The Object Oriented Improvements of PHP 5

By W. Jason Gilmore

While PHP 4 was the first version to offer Object-Oriented Programming (OOP) capabilities, many considered the feature to be poorly designed and more an afterthought than anything else. PHP 5 resolves many of the version 4 OOP inconsistencies, not to mention greatly enhances the language's object-oriented capabilities. While you're still free to program using a procedural methodology, those of you with object oriented backgrounds will certainly feel much more at home creating fully object-driven PHP applications. In this article I'll introduce many of these new and improved features. In addition, for those of you somewhat new to the subject, this article will also offer a bit of instruction regarding those concepts key to truly understanding OOP.

Private and Protected Members

Object-oriented programming is largely about the ability to hide what's not important to the user and to highlight what is. One of the primary means for doing so is through defining scope for class properties. PHP 4 offered no standardized means for specifying the variety of property scopes typically offered by full-featured OO languages. PHP 5 remedies this in part through support for two common scopes:

Private: Private members are accessible within the class in which they are created. Note that they are not accessible by any subclass!

Protected: Protected members are accessible within the class in which they are created, as well as by any subclass.

In the following example, we'll create a class which contains both a private and a protected member.

```
<?php
   class boxer {
      // Not available to streetfighter class
      private $age;

      // Available to streetfighter class
      protected $wins;

      function __construct() {
         $this->age = 15;
      }

      // Return boxer's wins
      function getWins() {
         return $this->wins;
      }

      // Return boxer's age
      function getAge() {
         return $this->age;
      }

   }
```

```
   // Extend the boxer class
   class streetfighter extends boxer {

      protected $teeth;

      function __construct() {
         // This works
         $this->wins = 12;
         // This does not
         $this->age = 30;
      }

   }

   $sf = new streetfighter();
   echo $sf->getWins();

?>
```

One oddity regarding misuse of private members is the lack of any error message. Rather, attempts to set or get them will simply be ignored.

Note that any member not assigned a scope of private or protected defaults to the public scope, meaning that it's directly accessible from the caller! Because such practice generally goes against the goals of OOP, its advised that you designate such members as private and manage them through getters and setters (otherwise known as properties). Unfortunately, a unified getter/setter methodology has yet to be worked into the language, leaving you to your own devices to come up with a method-based solution.

Final, Private, Protected and Static Methods

Like members, it's often useful to set various levels of scope for methods. PHP 5 defines four previously unavailable levels: final, private, protected and static. I'll discuss each in this section.

Final Methods

While methods which have been declared as final can be used within subclasses, but cannot be overridden. Let's consider an example:

```
<?php
class logger {

   final function getCredits() {
      return "Copyright 2004 you know who.";
   }

}

class myLogger extends logger {

   function getCredits() {
      return "Give credit where credit is due!";
   }
```

```
}

$mylog = new myLogger();
$mylog->getCredits();

?>
```

Executing this example results in the following fatal error:

Fatal error: Cannot override final method logger::getCredits() in /www/htdocs/developer/finalmethod.php

Private Methods

Like private members, private methods are only accessible within the class in which they are declared. An example follows:

```
<?php
   class boxer {

      protected $bouts;
      protected $wins;
      protected $losses;

      private function winPercentage() {
         return ($wins / $bouts) * 100;
      }

   }

   class streetfighter extends boxer {

      function __construct() {
         $this->bouts = 10;
         $this->wins = 5;
      }

      function stats() {
         return $this->winPercentage();
      }

   }
?>
```

Like private members, attempts to access private methods from outside of the originating class will simply result in no action, rather than any error message.

Go to page: 1 2 3 Next

Protected Methods

Like protected members, protected methods are accessible both within the class in which they are declared, as well as by all subclasses.

Static Methods

Most applications make use of various functions of use throughout the entire application. Because such functions aren't necessarily related to any particular object, they're often placed in a general utility class. However, such a strategy is followed because it's good OO programming practice, and not because we want to invoke a "utility" object (although you could). Rather, we just want to call the method as necessary, while still managing to encapsulate it in some sort of class. Class methods that can be called without first instantiating an object are known as static. Let's consider an example:

```
<?php
   class logger {

      public static function getCredits() {
         return "Copyright 2004 you know who.";
      }

   }

   echo logger::getCredits();
?>
```

Note that you cannot reference the $this keyword within static methods, because there was no prior object instantiation.

Abstract Methods

An abstract method is intended as a cue for any developer interested in implementing a specific class. While an abstract method doesn't define the method's implementation, it acts as a guide to the developer, informing him that he will have correctly implemented the class provided that it and all other abstract methods found in the class are implemented. An example is provided in the next section.

