PHP v5 Gets Serious About Objects—Part One

By Scott Courtney

IN case anyone hasn't seen my previous articles, I'm a bit of a PHP advocate. I'm not rabid about it, but I've found PHP to be one of the most useful languages around, not only for web sites but also for general-purpose scripting. PHP especially shines when you need to access the APIs of other systems, such as the LDAP or the GD or ImageMagick graphics libraries. It's easy to use, well documented, and very fast. I like it better than Perl or C (personal preference; your mileage may vary) for web work, and I use it for general-purpose scripting simply because I like being able to reuse code from my web applications within the cron jobs and other shell-invoked utilities that support those apps.

All that being said, you may be surprised to learn that PHP is not my favorite language. Java, the fully object-oriented language from Sun Microsystems, holds that honor. The reasons for this are very simple: PHP's handling of object-oriented (OO) programming has been anemic at best. With version 5.0, recently released, PHP's object-oriented features have beeen vastly improved. In this article, we'll take an in-depth look at the new features and why they are important. Rather than reinventing the proverbial wheel, PHP's developers are "seeing further by standing on the shoulders of giants." In this case, one of those giants is Java, and programmers used to Java will notice many familiar features in the new PHP.

This is Part One of a two-part article; the second installment will be published next issue.

REVIEW OF OO CONCEPTS

Object-oriented programming (OOP) is an approach that attempts to make the structure of a program closely follow the way the human mind organizes knowledge, extrapolating from a general concept to more specific concepts. For example, if you know how to drive an automobile and a pickup truck, you will have very little "learning curve" if you get behind the wheel of a minivan. The reason is that the mind groups these four-wheeled motor vehicles into a more generic knowledge category, or "class," and for each individual case adds only the knowledge that differs from the others. In OOP, your code works the same way by defining a parent "superclass" for each generic object type (such as Four Wheeled Vehicle) and then defining child "subclasses" for each individual object type (Minivan, PickupTruck, Sedan). The subclasses "inherit" all of the code from the superclass, except where the subclass explicitly defines new functionality or overrides the parent's implementation of a function. This is the concept of "inheritance."

Another important OO concept is "encapsulation." The notion of "information hiding" is important when developing modular, reusable software components. The idea is that each module exposes only the "public" parts of itself, revealing what it can do but not how the functionality is implemented internally. This insulates other parts of a system from internal changes to fix bugs or improve performance, and it facilitates team programming because the coupling between modules is looser. Encapsulation also means that the code to implement a feature, as well as the state variables for each "instance" of an object of that type, are contained within the object module itself. An object's behavior and the information about each particular object of that class are unified, not separated.

The third fundamental OO concept is "polymorphism." Put simply, this means that an instance of a subclass can be safely treated as if it were an instance of any of its ancestor superclasses. For example, suppose we have a generic class "Vehicle" with subclasses of "WheeledVehicle," "Aircraft," and "Boat." The Vehicle class understands and implements only behaviors that are common to *all* vehicles, such as keeping track of their speed, location, direction, and perhaps a few other things. Aircraft adds the notion of altitude and climb/descent rate, something not applicable to most other vehicle types. Polymorphism means that if I create an instance of Aircraft, I can pass a reference for that instance to a function that only knows about Vehicles, and the function will accept that reference and behave sensibly. Polymorphism is extremely important, because it allows subclasses to be used to extend the behavior of existing application classes without having to modify the application itself to understand the new subclasses. An off-the-shelf application that works with the Aircraft class will also accept my new custom Helicopter subclass of Aircraft without complaining.

PHP has supported inheritance and polymorphism for quite some time, but until now it has been weak in the area of encapsulation. In addition, PHP has lacked some of the nice convenience features of Java and other fully object-oriented language. Version 5 of PHP goes a long way toward closing the gap.

Traditionally, Java programmers speak of "methods" where PHP programmers speak of "functions." This is largely due to the fact that Java is a pure OO language, whereas PHP can be used OO or procedurally. In the newer PHP 5 documentation, the term "method" is beginning to appear for functions that occur inside an object class. This article will use the terms interchangeably, as is common practice in the PHP world.

VARIABLE AND FUNCTION VISIBILITY

Before version 5, PHP had no way of hiding variables or functions inside an object class. In other words, the entire notion of an object exposing only what it can do but not how it does so was a matter of mutual agreement between the class developer and the programmer who uses the class in an application. It has become common practice for PHP programmers to use an underscore (_) at the beginning of a variable or function name to indicate that applications should not directly refer to that entity. Comments in the code also are used to indicate this, but the class creator has no way of ensuring that the class user will read the comments. In PHP 5, new language keywords allow the creator of a class to hide or expose variables as appropriate to the application.

The visibility keywords are **public**, **protected**, and **private**, and they essentially work as they do in Java. As you would expect, a **public** entity can be accessed from inside or outside the object class without restriction, and a **private** entity can be accessed only within the class itself. Entities that are marked **protected** can be accessed from within the class itself or from within any subclasses (or sub-subclasses, etc.) of that class.

FIGURES 1 and 2 illustrate how a simple class differs between PHP 5 and earlier versions.

Protected variables are a sort of intermediate level of trust, a way to expose internal features in a limited way so that subclasses can be more efficiently implemented. Private variables are invisible outside their declaring class and cannot even be accessed by subclasses.

