Object Oriented Programming in PHP5

A WebApp Tutorial

Adrian Giurca

Chair of Internet Technology, Institute for Informatics

October 15, 2006

Revision History

Revision 1	Sept 20, 2005
Revision 2	Sept 13, 2006

Table of Contents

How "object-oriented" is PHP?

Let's try an answer:

- *Single inheritance.* PHP allows a class definition to inherit from another class, using the `extends` clause. Both member variables and member functions are inherited.
- *Multiple inheritance.* PHP offers no support for multiple inheritance and no notion of interface inheritance as in Java. Each class inherits from, at most, one parent class (though a class may implement many interfaces).
- *Constructors.* Every class can have one constructor function, which in PHP is called `__construct()`. Note that there are two underscore characters at the front of that function name. Constructors of parent classes are not automatically called but must be invoked explicitly.
- *Destructors.* PHP supports explicit destructor functions as of version 5. The destructor function of a class is always called `__destruct()`.
- *Encapsulation/access control.* PHP supports public, private, and protected properties and methods as of version 5.

- *Polymorphism/overloading.* PHP supports polymorphism in the sense of allowing instance of subclasses to be used in place of parent instances. The correct method will be dispatched to at runtime. There is no support for method overloading, where dispatch happens based on the method's signature-each class only has one method of a given name.
- *Static (or class) functions.* PHP offers static properties and static methods as of version 5. It is also possible to call methods via the `Classname::function()` syntax.
- *Introspection.* PHP offers a wide variety of functions here, including the capability to recover class names, methods names, and properties names from an instance.

In the next section, we cover the basic PHP syntax for OOP from the ground up, with some simple examples.

1. Basic PHP Constructs for OOP

The general form for defining a new class in PHP is as follows:

```
class MyClass extends MyParent {
 var $var1;
 var $var2 = "constant string";
 function myfunc ($arg1, $arg2) {
  //...
 }
 //...
}
```

As an example, consider the simple class definition in the listing below, which prints out a box of text in HTML:

```
class TextBoxSimple {
 var $body_text = "my text";
 function display() {
  print("<table><tr><td>$this->body_text");
  print(“</td></tr></table>");
 }
}
```

In general, the way to refer to a property from an object is to follow a variable containing the object with `->` and then the name of the property. So if we had a variable `$box` containing an object instance of the class `TextBox`, we could retrieve its `body_text` property with an expression like:

```
$text_of_box = $box->body_text;
```

Notice that the syntax for this access does not put a `$` before the property name itself, only the `$this` variable.

After we have a class definition, the default way to make an instance of that class is by using the `new` operator.

```
$box = new TextBoxSimple;
$box->display();
```

The correct way to arrange for data to be appropriately initialized is by writing a constructor function-a special function called `__construct()`, which will be called automatically whenever a new instance is created.

```
class TextBox {
 var $bodyText = "my default text";
 // Constructor function
 function __construct($newText) {
  $this->bodyText = $newText;
 }
 function display() {
  print("<table><tr><td>$this->bodyText");
  print(“</td></tr></table>");
 }
}

// creating an instance
```

```
$box = new TextBox("custom text");
$box->display();
```

PHP class definitions can optionally inherit from a superclass definition by using the `extends` clause. The effect of inheritance is that the subclass has the following characteristics:

- Automatically has all the property declarations of the superclass.
- Automatically has all the same methods as the superclass, which (by default) will work the same way as those functions do in the superclass.

In addition, the subclass can add on any desired properties or methods simply by including them in the class definition in the usual way.

```
class TextBoxHeader extends TextBox{
 var $headerText;
 // CONSTRUCTOR
 function __construct($newHeaderText, $newBodyText) {
  $this->headerText = $newHeaderText;
  $this->bodyText = $newBodyText;
 }
 // MAIN DISPLAY FUNCTION
 function display() {
  $header_html = $this->make_header($this->headerText);
  $body_html = $this->make_body($this->bodyText);
  print("<table><tr><td>\n");
  print("$header_html\n");
  print("</td></tr><tr><td>\n");
  print("$body_html\n");
  print("</td></tr></table>\n");
 }
 // HELPER FUNCTIONS
 function make_header ($text) {
  return($text);
 }
 function make_body ($text) {
 return($text);
 }
}
```

Function definitions in subclasses override definitions with the same name in superclasses. This just means that the overriding definition in the more specific class takes precedence and will be the one actually executed.