Abstract Classes

The object-oriented development strategy is particularly appealing because we easily relate to it. Humans are quite adept at thinking in terms of hierarchical structures. For example, its obvious to all of us that compact discs, MP3s and MPEGs all fall under the category of media. However, media is solely an abstract concept which we use to generalize a group of items that share a set of common characteristics which we collectively define as media.

Its often convenient for programmers to embody these shared characteristics and behaviors in an abstract class. Class abstraction affords the programmer the convenience of wielding control over a core set of properties and behaviors that he knows will be shared by a number of similar, yet unique classes. While an abstract class is never directly instantiated, any class which inherits it will also possess its properties and behaviors.

PHP 5 supports class abstraction. As an example, let's create the media class. This class holds one property, *$copyrightInfo*, one public function, setCopyrightInfo(), and one abstract function, play(). While this class in useless by itself, it can readily be inherited by any of our specific media types.

```
<?php
   abstract class media {
```

```
      private $copyrightInfo;

      abstract function play();

      function setCopyrightInfo($info) {
         $this->copyrightInfo = $info;
      }

   }

   class cd inherits media {
      function play() {
          echo "The CD has started playing!";
      }
   }

   $cd = new cd();
   $cd->setCopyrightInfo("Copyrighted by who else? Me!");
   $cd->play();

?>
```

Interfaces

Interfaces bring an additional level of both convenience and discipline to object-oriented programming by providing a means for blueprinting an application component, defining the methods which together make that component what it is. A class is said to implement an interface provided that it satisfies all of its defining characteristics.

To the great pleasure of many developers, PHP 5 offers interfaces! Let's consider an example:

```
<?php

   interface fighter {
      function power();
      function weight();
   }

   class boxer implements fighter {

      private $power;
      private $weight;

      function power() {
         return $this->power;
      }

      function weight() {
         return $this->weight;
      }

}
?>
```

A class can also simultaneously implement multiple interfaces. Let's consider another example:

```
<?php

   interface fighter {
      function power();
      function weight();
   }

   interface kickboxer {
      function kick();
   }

   class warrior implements fighter, kickboxer {

      private $power;
      private $weight;

      function power() {
         return $this->power;
      }

      function weight() {
         return $this->weight;
      }

      function kick() {
         return "kick!";
      }

}

?>
```

Failing to implement all of the functions as defined within an interface results in an error like so:

Fatal error: Class warrior contains 1 abstract methods and must therefore be declared abstract (kickboxer::headbutt) in /www/htdocs/developer/interface.php on line 30

Unified Constructors

While PHP 4 offered constructors, the syntax required that the constructor method be the same name as the class. PHP 5 resolves this inconvenience by providing a standardized constructor syntax, using the name __construct(). For example:

```
<?php

   class blog {

      private $title;

      function __construct() {
         $this->title = "Just another rant";
         echo "New blog object created!;
      }
   }

   $blogobj = new blog();
```

```
?>
```

Executing this code results in the assignment of the $title member, and output of the following message:

New blog object created!

Go to page: Prev 1 2 3 Next

Unified Destructors

Good programming practice dictates that you properly clean up all resources once you're done with them. This may involve closing a database connection, writing information to a log file, or anything else you deem appropriate. Prior to PHP 5, you were forced to devise your own methodology for dealing with such matters once work with an object is complete, resulting in inconsistent coding practice. PHP 5 resolves this with the introduction of unified destructors.

Like the new constructor syntax, destructors are called like so: __destruct(). Consider this example:

```
<?php
  class blog {

    private $title;

    function __construct() {
      $this->title = Just another rant...;
    }

    function __destruct() {
      $fh = fopen(bloglog.txt, a+);
      $date = date("Y-m-d at h:i:s");
      fwrite($fh, "New blog entry: $this->title. Created on $date\n");
      fclose($fh);
    }
  }

  $blogobj = new blog();
?>
```

Executing this script will result in a new object of type blog being created, and its $title property assigned the value "Just another rant...". Next, a handle to the file "bloglog.txt" will be opened, and an entry similar to the following will be added to it:

New blog entry: Just another rant.... Created on 2004-01-20.

Once appended, the filehandle is closed.

