parameter in the function definition.

In PHP 4, functions pass arguments and return values by value rather than by reference, *including objects, unless* you stipulate otherwise with a `&`. In OO programming with PHP 4 you had to write a lot of ampersands.

In PHP 5, the same is true except *except in the case of objects*. PHP 5 objects are passed, returned, assigned, etc *by reference*. The ampersand is unnecessary unless you want backward compatibility with PHP 4.

## Classes and Objects

We have seen data structures like arrays and we've seen functions. Arrays have state, but don't really do anything, they just have things done to them; functions define behaviors but they don't really contain much by way of data, or state. Imagine a data type that defines both state and behavior, i.e., that has variables holding data and has its own functions defined within it, and there you have an *object*.

Although the above paragraph is not a lie, the full story is of course not quite so simple.

An *object* is an instance of a *class*. The class definition is like a template. It contains function definitions, and declares the variables that it will contain. The functions are called *methods* in OOP parlance. These variables are sometimes called *properties*. They can be scalars, arrays, or other objects. The methods and properties of a class are sometimes know as *members*.

You can think of a dog's definition or zoological description as analogous to a class. It's a four-legged mammal that barks and drools and so forth. But the particular, individual dog who was trying to chew my pants in the park the other day was an instance of the class dog -- she was a dog object. Here is a PHP 4 style Dog class.

```
class Dog {

    /* PHP 4-style 'var' is deprecated in PHP 5 */
    var $name;
    var $gender;
    var $owner_name;

    /* this is a PHP 4 style constructor */
    function Dog($name='') {
        if ($name) { $this->name = $name; }
    }

    function bark() {
        echo "Arf! Arf!";
    }

    function chewLeg(&$something) { // no ampersand required in PHP 5!
        echo "DOG: yum yum yum...<br />";
        $something->setHasChewedLeg(true);
    }

    function drool() { echo "[drools]"; }
}
```

In PHP 4, you declare your class' member variables by using the `var` keyword, as above. In PHP 5, `var` is *deprecated* and we should use *access modifiers* (`public`, `private`, `protected`) instead -- but we will get into that later.

You declare and define your functions in essentially the same way as in plain old procedural PHP.

The way you create or *instantiate* an object is with the `new` keyword.

```
$dog = new Dog();
```

When you instantiate an object, the class' *constructor* gets called. If a class does not explicitly declare an constructor, it gets a free one courtesy of PHP -- that is, you can still instantiate a class that has no declared constructor. *In PHP 4, the constructor's name is the same as the class name.* In PHP 5, your constructor should be called `__construct` (two underscores prepended to the word `construct`). The PHP 4 style is also still supported in PHP 5.

Constructors are generally used for setting up the object's initial state. The data is passed in via the arguments to the constructor call, and the constructor uses it to set the object's properties. In the Dog class above, we gave its constructor an optional `$name` parameter; and if we got a `$name`, we assigned it to our `name` field.

By definition, a constructor returns an instance of the class in which it is defined. You cannot and should not try to return anything explicitly with a `return` statement.

Before you can instantiate an object of that class *the file containing the class definition has to be loaded* (e.g., with `require()` or `include()`). PHP 5 also provides a handy thing called autoloading (<http://php.net/manual/en/language.oop5.autoload.php>)

## **\$this**

If you read the Dog class definition carefully, you probably noticed that the way objects refer to themselves internally is with a special pseudovvariable called *\$this*. `$this` is a reference to the same object instance that called the method (so says the official manual; you may find it easier to think of this as meaning "this" (-: ). Hence the line in the constructor that sets `$this->name` to the `$name` passed to our constructor. Other languages (Java, Javascript) also use `this` while others (Python) speak of `self` but the concept is the same.