Before we move onto the more advanced features of PHP's version of OOP, it's important to discuss issues of scope—that is, which names are meaningful in what way to different parts of our code. It may seem as though the introduction of classes, instances, and methods have made questions of scope much more complicated. Actually, though, there are only a few basic rules we need to add to make OOP scope sensible within the rest of PHP:

- Names of properties and methods are never meaningful to calling code on their own-they must always be reached via the `->` construct. This is true both outside the class definition and inside methods.
- The names visible within methods are exactly the same as the names visible within global functions-that is, methods can refer freely to other global functions, but can't refer to normal global properties unless those properties have been declared global inside the method definition.

These rules, together with the usual rules about variable scope in PHP, are respected in the intentionally confusing example in the listing below. What number would you expect that code to print when executed?

```
$myGlobal = 3;
function myFunction ($myInput) {
 global $myGlobal;
 return($myGlobal * $myInput);
}

class MyClass {
 var $myProperty;
```

```
 function __construct($myConstructorInput) {
  $this->myProperty = $myConstructorInput;
 }
 function myMethod ($myInput) {
  global $myGlobal;
  return($myGlobal * $myInput * myFunction($this->myProperty));
 }
}

$myInstance = new MyClass(4);
print("The answer is: ".$myInstance->myMethod(5));
```

The answer is: `180` (or `3 * 5 * (3 * 4)`). If any of these numerical variables had been undefined when multiplied, we would have expected the variable to have a default value of `0`, making the answer have a value of `0` as well. This would have happened if we had:

- Left out the `global` declaration in `myFunction()`
- Left out the `global` declaration in `myMethod()`
- Referred to `$myProperty` rather than `$this->myProperty`

2. Advanced OOP Features

2.1. Public, Private, and Protected Members

Unless you specify otherwise, properties and methods of a class are `public`. That is to say, they may be accessed in three possible situations:

- From outside the class in which it is declared;
- From within the class in which it is declared;
- From within another class that implements the class in which it is declared;

If you wish to limit the accessibility of the members of a class, you should use `private` or `protected`.

By designating a member *private*, you limit its accessibility to the class in which it is declared. The private member cannot be referred to from classes that inherit the class in which it is declared and cannot be accessed from outside the class.

```
class MyClass {
 private $colorOfSky = "blue";
 $nameOfShip = "Java Star";
 function __construct($incomingValue) {
  // Statements here run every time an instance of the class
  // is created.
 }
 function myPublicFunction ($myInput) {
  return("I'm visible!");
 }
 private function myPrivateFunction ($myInput) {
  global $myGlobal;
  return($myGlobal * $myInput * myFunction($this->myProperty));
 }
}
```

A *protected* property or method is accessible in the class in which it is declared, as well as in classes that extend that class. Protected members are not available outside of those two kinds of classes, however.

```
class MyClass {
 protected $colorOfSky = "blue";
 $nameOfShip = "Java Star";
 function __construct($incomingValue) {
  // Statements here run every time an instance
```

```
  // of the class is created.
 }
 function myPublicFunction ($myInput) {
  return("I'm visible!");
 }
 protected function myProtectedFunction ($myInput) {
  global $myGlobal;
  return($myGlobal * $myInput * myFunction($this->myProperty));
 }
}
```

2.2. Interfaces

In large object-oriented projects, there is some advantage to be realized in having standard names for methods that do certain work. In PHP5, it is also possible to define an interface, like this:

```
interface Mail {
 public function sendMail();
}
```

then, if another class implemented that interface, like this:

```
class Report implements Mail {
 // Definition goes here
}
```

it would be required to have a method called `sendMail`. It's an aid to standardization.

2.3. Constants

A constant is somewhat like a variable, in that it holds a value, but is really more like a function because a constant is *immutable*. Once you declare a constant, it does not change.

```
class MyClass {
 const REQUIRED_MARGIN = 1.3;
 function __construct($incomingValue) {
  // Statements here run every time an instance of the class
  // is created.
 }
}
```

In that class, `REQUIRED_MARGIN` is a constant. It is declared with the keyword `const`, and under no circumstances can it be changed to anything other than `1.3`. Note that the constants name does not have a leading `$`, as variable names do.

2.4. Abstract Classes

An abstract class is one that cannot be instantiated, only inherited. You declare an abstract class with the keyword `abstract`, like this:

```
abstract class MyAbstractClass {
 abstract function myAbstractFunction() {
 }
}
```

Note that function definitions inside an abstract class must also be preceded by the keyword `abstract`. It is not legal to have abstract function definitions inside a non-abstract class.