Class Constants

It's often useful to define constants on a per-class basis. This is now possible in PHP 5. For example, suppose that you wanted to define a standard thumbnail height and width for an image gallery class:

```
<?php

  class imagegallery {

    const THUMB_HEIGHT = 120;
    const THUMB_WIDTH = 120;

    private $height;
    private $width;

    function create_thumbnail($height="",$width="") {
      if (!isset($height)) $this->height = THUMB_HEIGHT;
      if (!isset($width)) $this->width = THUMB_WIDTH;
    }
  }

  echo "Default thumbnail height: ".imagegallery::THUMB_HEIGHT;
?>
```

These constants are only relevant to the imagegallery class, allowing you to reuse the terms "HEIGHT" and "WIDTH" within other classes or scripts comprising the application. Furthermore, as I demonstrate at the conclusion of the above example, you can reference the class constants from outside of the class by prefacing their name with the class name (not a class instance!) and double-colons.

Conclusion

PHP 5 offers major enhancements to its object oriented feature set, which certainly only adds to its allure as a powerful Web scripting language. However, these improvements are only the tip of the iceberg in regards to what PHP 5 has to offer. In the next installment of this two-article series, I'll round out the many features that users can expect from the forthcoming release.

I welcome questions and comments! E-mail me at jason@wjgilmore.com. I'd also like to hear more about your experiences experimenting with PHP 5!

About the Author

W. Jason Gilmore (http://www.wjgilmore.com/) is an Internet application developer for the Fisher College of Business. He's the author of the upcoming book, PHP 5 and MySQL: Novice to Pro, due out by Apress in 2004. His work has been featured within many of the computing industry's leading publications, including Linux Magazine, O'Reillynet, Devshed, Zend.com, and Webreview. Jason is also the author of A Programmer's Introduction to PHP 4.0 (453pp., Apress). Along with colleague Jon Shoberg, he's co-author of "Out in the Open," a monthly column published within Linux magazine.

Go to page: Prev 1 2 3

A Look at PHP 5, Part II

By W. Jason Gilmore

In the first article of this two-part series, we examined what is arguably the most significant feature of PHP 5; the significant advances to its object-oriented programming capabilities. Among other topics, I introduced the addition of the final,

private, protected, abstract and static method scopes, as well as the new private and protected class member scopes. We also examined the new abstract class and interface features, before concluding the article with a look at the new unified constructors and destructor feature, in addition to an introduction to the ability to add class-specific constants. Yet despite all of this, the new OOP features are just a smattering of what's available in PHP 5. In this article, we'll examine several other cool new extensions and features available to this major release.

In particular, we'll take a look at the addition of three major features: The default packaging of the SQLite and SimpleXML libraries, and the addition of exception handling. Each in their own right, all three features stand to contribute significantly to your PHP programming repertoire.

SQLite

SQLite (http://www.sqlite.org/) is a lightweight SQL database engine which offers a surprising number of advanced features. Although surprisingly small (for example, the Windows binary is 275kb in size), SQLite is largely SQL-92 compliant, supports ACID transactions, and is capable of working with databases up to 2 terabytes in size. Perhaps due both in part to the unbundling of the MySQL client library and the realization that the scope of many projects simply did not require the firepower provided by the traditional database servers, the decision was made to enable support for the SQLite extension by default.

To use PHP's SQLite extension, you'll need to download and install the package, available via the URL referenced in the previous paragraph. Once installed, you can create a new database via its command-line interface. This interface is quite similar to those available to MySQL and PostgreSQL. To create a new database, just pass its name as an argument to the sqlite command. Next, you can create a table and insert a few rows using commands again quite similar to MySQL and PostgreSQL. I'll demonstrate this process here:

```
%>sqlite library.db
sqlite>CREATE TABLE book (
...>rowID INTEGER PRIMARY KEY,
...>title VARCHAR(55),
...>author VARCHAR(55));
sqlite>INSERT INTO books VALUES(NULL,"Practical Python","Magnus
Hetland");
sqlite>INSERT INTO books VALUES(NULL,"MySQL","Michael Kofler");
sqlite>.quit
```

Next, let's use PHP's SQLite library to extract this information from the newly created "library.db" database:

```
<?php
   $sqldb = sqlite_open("/www/sqlitedatabases/library.db");
   $results = sqlite_query($sqldb, "SELECT title,author FROM book");
   while (list($title, $author) = sqlite_fetch_array($results)) {
        echo "Title: $title (Author: $author) <br />";
   }
   sqlite_close($sqldb);
?>
```

Returning:

```
Title: Practical Python (Author: Magnus Hetland)
Title: MySQL (Author: Michael Kofler)
```

For those of you with experience working with any PHP-compatible database, you'll find the above syntax to be quite familiar. Indeed, it matches almost exactly that used by PHP's MySQL library, with one significant difference; no authentication is required! This is because SQLite's permission framework is entirely dependent upon the permissions assigned to the file within which the database is stored. Therefore if the library.db file possesses adequate read-permissions, then it can be read via a PHP script. If it possesses adequate write-permissions, then SQLite insertion/update/deletion queries can be used, and so on. Indeed this convenient and effective permissions system works exceptionally well for users interested in keeping administration overhead to a minimum.