## **The arrow syntax**

You invoke object methods (and in PHP 4, you can also access the object properties though in PHP 5, it is subject to access modifiers, as we will see) by using the arrow operator:

```
$object->method();
```

The arrow is the equivalent of the dot in languages like Java and Javascript.

```
<?php
    dog = new Dog();
    $dog->bark();
?>
```

## **Static versus Instance Methods and Variables**

Sometimes you can invoke a method on the class itself rather than on an object of the class. In such cases it is said to be called statically, or in static context. You call a method statically by saying the name of the class, followed by double colons (a/k/a the *scope resolution operator*, a/k/a Paamayim Nekudotayim), followed by the method name and parens (with args if any).

```
class Addition() {
    function add($x,$y) {
        return $x + $y;
    }
}

echo Addition::add(2,2);
```

In PHP 4 you can just go ahead and try it, and see that if the method does not contain any references to `$this`, it will work. Otherwise, it most likely won't because if there is no object, there is no `$this`, and you'll get a complaint about an undefined variable.

In PHP 5 the same thing (calling a non-static method in static context) triggers a *run-time notice* if your error reporting is maxxed out to `E_STRICT`. In PHP 5, *you can declare variables and functions to be static with the `static` keyword*. This means you intend these methods to be invoked on the class, not on instances of the class.

Whereas in PHP 4, the parser chokes on the `static` keyword in this selfsame context. In other words, when you code PHP these days you have to decide whether you are writing for PHP 4, for PHP 5, or for PHP 5 with backward compatibility.

Sometimes you may write utility methods that you would like to be able to invoke statically for the sake of convenience and efficiency. An example is the `Text/Password` package from PEAR. Suppose all you want is to generate one strong, random password; you don't want to instantiate any object, because you don't plan to do anything further with it. And indeed you don't have to, because you can call the `Password` class' `create()` method statically:

```
<?php
require_once "Text/Password.php";
$password = Text_Password::create(8, 'unpronounceable', 'alphanumeric')
?>
```

`static` is an unfortunate term, because it is a bit misleading; it does not mean constant. `static` variables are for situations where you want a the variable shared by all instances of the class. Suppose we want to keep track of the `Dog` population: simply have your constructor increment a 'static' counter whenever your constructor is called. Here is another `Dog` implementation that works in PHP 5 (but not in 4).

```
<?php
class Dog {

    private $name;
    private $gender;
    private $owner_name;

    private static $count = 0;
```

```

/* this is a PHP 5 style constructor */
function __construct($gender) {

    $this->$gender = $gender;
    self::$count++;

}

function bark() {
    echo "Arf! Arf!";
}

function chewLeg(&$something) { // no ampersand required in PHP 5!
    echo "DOG: yum yum yum...<br />";
    $something->setHasChewedLeg(true);
}

function drool() {
    echo "[drools]";
}

function setOwnerName($owner) { $this->owner_name = $owner; }

function getOwnerName() { return $this->owner_name; }

function setName($name) { $this->name = $name; }

function getName() { return $this->name; }

function getPopulation() { return self::$count; }

/* destructors, if defined, run automatically when PHP
garbage-collects your object. Nice for closing things like
files, databases*/

function __destruct() {
    echo "$this->name gone to doggie heaven, dog population now "
        . --self::$count;
}

} // end class definition

$dog = new Dog('male');
$dog->setName('Grep');
$dog->bark();
$other_dog = new Dog('female');
$other_dog->setName('Propecia');
echo " Total dog population is now {$dog->getPopulation()}!";

?>

```

From within a class, you use the keyword `self` to refer to the class as opposed to the instance.

## Access Modifiers (Visibility) in PHP 5

Notice how the Dog's properties were declared as `private`, and we allowed access to them through `getXXX()` and `setXXX()` methods, often called *accessor* methods. This is a common

technique for controlling how you let other code that uses your class – client code – manipulate the state of your object. In OOP parlance this is known as *encapsulation*, and is widely hailed as a Very Good Thing.