2.5. Simulating class functions

Some other OOP languages make a distinction between *instance* properties, on the one hand, and *class* or *static* properties on the other. Instance properties are those that every instance of a class has a copy of (and may possibly modify individually); class properties are shared by all instances of the class. Similarly, instance methods depend

on having a particular instance to look at or modify; class (or static) methods are associated with the class but are independent of any instance of that class.

In PHP, there are no declarations in a class definition that indicate whether a function is intended for per-instance or per-class use. But PHP does offer a syntax for getting to functions in a class even when no instance is handy. The `::` syntax operates much like the `->` syntax does, except that it joins class names to member functions rather than instances to members. For example, in the following implementation of an extremely primitive calculator, we have some methods that depend on being called in a particular instance and one methods that does not:

```
class Calculator{
 var $current = 0;
 function add($num) {
  $this->current += $num;
 }
 function subtract($num) {
  $this->current -= $num;
 }
 function getValue() {
  return($current);
 }
 function pi() {
  return(M_PI); // the PHP constant
 }
}
```

We are free to treat the `pi()` methods as either a class methods or an instance methods and access it using either syntax:

```
$calcInstance = new Calculator;
$calcInstance->add(2);
$calcInstance->add(5);
print("Current value is ".$calcInstance->current ."<br/>");
print("Value of pi is ".$calcInstance->pi()."<br/>");
print("Value of pi is ".Calculator::pi()."<br/>");
```

This means that we can use the `pi()` function even when we don't have an instance of `Calculator` at hand.

2.6. Calling parent functions

Asking an instance to call a function will always result in the most specific version of that function being called, because of the way overriding works. If the function exists in the instance's class, the parent's version of that function will not be executed.

Sometimes it is handy for code in a subclass to explicitly call functions from the parent class, even if those names have been overridden. It's also sometimes useful to define subclass functions in terms of superclass functions, even when the name is available.

2.6.1. Calling parent constructors

Look to the following example:

```
class Name{
 var $_firstName;
 var $_lastName;
 function Name($first_name, $last_name){
  $this->_firstName = $first_name;
  $this->_lastName = $last_name;
 }
 function toString() {
  return($this->_lastName.", ".$this->_firstName);
 }
}

class NameSub1 extends Name{
 var $_middleInitial;
```

```
 function NameSub1($first_name, $middle_initial, $last_name) {
  Name::Name($first_name, $last_name);
  $this->_middleInitial = $middle_initial;
 }
 function toString() {
  return(Name::toString()." " .$this->_middleInitial);
 }
}
```

In this example, we have a superclass (`Name`), which has a two-argument constructor, and a subclass (`NameSub1`), which has a three-argument constructor. The constructor of `NameSub1` functions by calling its parent constructor explicitly using the `::` syntax (passing two of its arguments along) and then setting an additional property. Similarly, `NameSub1` defines its nonconstructor `toString()` function in terms of the superclass function that it overrides.

It might seem strange to call `Name::Name()` here, without reference to `$this`. The good news is that both `$this` and any member variables that are local to the superclass are available to a superclass method when invoked from a subclass instance.

2.6.2. The special name `parent`

There is a stylistic objection to the previous example, which is that we have hardcoded the name of a superclass into the code for a subclass. Some would say that this is bad style because it makes it harder to revise the class hierarchy later. A fix is to use the special name `parent`, which when used in a method, always refers to the superclass of the current class. Here is a revised version of the example using `parent` rather than `Name`:

```
class NameSub2 extends Name{
 var $_middleInitial;
 function NameSub2($firstName, $middleInitial,$lastName) {
  $parentClass = get_parent_class($this);
  parent::$parentClass($firstName, $lastName);
  $this->_middleInitial = $middleInitial;
 }
 function toString() {
  return(parent::toString()." ".$this->_middleInitial);
 }
}
```

2.7. Serialization

Serialization of data means converting it into a string of bytes in such a way that you can produce the original data again from the string (via a process known, unsurprisingly, as *unserialization*). After you have the ability to serialize/unserialize, you can store your serialized string pretty much anywhere (a system file, a database, and so on) and recreate a copy of the data again when needed.

PHP offers two functions, `serialize()` and `unserialize()`, which take a value of any type (except type `resource`) and encode the value into string form and decode again, respectively.