Prior to its packaging with PHP, I had no prior experience with SQLite, but have since used it for several internal projects. Fast, capable, and amazingly easy to configure and use, SQLite is definitely worth checking out. Learn more about SQLite via the PHP manual:
http://www.php.net/sqlite

SimpleXML

SimpleXML is one of the newer and resultingly, lesser-known extensions to work its way into the PHP 5 distribution, available by default as of the Beta 3 release. However, I suspect that its relative obscurity will soon change, given the extraordinarily convenient solution it offers to a problem that has long nagged developers: XML parsing. True to its name, SimpleXML works by loading an XML document into an object, allowing you to then dereference an XML node in a fashion quite similar to dereferencing an class member. Consider the following XML document:

Listing 1-1: A typical RSS feed (gilmore.xml)

```
<?xml version="1.0" encoding="utf-8" ?>
<rss version="0.91">
<channel>
   <title>W. Jason Gilmore</title>
   <link>http://www.wjgilmore.com/</link>
   <description>Ramblings of a computer maniac...</description>
</channel>
<item>
   <title>Debugging NuSOAP</title>
   <link>http://www.wjgilmore.com/archives/000064.html</link>
   <description>Significantly reduce your PHP/NuSOAP debugging
time!</description>
</item><item>
   <title>C# Patterns</title>
   <link>http://www.wjgilmore.com/archives/000058.html</link>
   <description>Found an excellent software patterns site this
evening.</description>
</item>
</rss>
```

You might recognize this as a typical RSS feed. While there are indeed numerous solutions available for parsing RSS feeds, there aren't too many that can do it as easily as SimpleXML:

```
<?php
   $xml = simplexml_load_file("gilmore.xml");
```

```
   echo $xml->channel[0]->title;
?>
```
Returning: W. Jason Gilmore It doesn't get much easier than that, does it? Let's consider a somewhat more practical example, using the same RSS feed. This time, we'll convert the RSS entries to an XHTML-equivalent:
```
<?php
  $xml = simplexml_load_file("gilmore.xml");
  echo "<strong>".$xml->channel[0]->title."</strong><br />";
  foreach($xml->item as $item) {
    echo "<a href=\"$item->link\">$item->title</a><br />";
  }
?>
```
Returning:
```
<strong>W. Jason Gilmore</strong><br />
<a href="http://www.wjgilmore.com/archives/000064.html">Debugging
NuSOAP</a><br />
<a href="http://www.wjgilmore.com/archives/000058.html">C#
Patterns</a><br />
```

Learn more about this promising new extension via the PHP manual: http://www.php.net/simplexml

On an aside, in a previous tutorial I demonstrated how to create a custom RSS feed aggregator using PHP, MySQL and Magpie. The article can be found here.

Exception Handling

Many modern languages implement a common error detection and reporting methodology. One of the longstanding PHP annoyances has been the lack of such a feature. I'm happy to announce the PHP 5 resolves this problem, offering an "try-catch" exception model similar to that found in languages such as Python, Java and C#. Let's demonstrate this new feature, revising the previous SimpleXML example to include exception handling.

```
<?php
try {
   $xml = simplexml_load_file("gilmore.xml");
   if (!$xml) throw new Exception("Could not read RSS feed!");
   echo "<strong>".$xml->channel[0]->title."</strong><br />";
   foreach($xml->item as $item) {
      echo "<a href=\"$item->link\">$item->title</a><br />";
   }
} catch (Exception $e) {
   echo "Exception: ".$e->getMessage()." on line ".$e->getLine();
}
?>
```
As you can see, incorporating exception handling into our code can greatly increase its readability. Assuming that the simplexml_load_file() function successfully loads the XML file, the feed contents will be parsed and output accordingly, otherwise the following message will be output:
```
Exception: Could not read RSS feed! on line 13
```

Extending the default Exception handler

You can create your own custom exception handlers by extending the default Exception class. Let's revise the above example, this time using our own custom class:

```
<?php
class RSSException extends Exception {

   private $feed;
   protected $message;

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

   function getFeed() {
      return $this->feed;
   }
}

try {
   $feed = "gilmore.xml";
   $xml = simplexml_load_file($feed);
   if (!$xml) throw new RSSException($feed, "Could not read RSS
feed!");
   echo "<strong>".$xml->channel[0]->title."</strong><br />";
   foreach($xml->item as $item) {
      echo "<a href=\"$item->link\">$item->title</a><br />";
   }
} catch (RSSException $e) {
   echo "Exception: ".$e->getMessage().". Feed name: ".$e->getFeed();
}

?>
```

Assuming there's a problem, the following output will occur:

```
Exception: Could not read RSS feed!. Feed name: gilmore.xml
```

Note that there is currently no official documentation pertinent to PHP 5's new exception handling feature, although I suspect that it will soon be made available.