We kept the Dog population counter private because we do not want anyone else to touch it; our class knows when to increment or decrement it. (I bet the census bureau wishes it had it so easy.) More details about access modifiers:

**Visibility** The visibility of a property or method can be defined by prefixing the declaration with the keywords *public*, *protected* or *private*. Whatever is *public* items can be accessed everywhere. *protected* limits access to inherited classes (and to the class that defines the item). *private* limits visibility only to the class that defines the item. The default (when there is no access modifier) is *public*.

Again, there are compatibility issues to keep in mind: if you run code in PHP 4 that declares something as *public* (or *protected*, or *private*) -- just as you are supposed to do in PHP 5 -- it won't even parse.

For all the gory details about the new PHP 5 object model please see <http://php.net/manual/en/language.oop5.php> for complete information.

## Inheritance

One of the most powerful features of OOP is *inheritance*. If you declare a class as being a subclass of some other class, you inherit all the behavior and data from the parent (subject to access modifiers, in PHP 5). You do this with the `extends` keyword in the class declaration.

Let's recast the Dog class as an extension of a class Animal.

```
class Animal {
    protected $gender;

    public function __construct($gender) {
        $this->gender = $gender;
    }

    public function breathe() {
        echo "[breathe breathe breathe....]" ;
    }

    public function eat() {
        echo "[eat eat eat....]" ;
    }
}
```

```

    public function reproduce($gender) {

        return new $this($gender);

    }

    public function sleep($seconds = 2) {
        if (! headers_sent()) {ob_end_flush();}
        ini_set('output_buffering' , 'Off');
        echo "sleeping..." ;
        flush();
        sleep($seconds); // annoying, eh?
        echo "ok, I woke up..." ;
    }
}

class Dog extends Animal {

    function __construct($gender) {

        parent::__construct($gender);
        self::$count++;

    }
    /* otherwise same as before */
}

$dog = new Dog('male');
$dog->setName('Grep');
echo "<br />{$dog->getName()} is eating... " ;
$dog->eat();
echo "<br />{$dog->getName()} is sleeping... " ;
$dog->sleep();
echo "<br />";
$dog->bark();

```

Dog is now automatically endowed with sleep(), eat() and reproduce() methods. Notice how the revised Dog constructor called the parent constructor. You can use the parent keyword for that... in PHP 5. A subclass can invoke any parental method or access any property with this syntax, subject to access restrictions.

## Overriding

You can also override an inherited method simply by redefining it – likewise in PHP 4 – unless that method is qualified as `final` in its declaration, in which case you would die with a fatal error if you tried to override it. Imagine a Poodle class – a more specialized Dog. It might override the Dog's relatively benign bark() method like so:

```

require_once('./Dog.php');

class Poodle extends Dog {

    public final function bark() {
        echo str_repeat("YAP ",100) . "YAP!";
    }
}

$poodle = new Poodle('female');

```



```

    public function showAffection() {
        printf(
            '%s: "I am loyal and adoring and trying to please you as
            always"...<br />' ,__CLASS__);
    }
}

```

Here is another implementation of HousePet:

```

class Cat extends Animal implements HousePet {
    public function annoyOwner() {
        printf("%s is now annoying owner...<br />",__CLASS__);
    }
    public function incurVeteranarianBills($dollars) {
        printf("%s is now costing owner %d dollars...<br />",
            __CLASS__,$dollars);
    }
    public function destroyHouseholdItems() {
        printf(
            "%s is now destroying just 1 or 2 household items...<br />",
            __CLASS__);
    }
    public function showAffection() {
        printf(
            '%s: "I usually spurn you, but right now I want you to pet
            me"...<br />' ,__CLASS__);
    }
}

```

*Type hints* specify the class that an argument must belong to. Imagine a class called Household:

```

class HouseHold {
    private $pets=array();
    function addPet(HousePet $pet) {
        $this->pets[] = $pet;
    }
    function showPetBehaviors() {
        foreach ($this->pets as $pet) {
            $pet->annoyOwner();
            $pet->incurVeteranarianBills(600);
            $pet->destroyHouseholdItems();
            $pet->showAffection();
        }
    }
}