Here is a quick example, which we'll extend later in this section:

```
class ClassToSerialize {
 var $storedStatement = "data";
 function __construct($statement) {
  $this->storedStatement = $statement;
 }
 function display (){
  print($this->storedStatement."<br/>");
 }
}

$instance1 = new ClassToSerialize("You're objectifying me!");
$serialization = serialize($instance1);
$instance2 = unserialize($serialization);
$instance2->display();
```

This class has just one property and a couple of methods, but it's sufficient to demonstrate that both properties and methods can survive serialization.

PHP provides a hook mechanism so that objects can specify what should happen just before serialization and just after unserialization. The special member function `__sleep()` (that's two underscores before the word *sleep*), if defined in an object that is being serialized, will be called automatically at serialization time. It is also required to return an array of the names of variables whose values are to be serialized. This offers a way to not bother serializing member variables that are not expected to survive serialization anyway (such as database resources) or that are expensive to store and can be easily recreated. The special function `__wakeup()` (again, two underscores) is the flip side-it is called at unserialization time (if defined in the class) and is likely to do the inverse of whatever is done by `__sleep()` (restore database connections that were dropped by `__sleep()` or recreate variables that `__sleep()` said not to bother with).

```
class ClassToSerialize2 {
 var $storedStatement = "data";
 var $easilyRecreatable = “data again”;
 function __construct($statement) {
  $this->storedStatement = $statement;
  $this->easilyRecreatable = $this->storedStatement." Again!";
 }
 function __sleep() {
  // Could include DB cleanup code here
  return array('storedStatement');
 }
 function __wakeup() {
  // Could include DB restoration code here
  $this->easilyRecreatable =$this->storedStatement." Again!";
 }
 function display (){
  print($this->easilyRecreatable."<br/>");
 }
}

$instance1 = new ClassToSerialize2("You're objectifying me!");
$serialization = serialize($instance1);
$instance2 = unserialize($serialization);
$instance2->display();
```

The serialization mechanism is pretty reliable for objects, but there are still a few things that you must know:

- The code that calls `unserialize()` must also have loaded the definition of the relevant class. (This is also true of the code that calls `serialize()` too, of course, but that will usually be true because the class definition is needed for object creation in the first place.)
- Object instances can be created from the serialized string only if it is really the same string (or a copy thereof). A number of things can happen to the string along the way, if stored in a database (make sure that slashes aren't being added or subtracted in the process), or if passed as `URL` or `form` arguments. (Make sure that your URL-encoding/decoding is preserving exactly the same string and that the string is not long enough to be truncated by length limits.)
- If you choose to use `__sleep()`, make sure that it returns an array of the variables to be preserved; otherwise no variable values will be preserved. (If you do not define a `__sleep()` function for your class, all values will be preserved.)

See also the current manual for new changes.

2.8. Introspection Functions

Introspection allows the programmer to ask objects about their classes, ask classes about their parents, and find out all the parts of an object without have to crunch the source code to do it. Introspection also can help you to write some surprisingly flexible code, as we will see.

Table 1. Class/Object Functions

Function	Description	Operates on Class Names	Operates on Instances	As of PHP Version
`get_class()`	Returns the name of the class an object belongs to.	No	Yes	4.0.0
`get_parent_class()`	Returns the name of the superclass of the given	Yes(as of PHP 4.0.5 instance or class. v.4.0.5)	Yes	4.0.0
`class_exists()`	Returns `TRUE` if the string argument is the name of a class, `FALSE` otherwise.	Yes	No	4.0.0
`get_declared_classes()`	Returns an array of strings representing names of classes defined in the current script.	N/A	N/A	4.0.0
`is_subclass_of()`	Returns `TRUE` if the class of its first argument (an object instance) is a subclass of the second argument (a class name), `FALSE` otherwise.	No	Yes	4.0.0
`is_a()`	Returns `TRUE` if the class of its first argument (an object instance) is a subclass of the second argument (a class name), or is the same class, and `FALSE` otherwise.	No	Yes	4.2.0
`get_class_vars()`	Returns an associative array of var/value pairs representing the name of variables in the class and their default values. Variables without default values will not be included.	Yes	No	4.0.0
`get_object_vars()`	Returns an associative array of var/value pairs representing the name of variables in the instance and their default values. Variables without values will not be included.	No	Yes	4.0.0

Function	Description	Operates on Class Names	Operates on Instances	As of PHP Version
`method_exists()`	Returns `TRUE` if the first argument (an instance) has a method named by the second argument (a string) and `FALSE` otherwise.	No	Yes	4.0.0
`get_class_methods()`	Takes a string representing a method name, an instance that should have such a method, and additional arguments. Returns the result of applying the method (and the arguments) to the instance.	No	Yes	4.0.0
`call_user_method()`	Takes a string representing a method name, an instance that should have such a method, and additional arguments. Returns the result of applying the method (and the arguments) to the instance.	No	Yes	4.0.0
`call_user_method_array()`	Same as `call_user_method()`, except that it expects its third argument to be an array containing the arguments to the method.	No	Yes	4.0.5

Example 1. Matching variables and DB columns

One frequent use for PHP objects in database-driven systems is as a wrapper around the entire database API. The theory is that the wrapper insulates the code from the specific database system, which will make it trivial to swap in a different RDBMS when the technical needs change.