Conclusion

I welcome questions and comments! E-mail me at jason@wjgilmore.com. I'd also like to hear more about your experiences experimenting with PHP 5!

About the Author

W. Jason Gilmore is an Internet application developer for the Fisher College of Business. He's the author of the upcoming book, *PHP 5 and MySQL: Novice to Pro*, due out by Apress in 2004. His work has been featured within many of the computing industry's leading publications, including Linux Magazine, O'Reillynet, Devshed, Zend.com, and Webreview. Jason is also the author of *A Programmer's Introduction to PHP 4.0* (453pp., Apress). Along with colleague Jon Shoberg, he's co-author of "Out in the Open," a monthly column published within Apress' dot-dot magazine.

Previous article: The Object Oriented Improvements of PHP 5

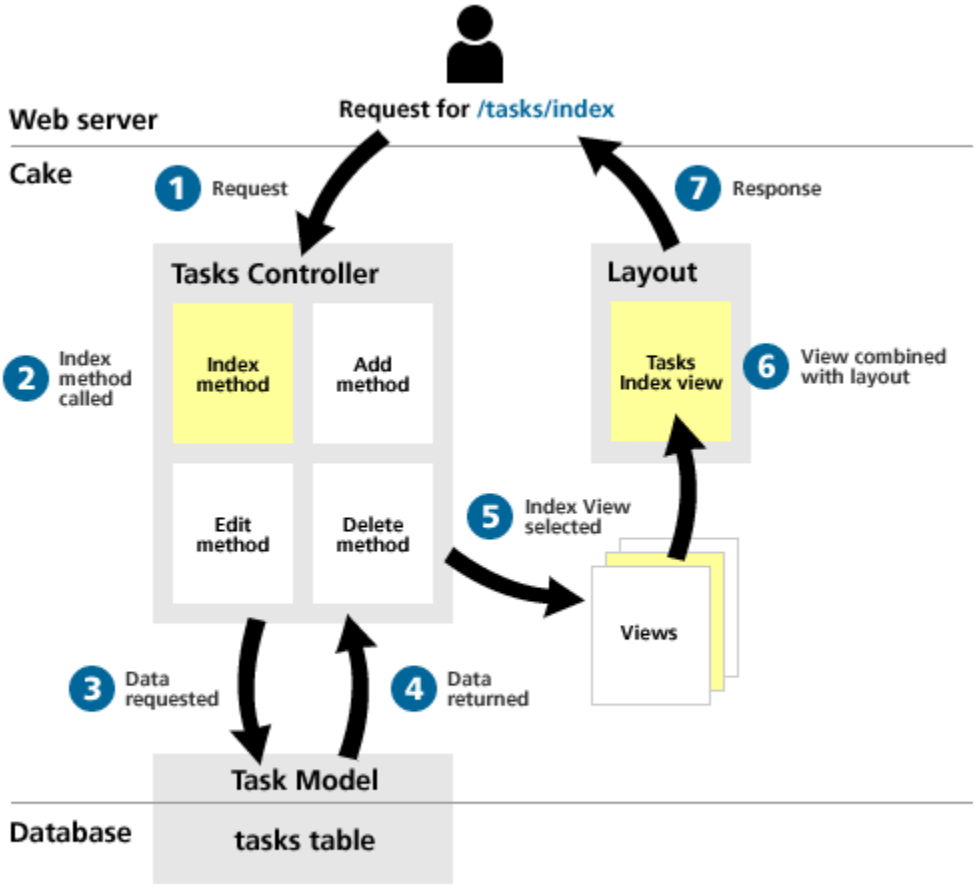
How Cake works

A representation of Cake in action

Quick guide to CakeGlossaryTutorialsLinks

The illustration below is a basic representation of how Cake handles incoming requests.

Note: It doesn't show the "invisible" elements (such as the dispatcher) and is deliberately simplified, but hopefully it will help you understand how the code works together.



Comments and feedback

http://www.freeopenbook.com/php-hacks/toc.html