In PHP 5, the old style declaration of "var $some_name" is no longer valid. For backward compatibility, it is treated as "public $some_name" with a warning issued. PHP 5 also includes support for visibility declarations on methods (functions), which works the same way as on variables.

Subclasses can broaden the visibility of methods they override, but cannot tighten it. That is, if the parent method is public then the subclass must also be public, whereas if the parent method is protected then the subclass method can be protected or public, but not private—and so on.

CONSTRUCTORS AND DESTRUCTORS

In PHP 5, constructors (the functions invoked as an object instance is created) have changed significantly, and destructors (functions invoked as an instance is destroyed) are a new feature.

Webliography

PHP Home Page http://www.php.net/
PHP Online Manual (English) http://www.php.net/manual/en/
PHP 5 Objects http://www.php.net/manual/en/language.oop5.php

Constructors used to be defined by declaring a function within a class whose name was the same as the name of the class. Now, however, the correct method is to define a function called __construct() (with two leading underscores in its name). As with the old constructors, the parameter list can be empty or can have any number of required and/or optional parameters. Also, the behavior of PHP with regard to parent constructors has not changed: they are still not automatically invoked. This is an important difference between PHP and Java. PHP does, however, provide the ability to call the parent constructor using this syntax:

```
parent::__construct( ... parameters here, if any ... );
```

Simply put this statement within the constructor of the subclass. The new __construct() syntax makes it easier to refactor large class libraries, because previously one would have had the name of each class' parent appearing in the parent constructor call. Now, the parent constructor call is truly generic and does not need to be edited if the class gets a different parent.

Destructors are functions named, appropriately enough, __destruct() (again, with two leading underscores). They are invoked automatically by PHP when all references to the object instance have been removed or when an object is explicitly destroyed. Destructor functions are useful for providing unconditional cleanup behaviors, such as writing messages to a log or closing files, network sockets, database connections, and so on. As with constructors, the parent destructor is *not* automatically invoked, but you can use the syntax:

```
parent::__destruct();
```

to accomplish this in your code if needed. In general, invoking parent constructors and destructors in your code is considered good form, unless you have a good reason not to do so because of unwanted side effects in the parent class' implementation of these functions.

STATIC VARIABLES AND FUNCTIONS AND OBJECT CONSTANTS

By default, most variables belong to, and functions act upon, an instance of an object class, rather than upon the class itself. The **static** keyword allows you to override this, creating variables that exist only once per class regardless of how many instances of that class are created.

A static function is one that can be safely called without any instances of the object existing, but which is restricted from accessing an implicit instance using **$this**. Static functions are often used for utility or convenience features that do not depend on an instance of the class. For example, a class having to do with databases might have a generic static function that creates an SQL clause such as " IN (*value1, value2* ....)" given an array of values. That function stands alone and does not depend on a database connection because it does not in itself act on the database; ergo, it is a good candidate for a static function.

Static variables and functions are declared simply by adding the **static** keyword immediately before their name and after any other modifiers such as **protected** or **public**.

Static functions are invoked using the scope resolution operator, ::, rather than the dereference operator, ->.

Object constants are closely related to static variables, except that they are defined and initialized at compile time, and they do not use the "$" symbol.

COMING UP NEXT

Next time, we will examine several new PHP features that support more robust OO design, moving beyond basic classes to more advanced OO concepts.

NaSPA member Scott Courtney is a senior engineer with Sine Nomine Associates, an engineering consulting company. His career has included fifteen years in engineering and IT at a large manufacturing company. He also worked as a technical journalist and editor for an online publisher for one year. Scott is an active open source developer in both PHP and Java languages and maintains a number of production Web sites using open source tools.

FIGURE 1: IMPLEMENTING A PROTECTED VARIABLE THE OLD WAY. THE LAST LINE OF CODE WILL GENERATE NO COMPILER ERROR, BUT CODE LIKE THIS MAY CAUSE THE MYCLASSOLD OBJECT TO MALFUNCTION BECAUSE THE DEVELOPER OF THAT CLASS DIDN'T PLAN FOR DIRECT ACCESS TO $_WIDGET_COUNT BY APPLICATIONS.

```
class MyClassOld extends YourClass {

    // This variable is public
    var $database_name;
    // This one is private or protected by convention, but not enforced as such by PHP
    var $_widget_count;

    // This is how apps are *supposed* to obtain the count of widgets
    function getWidgetCount() {
        return $this->_widget_count;
    }
}

// Create an object
$myobject =& new MyClassOld();
// So far no problem...access a public variable
$myobject->database_name = "mysql://joeuser:bigsecret@localhost/accountingdb";
// But now things go awry, as we access the private variable. ** WRONG **
print "We have made " . $myobject->_widget_count . " widgets so far.\n";
```

FIGURE 2: A PROTECTED VARIABLE IN PHP 5. PROTECTION IS ENFORCED BY THE COMPILER, SO THE LAST LINE WILL GENERATE AN ERROR DURING COMPILATION.

```
class MyClassNew extends YourClass {

    // This variable is public
    public $database_name;
    // This one is protected by the PHP runtime
    protected $widget_count;

    // This is how apps obtain the count of widgets
    public function getWidgetCount() {
        return $this->widget_count;
    }
}

// Create an object
$myobject =& new MyClassNew();
// So far no problem...access a public variable
$myobject->database_name = "mysql://joeuser:bigsecret@localhost/accountingdb";
// This line will generate a compile-time error. ** WRONG **
print "We have made " . $myobject->widget_count . " widgets so far.\n";
```