```

```
}
```

Notice how `addPet()` stipulates that the incoming thing has to be a `HousePet`. PHP regards a class that implements an interface as an instance of that interface, so it works. The interesting thing here is that `addPet()` methods does not need to know the class of the thing it's getting, and doesn't need to care, because whatever it is guaranteed to implement the `HousePet` interface. In big projects, this can save considerable maintenance and development pain. As long as it implements the interface. I could give you a completely new and different implementation of `HousePet` – with a different inheritance tree from the previous version, different everything -- for you to use in your `Household` class and you will not have to update anything.

## Abstract Classes and Methods

The official documentation instructs us:

“PHP 5 introduces abstract classes and methods. [It is not possible to] create an instance of a class that has been defined as abstract. Any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature; they cannot define the implementation.

“The class that implements the abstract method must define with the same visibility or weaker. If the abstract method is defined as protected, the function implementation must be defined as either protected or public.”

In other words, if you are sure that your class is something that really should be extended rather than instantiated as is, you can declare the whole class – or just one or more of its methods – as `abstract`. Implementation is deferred to child classes. This feature allows you, in a sense, to combine the notions of inheritance and interfaces in the same class. You can pass on some methods by inheritance while delegating the implementation of others to child classes. You can think of `abstract` as the reverse of `final`. The latter says you cannot override this; the former say you have to override this. In our earlier `Animal` example, we might have defined that class as `abstract` on the grounds that you can't instantiate a generic animal, it has to belong to some species (subclass).

In a hypothetical PHP University, you might well want to model certain entities in a way the involves abstract classes. Such a model would have `Student` objects and `Instructor` objects. You might want these classes to be responsible for persisting their own states in a database, and or for input validation in much the same way we have been doing with procedural (non-OO) code. You know you want to obligate your subclasses to do certain things, but you can't say how<sup>1</sup>. Why not design an abstract `PHP_Universitarian` and then create `Student` and `Instructor` classes that extend it.

```
abstract class PHP_Universitarian {  
  
    protected $lastname, $firstname, $middlename, $email, $id, $lastmod;
```

---

<sup>1</sup> PHP can require a subclass or an implementation of an interface to do something, but PHP can't control how well it does it. You could just write stub methods that nominally meet the requirement but don't do anything at all. *Unit testing* is one of the methods people use to ensure that things work as they are supposed to.

```

abstract function validate(&$data);
abstract function update();
abstract function insert();
abstract function delete();
abstract function findById($id);

function setFrom($data) {

    $props = array_keys(get_object_vars($this));

    foreach ($props as $prop) {
        if (array_key_exists($prop,$data)) {
            $this->$prop = $data[$prop];}
    }
}
/* more methods that you want all subclasses to be able to use */
}

```

## Exceptions

PHP 5 also features an error handling mechanism called Exceptions that provides a convenient and elegant way of handling error conditions, and relieves you of the burden of manually checking the return value of countless function calls. Time permitting, we will start in this session to talk about Exceptions, the `throw` keyword and the `try` and `catch` constructs; otherwise, we will get to this next week. In the meantime you should check out <http://php.net/exceptions> if you're curious

## Why OOP

OOP is not a panacea for all that ails software development, and procedural code is not per se inferior to OO code. But OOP does have much to offer:

- It makes it easier to design things in a rational, modularized way, especially in large projects.
- Code re-use can lead to a virtuous circle in which it gets (re-)used, so more bugs are exposed and squashed, so it further improves, and gets used even more. OOP by its nature strongly encourages re-use (e.g., through inheritance), whether it is re-use within the same application, or re-use resulting from publishing more general code for others to re-use in their applications.
- Many third-party libraries (such as PEAR and Smarty) have OO interfaces. You have to speak OO if you want to use them.
- If PHP is your first language and you plan to study others that are OO, learning OOP in PHP 5 will prove useful later.
- Many more reasons that I can't think of off the top of my head (-:

## Assignment: Implement Restaurant\_Arithmetic

Write a class named `MyRestaurantCalculator` that implements this `Restaurant_Arithmetic` interface (which you can download from the class site).

```
interface Restaurant_Arithmetic {
```

```

/* return the sum of X and Y */
function add($x,$y);

/* return the absolute difference between $x and $y
   that is, $object->subtract(2,3) and $object->(3,2) should
   both return 1. see the abs function for help.
*/
function subtract($x,$y);

/* return the product of x and y rounded it off
   to the nearest cent */
function multiply($x,$y);

/* return x divided by y rounded to the nearest cent
   if y is zero, just return null before you divide*/
function divide($x,$y);

/* return $amount plus the amount times $percent/100
rounded it off to the nearest cent (see the round() function for help)
in other words, $object->addSalesTaxTo(113.43,8.25) should return
123.03
*/
function addSalesTaxTo($amount,$percent);

/*
addTipTo is actually the same as the other, so make one
an alias of the other. e.g., return $this->other_func($arg)
*/
function addTipTo($amount,$percent);

/* return the sales tax on $amount at $percent percent */
function computeSalesTaxOn($amount,$percent);
}

```

Here is some test code you should run against your MyRestaurantCalculator class. Please put this as well as the Restaurant\_Arithmetic interface in the same file as your class definition, save it as MyRestaurantCalculator.php, and upload it to the class site. I should be able to run your MyRestaurantCalculator.php and see just the test result output. Don't change any of the supplied code, just provide the MyRestaurantCalculator class definition.

```

$calc = new MyRestaurantCalculator();

if (! $calc instanceof Restaurant_Arithmetic ) {
    exit("Oops. You failed the assignment." );
} else {
    echo "OK, class implements the interface<br />\n" ;
}

$result = $calc->add(23.54, 143.76);
echo $result == 23.54 + 143.76 ? "add method OK" : "add method failed";

echo "\n<br />\n";

$result = $calc->subtract(3,2);
echo $result == 1 ? "subtract method OK" : "subtract method failed";

echo "\n<br />\n";
$result = $calc->subtract(33.43,278.31);

```

```
echo $result == 244.88 ? "subtract method OK" : "subtract method failed";
echo "\n<br />\n";

$result = $calc->multiply(88.32,4);
echo $result == 353.28 ? "multiply method OK" : "multiply method failed";
echo "\n<br />\n";

$result = $calc->divide(353.28,4);
echo $result == 88.32 ? "divide method OK" : "divide method failed";
echo "\n<br />\n";

$result = $calc->divide(353.28,0);
echo $result == NULL ? "divide method OK" : "divide method failed";
echo "\n<br />\n";

$result = $calc->computeSalesTaxOn(312.48,8.25);
echo $result == 25.78 ? "computeSalesTaxOn method OK" : "computeSalesTaxOn
method failed";
echo "\n<br />\n";

$result = $calc->addSalesTaxTo(312.48,8.25);
echo $result == 338.26 ? "addSalesTaxTo method OK" : "addSalesTaxTo method
failed";
echo "\n<br />\n";

$result = $calc->addTipTo(312.48,18);
echo $result == 368.73 ? "addTipTo method OK" : "addTipTo method failed";
```

Happy results will look something like this in a browser:

```
class implements the interface: OK
add method OK
subtract method OK
subtract method OK
multiply method OK
divide method OK
divide method OK
computeSalesTaxOn method OK
addSalesTaxTo method OK
addTipTo method OK
```

## Reading

Sklar, *Learning PHP 5* pp 242-244  
<http://php.net/manual/en/language.oop5.php>  
Atkinson, *Core PHP Programming* Chapter 6

---

Last modified 04-Apr-2005 07:23 am