Another use that is almost as common (and that your authors like better) is to have object instances correspond to database result rows. In particular, the process of reading in a result row looks like instantiating a new object that has member variables corresponding to the result columns we care about, with extra functionality in the member functions. As long as the fields and columns match up (and as long as you can afford object instantiation for every row), this can be a nice abstraction away from the database.

A repetitive task that arises when writing this kind of code is assigning database column values to member variables, in individual assignment statements. This feels like it should be unnecessary, especially when the columns and the corresponding variables have exactly the same names. In this example, we try to automate this process.

Let's start with an actual database table. Following are the MySQL statements necessary to create a simple table and insert one row into it:

```
mysql> create table book
(id int not null primary key auto_increment,
author varchar(255), title varchar(255),
publisher varchar(255));
mysql> insert into book (author, title, publisher)
values ("Robert Zubrin", "The Case For Mars","Touchstone");
```

Because the `id` column is auto-incremented, it will happen to have the value `1` for this first row.

Now, let's say that we want a `Book` object that will exactly correspond to a row from this table, with fields corresponding to the DB column names. There's no way around actually defining the variable names (because PHP doesn't let us dynamically add variables to classes), but we can at least automate the assignment.

The code in listing below assumes a database called `oop` with the table created as above, and also that we have a file called `dbconnect_vars` that sets `$host`, `$user`, and `$pass` appropriately for our particular MySQL setup (the code assumes the connection works, that the row was retrieved successfully, and so on). The main point we want to highlight is the hack in the middle of the `Book` constructor.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
 <title>Books</title>
 <?php
 include_once("dbconnect_vars.php");
 class Book{
  var $id;
  // variables corresponding to DB columns
  var $author = "DBSET";
  var $title = "DBSET";
  var $publisher = "DBSET";
  function __construct($db_connection, $id) {
   $this->id = $id;
   $query = "select * from book "."where id = $id";
   $result = mysql_query($query, $db_connection);
   $db_row_array = mysql_fetch_array($result);
   $class_var_entries = get_class_vars(get_class($this));
   while ($entry = each($class_var_entries)) {
    $var_name = $entry['key'];
    $var_value = $entry['value'];
    if ($var_value == "DBSET") {
     $this->$var_name = $db_row_array[$var_name];
    }
   }
  }
  function toString () {
```

```
   $return_string = "BOOK<br/>";
   $class_var_entries = get_class_vars(get_class($this));
   while ($entry = each($class_var_entries)) {
    $var_name = $entry['key'];
    $var_value = $this->$var_name;
    $return_string .="$var_name: $var_value<br/>";
   }
   return($return_string);
  }
 }

 $connection = mysql_connect($host, $user, $pass) or die("Could not connect to DB");
 mysql_select_db("oop");
?>
 </head>
<body>
 <?php
 $book = new Book($connection, 1);
 $book_string = $book->toString();
 echo $book_string; ?>
</body>
</html>
```

The database query returns all columns from the `book` table, and the values are indexed in the result array by the column names. The constructor then uses `get_class_vars()` to discover all the variables that have been set in the object, tests them to see if they have been bound to the string `"DBSET"`, and then sets those variables to the value of the column of the same name. The result is the output:

```
BOOK
Author: Robert Zubrin
Title: The Case For Mars
Publisher: Touchstone
```

3. OOP Style in PHP. The PEAR Coding Style

We offer in the following some brief notes on writing readable, maintainable PHP OOP code. For more information on the coding style, see the PEAR Web site (at http://pear.php.net).

PEAR recommends that class names begin with an uppercase letter and (if in a PEAR approved directory hierarchy of packages) have that inclusion path in the class name, separated by underscores. So your class that counts words, and which belongs to a PEAR package called `TextUtils`, might be called `TextUtils_WordCounter`. If building large OOP packages, you may want to emulate this underscore convention with your own package names; otherwise you can simply give your classes names like `WordCounter`.

Member variables and member function names should have their first real letter be lowercase and have word boundaries be delineated by capitalization. In addition, names that are intended to be private to the class (that is, they are used only within the class, and not by outside code) should start with an underscore. So the variable in your WordCounter class that holds the count of words might be called `wordCount` (if intended to be messed with from the outside) or `_wordCount` (if intended to be private to the class).

Another style of documenting your intent about use of internal variables is to have your variables marked as private, in general, and provide "getter" and "setter" functions to outside callers. For example, we might define a class like this:

```
class Customer{
 private var _name; // comments come here
 private var _creditCardNumber;
 private var _rating;

 /*
 * Comments come here
 */
 function getName (){
  return($this->_name);
 }
```

```
 function getRating (){
  return($this->_rating);
 }

 function setRating($rating){
  $this->_rating = $rating;
 }
[... more functions ]
}
```

3.1. Indenting, whitespace, and line length

Code is much easier to read if you use indentation to indicate the relationship among lines of code that are tied together in a common functional block, as well as whitespace to logically group elements.

Another issue is the number of spaces to indent each new code block—some people insist that two saves space, others swear by four, and some outliers actually employ eight-space indents (the horror!). If you want your code to be accepted into PEAR, it must use four-space indents. Because different editors on different platforms interpret tab characters differently, it's recommended that you use groups of four space characters in all places you would, under other circumstances, use a tab character.

Table 2. Indenting, whitespace, and line length

No	Yes

`switch ($flag) {`
`case 1:`
`doWork();`
`break;`
`case 2:`
`doOtherWork();`
`break;`
`default:`
`doNothing();`
`break;`
`}`
	`switch ($flag) {`
	
	` case 1:`
	`  doWork();`
	`  break;`
	
	` case 2:`
	`  doOtherWork();`
	`  break;`
	
	` default:`
	`  doNothing();`
	`  break;`
	`}`

3.2. Formatting control structures

Control structures—like *if*, *if/else*, *if/elseif*, and *switch* statements—can be confusing if not properly formatted. PEAR has recommended styles for all of these language constructs.

Table 3. Formatting control structures

Structure	Formatting recommendation

if Statements	`if ((condition1) && (condition2)) {`
	`  doSomething();`
	`}`

if/else Statements	`if((condition1) && (condition2)){`
	`  doSomething();`
	`} else {`
	`  doSomethingElse();`
	`}`
	
	The `else` appears on the same line as the closing bracket that terminates the `if` block.

if/elseif Statements	`if((condition1) && (condition2)){`
	`  doSomething();`
	`} elseif {`
	`  doSomethingElse();`
	`}`

switch Statements	`switch ($flag) {`
	
	` case 1:`
	`  doWork();`
	`  break;`
	
	` case 2:`
	`  doOtherWork();`
	`  break;`
	
	` default:`
	`  doNothing();`
	`  break;`
	`}`

3.3. Formatting functions and function calls

Much of PHP is concerned with defining functions, then making calls to them; and obviously code libraries like PEAR will be almost all functions. Properly formatting your functions can make it more obvious what's going on and can therefore make debugging and maintenance easier.

The PEAR style rules mandate that functions be defined with both their beginning and ending braces flush with the left margin, like this:

```
function myFunction(){
// Function code goes here.
}
```

Personally, I prefer another variant that seems to be more complete for me and also save space.

```
/*
 * Comments goes here
 */
function myFunction(){
 // Function code goes here.
}
```

3.4. PHPDoc

For very large and complex programs, code-embedded comments are not sufficient. You want separate documentation that someone can read without delivering into the code itself.

For example, if you have followed a given commenting convention, you can point the `javadoc` tool at your Java code and it will extract class and method comments into a set of HTML pages documenting the API. This is a solution for the problem of keeping docs in sync with code. (It will break down, for example, if people begin writing new methods by copying old methods, and leaving the original comments in place.) But at least developers have to write only one description of a given method rather than two.

There is an analogous `phpdoc` tool that uses PHP (naturally) to scan PHP code for special comments, producing HTML output. For more on `phpdoc`, see www.phpdoc.de/.

Bibliography

PHP web site, http://www.php.net

PEAR web site, http://www.pear.php.net

Tim Converse,Joyce Park, Clark Morgan, PHP5 and MySQL Bible, Wiley Publishing, Inc., 2004.

David Sklar, Learning PHP 5, O'Reilly 2004.

David Lane, Hugh E. Williams , Web Database Application with PHP and MySQL, 2nd Edition, O'Reilly 2004.